



# UNIVERSIDAD DE GRANADA

Universidad de Granada

Master en ciencia de datos e ingeniería de computadores

Trabajo de Fin de Master

---

Separación de cascadas electromagnéticas en el  
experimento SBND mediante técnicas de Machine Learning

---

Autor:

**Alejandro Ponce Miguela**

Tutores:

**Alberto Guillén Perales**

**Bruno Zamorano García**

Departamento de Arquitectura y Tecnología de Computadores

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicaciones

4 de julio de 2022

Don ALEJANDRO PONCE MIGUELA con D.N.I. 30235577, estudiante de master en la Universidad de Granada y autor del trabajo titulado Separación de cascadas electromagnéticas en el experimento SBND mediante técnicas de Machine Learning:

DECLARO

Que el trabajo fin de grado que he presentado para su evaluación es original y de elaboración personal, lo que implica la no reproducción de fragmentos de obras no amparados por el límite de cita, regulado en el artículo 32 de la Ley de Propiedad Intelectual, y no copio ni utilizo ideas, formulaciones, parafraseo, etcétera, tomadas de cualquier obra, sin expresar de manera clara su origen tanto en el cuerpo del trabajo como en su bibliografía.

De no cumplir con este requisito propio de cualquier trabajo académico, soy plenamente consciente que ello conllevará automáticamente la calificación de NO APTO y, en su caso, podré ser objeto de sanción académica, previa apertura de expediente disciplinario o de otro orden legal.

Sevilla, a 4 de julio de 2022.



Don ALEJANDRO PONCE MIGUELA

## Resumen

Medidas anómalas en distintos experimentos de neutrinos sugieren la existencia de una nueva partícula no contemplada por el Modelo Estándar: los neutrinos estériles. Actualmente, se están desarrollando varios experimentos, como SBN o DUNE, para confirmar o no la existencia de esta nueva partícula. Determinar la partícula que origina una cascada electromagnética es fundamental para la detección de neutrinos en los detectores de proyección temporal con argón líquido (LArTPC), detectores que usan los experimentos mencionados. Sin embargo, aún no se tienen métodos que discriminen los eventos con suficiente precisión. Nosotros abordamos este problema mediante la creación de imágenes y el uso de redes neuronales convolucionales para determinar su origen. Además, presentamos distintas formas de representar los datos para que los modelos obtengan mejores resultados. Por otro lado, debido a su gran importancia, la solución se desarrolla haciendo uso de técnicas MLOps, que aun no siendo habituales en la investigación han mostrado ser valiosas cuando se tienen distintas etapas en el desarrollo. Nuestro trabajo proporciona un estudio de cómo afectan a los modelos distintas maneras de tratar los datos, tanto al crear como al preprocesar las imágenes. Adicionalmente, presentamos un modelo final que clasifica las cascadas electromagnéticas con una pureza de un 96.2%.

**Palabras claves:** Deep Learning, clasificación de imágenes, CNN, neutrino estéril, SBND.

## Abstract

Anomalous measurements in different neutrino experiments suggest the existence of a new particle not considered by the Standard Model: sterile neutrinos. Nowadays, several experiments, such as SBN and DUNE, are being developed in order to confirm this new particle existence. Identifying the particle that leads to an electromagnetic shower is fundamental for neutrino detection in liquid argon time-projection detectors (LArTPC), the ones used by aforementioned experiments. However, methods that discriminate the events with enough precision are not yet available. Our approach to this problem is based on imaging and the usage of convolutional neural networks to determine its origin. In addition, we present different ways to represent the data so that the models obtain better results. On the other hand, due to its great importance, the solution is developed making use of MLOps techniques; not commonly used by researchers, but shown to be valuable when having different stages in the development. Our work provides a study of how models are affected by different ways of handling the data, both when creating and preprocessing the images. Furthermore, we present a final model that classifies electromagnetic showers with a purity of 96.2 %.

**Keywords:** Deep Learning, images classification, CNN, sterile neutrino, SBND.

# Índice general

<b>1. Introducción y objetivos</b>	<b>1</b>
1.1. Objetivos . . . . .	3
<b>2. Fundamentación: Conceptos y estado del arte</b>	<b>4</b>
2.1. Motivación física . . . . .	4
2.1.1. Cascada electromagnética . . . . .	8
2.1.2. Detector LArTPC . . . . .	11
2.2. Conceptos de computación . . . . .	14
2.2.1. Deep Learning . . . . .	15
2.2.2. MLOps . . . . .	22
2.3. Estado del arte . . . . .	24
<b>3. Machine Learning aplicado a la clasificación de cascadas electromagnéticas</b>	<b>26</b>
3.1. Descripción del problema . . . . .	27
3.2. Preprocesamiento de los datos . . . . .	31

<b>ÍNDICE GENERAL</b>	<b>v</b>
3.2.1. Formas de representación . . . . .	31
3.2.2. Transformación de los datos a imágenes . . . . .	32
3.2.3. Comentarios finales . . . . .	35
3.3. Modelos propuestos . . . . .	36
3.4. Flujo de trabajo . . . . .	38
<b>4. Análisis de los datos</b>	<b>42</b>
<b>5. Desarrollo de los modelos para la clasificación</b>	<b>49</b>
5.1. Representaciones de los datos . . . . .	49
5.2. Arquitecturas . . . . .	56
5.3. Métricas . . . . .	61
5.4. Hiperparámetros . . . . .	62
5.5. Entorno de trabajo . . . . .	65
<b>6. Resultados y discusión</b>	<b>67</b>
6.1. Metodología de experimentación . . . . .	67
6.2. Pruebas iniciales . . . . .	68
6.3. Selección de una representación . . . . .	71
6.4. Selección de la resolución . . . . .	73
6.5. Ajuste de hiperparámetros . . . . .	76
6.6. Breve estudio del modelo B2 . . . . .	83
6.7. Concatenación de redes, 3 proyecciones . . . . .	88

<b>ÍNDICE GENERAL</b>	<b>VI</b>
<b>7. Conclusiones y perspectiva de futuro</b>	<b>93</b>
<b>A. Hiperparámetros de los estudios</b>	<b>96</b>
A.1. Estudio de la capa de normalización por <i>batch</i> y transformación logarítmica	96
A.2. Estudio de la representación . . . . .	96
A.3. Estudio de la resolución . . . . .	98
A.4. Ajuste del modelo LeNet simple . . . . .	99
A.5. Ajuste del modelo LeNet complejo . . . . .	100
A.6. Ajuste modelos preentrenados: Optimización . . . . .	100
A.7. Ajuste modelos preentrenados . . . . .	101
<b>B. Planificación y Presupuesto</b>	<b>102</b>
B.1. Planificación . . . . .	102
B.2. Presupuesto . . . . .	103

# Capítulo 1

## Introducción y objetivos

El Deep Learning se ha convertido en el estado del arte en distintos problemas de Machine Learning como son la clasificación, segmentación y detección de objetos en imágenes con el uso de redes neuronales convolucionales [1, 2]; o los problemas de procesamiento de lenguaje natural con el uso de los transformers [3, 4, 5]. En este trabajo vamos a utilizar el Deep Learning para resolver un problema de ciencia de datos aplicado a la física de partículas.

La física, al ser una ciencia experimental, siempre ha necesitado estudiar los datos recopilados de distintos experimentos. Por este motivo, la física ha utilizado constantemente herramientas de ciencia de datos. Un ejemplo es la regresión, que permite, dada una ley física, comprobar si los datos experimentales verifican el comportamiento descrito por dicha ley. En la actualidad, tenemos modelos de ciencia de datos mucho más complejos, así como problemas físicos muy complicados donde las herramientas clásicas de ciencia de datos no son suficiente. Además, se están desarrollando experimentos donde tanto la complejidad como el volumen de datos hacen que la física sea un perfecto laboratorio para probar modelos de ciencia de datos, así como para el desarrollo de nuevos algoritmos. Solo en el CERN, se generan en torno al petabyte de datos por segundo [6]. Estos problemas complejos están poniendo de manifiesto cómo el uso del Deep Learning puede mejorar considerablemente los resultados [6].

No obstante, la aplicación del Deep Learning en física supone una serie de dificultades, que también se encuentran al aplicar estos modelos en otros ámbitos. Estos modelos son

“cajas negras”, es decir, que dada una entrada no se puede entender qué hace el modelo para obtener el resultado final, dificultando su interpretación. El uso generalizado de estos modelos en diversos ámbitos ha puesto de manifiesto los problemas éticos y morales que surgen de la falta de interpretabilidad. En física, este problema no surge por problemas éticos, lo que ocurre es que no basta con saber que algo está ocurriendo, se quiere también entender el porqué de lo observado.

Este trabajo consiste en una colaboración con el departamento de Física Teórica y del Cosmos de la Universidad de Granada, donde vamos a trabajar con uno de los detectores de neutrinos del programa Short Baseline Neutrino (SBN). Nuestro objetivo ha consistido en clasificar dos tipos de eventos muy similares que pueden tener lugar en el interior de estos detectores. Al ser una colaboración, una parte del trabajo consistirá en aprender los fundamentos y principios necesarios para el correcto desarrollo del problema presentado, tal y como se planteará a continuación. Estos principios se desarrollarán de manera simple con la idea de mostrar la complejidad del problema y la necesidad de encontrar una solución.

Para la resolución de este problema hemos partido de cero, por lo que hemos tenido que decidir cómo procesar los datos. Hemos abordado esta cuestión como un problema de clasificación de imágenes, ya que a partir de estos datos podemos crear imágenes de manera natural. Además, se han planteado distintas formas de preprocessar estas imágenes y probado distintos modelos de Deep Learning para tratar de resolver el problema. Una vez se ha obtenido un modelo que dé buenos resultados, hemos usado diversas técnicas para interpretarlo.

El trabajo se ha desarrollado en Python al ser un lenguaje de programación de uso generalizado en ciencia de datos. Además, posee grandes librerías para Deep Learning como Pytorch, que es la que hemos utilizado. Por otro lado, con la intención de que este modelo pueda usarse en un futuro, así como por buenas praxis de programación, vamos a hacer uso de prácticas MLOps a la hora de implementar la solución. Una parte de este trabajo consiste en aprender y aplicar estas herramientas y técnicas.

## 1.1. Objetivos

El objetivo fundamental de este trabajo es conseguir la clasificación de los datos proporcionados, problema que actualmente no está cerrado y es fundamental para el objetivo del programa SBN [7]. Además, será necesario aprender sobre la física del problema y, por otro lado, sobre aspectos de desarrollo de aplicaciones que, como físico<sup>1</sup>, serán herramientas nuevas pero cruciales. De manera que podemos hacer la siguiente separación entre objetivos de carácter general y específico:

- (O.G. 1) Resolver el problema de clasificación de cascadas electromagnéticas en el interior de un detector LArTPC.
- (O.E. a) Aprender la física necesaria para la correcta comprensión del problema, así como para facilitar su resolución.
- (O.E. b) Proponer distintas formas de procesar los datos usando imágenes.
- (O.E. c) Sugerir distintos modelos para realizar la clasificación.
- (O.E. d) Aprender la librería Pytorch para utilizar modelos de Deep Learning.
- (O.E. e) Aprender sobre prácticas de desarrollo basadas en MLOps, así como herramientas que faciliten su implementación, como pueden ser MLFlow o Git para el control de versiones.

---

<sup>1</sup>Soy graduado en Física por la Universidad de Sevilla.

# Capítulo 2

## Fundamentación: Conceptos y estado del arte

En este capítulo vamos a mostrar los fundamentos necesarios para comprender tanto el problema propuesto, como todos los principios físicos y de computación aplicados para la resolución del problema. Al final de este capítulo se muestra el estado del arte.

### 2.1. Motivación física

Los neutrinos  $\nu$  son partículas elementales predichas de manera teórica por Wolfgang Pauli en 1930 [8] como explicación de una serie de medidas experimentales anómalas en el decaimiento  $\beta^1$ . En concreto, se observaba que los electrones emitidos tenían un espectro continuo de energías. Sin embargo, la conservación de la energía establecía que para esta reacción la energía debería ser constante [9]. La única explicación que se encontró es que en la reacción tendría que haber una tercera partícula neutra cuya interacción con la materia fuera muy poco probable, por lo que recibió el nombre de neutrino. La detección de estas

---

<sup>1</sup>El decaimiento  $\beta$  es una reacción donde un núcleo convierte un neutrón en un protón, un electrón y un antineutrino. Inicialmente, se pensaba que únicamente se emitían el protón y el electrón. Fue Enrico Fermi quién, con la idea de W. Pauli, formuló la teoría del decaimiento  $\beta$ . E. Fermi en su teoría estableció que la partícula extra de la reacción era un neutrino, pero posteriormente se vio que realmente era su antipartícula, el antineutrino [9].

partículas resultó ser muy compleja, tanto que las evidencias experimentales no llegaron hasta 1956, cuando se diseñó un experimento para detectar la reacción inversa [9].

El neutrino es lo que se denomina una partícula elemental, que son aquellas que no están formadas por otras partículas. Por ejemplo, un átomo de hidrógeno ( $H_1^1$ ) está formado por un protón y por un electrón, a su vez el protón está formado por quarks. Sin embargo, el electrón no está formado por otras partículas. El Modelo Estándar, marco teórico en el que se fundamenta la física de partículas, estructura las partículas elementales conocidas en tres grupos: quarks, leptones y bosones. Los quarks forman los hadrones<sup>2</sup>, los leptones son partículas que no sienten la interacción fuerte, como los neutrinos y los electrones, y los bosones son las partículas encargadas de mediar las interacciones, como el fotón. Dentro de los leptones tenemos tres generaciones de partículas que se diferencian fundamentalmente en su masa: electrones  $e^-$ , muones  $\mu^-$  y tauones  $\tau^-$ . Cada una de estas partículas tiene asociada un neutrino, es decir, tenemos tres tipos de neutrinos que se denominan neutrino electrónico  $\nu_e$ , muónico  $\nu_\mu$  y tauónico  $\nu_\tau$ .

La masa de los neutrinos no está determinada a día de hoy, lo que se sabe es que es distinta de cero y que sus masas son muy pequeñas (lo único que se tiene son cotas superiores en la masa [10]). Las dificultades en su detección se deben a que la probabilidad de que interactúen con el medio es muy baja, lo que hace que sean capaces de “atravesar” la materia sin perder energía. Para su detección inicial se emplearon fuentes muy intensas (partículas por unidad de tiempo) para compensar la baja probabilidad de interacción. El primer detector consistía en un tanque de agua donde se buscaba la reacción  $\beta$  inversa y los neutrinos procedían de una central nuclear [9]. Otros detectores son, por ejemplo, el Super Kamiokande [11] que se basa en la radiación de Cherenkov<sup>3</sup> o las cámaras de proyección temporal (TPC) que se usan en experimentos como SBN [7] o DUNE [13].

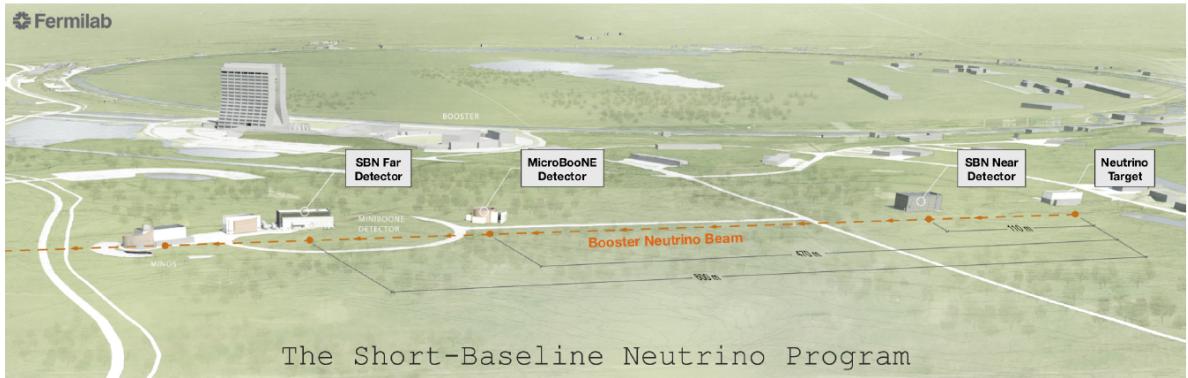
El problema de ciencia de datos que hemos tratado en este trabajo consiste en hacer uso de las medidas experimentales simuladas de un detector TPC para tratar de clasificar los neutrinos que interactúan por el detector. Los datos que hemos empleado en este trabajo proceden de simulaciones realizadas sobre el experimento SBND, que actualmente está en proceso de construcción (en secciones posteriores describiremos cómo funciona

<sup>2</sup>Los hadrones se definen como las partículas que están formadas por quarks. Por ejemplo, los neutrones y protones son hadrones.

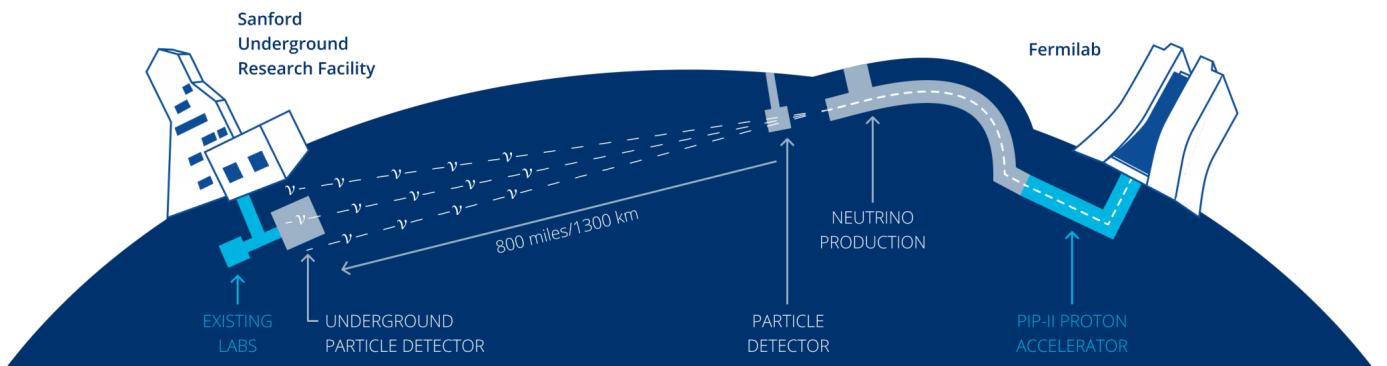
<sup>3</sup>Luz emitida por una partícula cargada al moverse por un medio dieléctrico a una velocidad superior a la de la luz en dicho medio [12].

exactamente). En las Figuras 2.1 y 2.2, mostramos de manera esquemática las instalaciones del programa SBN y DUNE.

Las fuentes de neutrinos son muy diversas. Pueden ser producidos de manera artificial, en reactores nucleares o en aceleradores de partículas, o de manera natural, en procesos como la fusión del hidrógeno en el Sol o en otros muchos eventos cósmicos, como supernovas [14]. Gracias a su débil interacción con la materia, los neutrinos son capaces de recorrer grandes distancias, por lo que muchísimos neutrinos llegan a la superficie terrestre cada segundo. Si consideramos únicamente los neutrinos solares, unos diez mil millones de neutrinos pasan por una uña cada segundo ( $7 \times 10^{10}$  partículas por segundo y por centímetro cuadrado [15]).



**Figura 2.1:** Visión esquemática del experimento SBN en Fermilab [7]. De derecha a izquierda se muestra la fuente de neutrinos y su recorrido. También se muestran los tres detectores que se usan para caracterizar el haz de neutrinos: un detector cercano (conocido como SBND), uno intermedio (MicroBooNE) y uno lejano (ICARUS).



**Figura 2.2:** Esquema del experimento DUNE en Fermilab desde una vista transversal [16]. Se muestran la fuente de neutrinos, su recorrido y los distintos detectores usados para caracterizar el flujo.

Hasta ahora hemos hablado únicamente de los distintos tipos de neutrinos que se han observado, los que recoge el Modelo Estándar. Un hecho importante es que, a medida que estos neutrinos se desplazan, estos cambian de un tipo a otro, es lo que se conoce como oscilación de neutrinos y está experimentalmente probado [17]. No obstante, hay una serie de medidas anómalas al contar el número de neutrinos en ciertos experimentos, donde se observa un exceso con respecto a los predichos teóricamente. Estas anomalías podrían explicarse introduciendo un nuevo neutrino que no siente la interacción débil [7] estando, por tanto, trabajando en física más allá del Modelo Estándar.

Tanto SBN como DUNE, son detectores que han sido diseñados con la intención de obtener suficientes evidencias de las anomalías detectadas para probarlas o refutarlas ( $5\sigma^4$ ). Ambos detectores están en construcción ahora mismo, pero se puede trabajar con simulaciones para poder ir desarrollando el software necesario para el tratamiento de los datos experimentales. Nosotros vamos a trabajar con SBND, que se espera que se ponga en funcionamiento en 2023 [18].

Ambos detectores son TPC que usan como medio argón líquido (LArTPC) y, en resumen, la idea fundamental es que los neutrinos interactúan con el argón. Hay dos mecanismos por los que se puede dar la interacción débil, que es la única interacción que sienten los neutrinos. Estos mecanismos son las corrientes neutras y las cargadas. A nosotros nos interesa la corriente cargada, ya que es la única que nos permite saber qué tipo de neutrino ha llegado al detector. En las corrientes cargadas se emite un leptón cargado, por lo que son estas las partículas que vamos a buscar en el detector para la detección de neutrinos:

$$\nu_x + n \rightarrow x^- + p^+, \quad (2.1)$$

donde  $x = e, \mu, \tau$ . Sin embargo, surgen una serie de dificultades. Por un lado, los electrones dan lugar a lo que se conoce como cascada electromagnética, que se empleará para determinar si tenemos un neutrino del electrón. El problema es que los fotones pueden ori-

---

<sup>4</sup>En física, cuando se hace un experimento y se toman unas medidas, estas llevan un error asociado debido a las imprecisiones de los instrumentos de medida. Con estas medidas podemos calcular distintas magnitudes que también llevan asociado un error. Estos errores se suelen denotar con  $\sigma$  y nos dan un intervalo de valores posibles para el valor real de la medida. Cuando se hace un experimento, normalmente se tiene un valor teórico que se espera obtener, sin embargo, puede darse el caso que la medida obtenida difiera de la esperada. En el caso que la medida observada difiere en  $5\sigma$  de la teórica, se establece que la teoría es errónea, ya que la probabilidad de que la diferencia se deba a motivos experimentales es extremadamente baja.

ginar también estas cascadas, por lo que es necesario ser capaces de distinguir las cascadas producidas por un electrón y las producidas por un fotón. Por otro lado, tenemos que tener en cuenta que la reacción descrita en la Ec. 2.1 no es la única posible. Cuando el neutrino tiene mucha energía, pueden existir partículas adicionales en el estado final que, al decaer, produzcan dos fotones y, por tanto, originen cascadas electromagnéticas.

Por lo tanto, pueden darse cascadas electromagnéticas por tres motivos distintos. La primera opción es que la cascada sea generada por un electrón producido por un neutrino electrónico incidente al darse la Ec. 2.1. Otra opción es que la reacción final tenga partículas adicionales que decaigan en pares de fotones (puede darse tanto para las corrientes neutras como para las cargadas) produciendo cascadas. La última opción es que la cascada se produzca por un fotón que se encuentre en el detector por cualquier otro motivo. Si queremos saber el número de neutrinos del electrón que llegan al detector, tenemos que saber distinguir si el origen de la cascada es un electrón o un fotón. Como comentario adicional, en el detector también vamos a poder identificar los neutrinos que interactúan mediante la corriente neutra, pero los vamos a descartar porque no podemos determinar el tipo de neutrino que origina el evento. Esto lo podemos hacer al conocer el porcentaje de neutrinos que va a interactuar de esta forma. En resumen, para contar los neutrinos que llegan al detector, tenemos que distinguir las cascadas electromagnéticas producidas por electrones de las producidas por fotones. Una correcta contabilización es fundamental para poder determinar si realmente se tienen medidas anómalas. A continuación, vamos a mostrar en detalle qué es una cascada electromagnética y cómo funciona el detector del que provienen las simulaciones que vamos a utilizar.

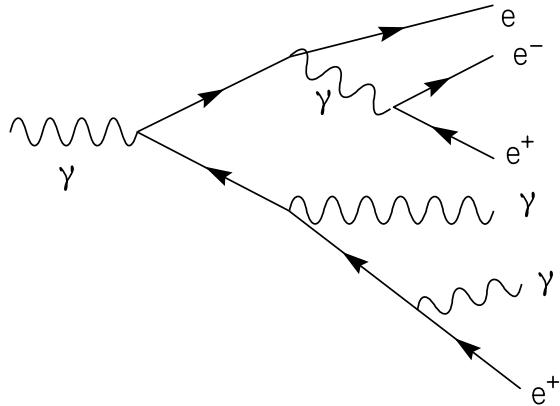
### 2.1.1. Cascada electromagnética

Las cascadas electromagnéticas son una sucesión de reacciones originadas por un electrón o por un fotón en el interior de un material. Estas partículas interactúan con el medio generando electrones, fotones y positrones (antipartícula del electrón), que a su vez interactúan con el medio generando nuevos electrones, fotones y positrones mediante una reacción en cadena.

De manera muy simplificada, estos eventos tienen lugar cuando tenemos electrones y fotones con altas energías (del orden de varios MeV). A estas energías, los electrones

interaccionan con la materia principalmente con el mecanismo que se conoce como radiación de Bremsstrahlung, que consiste en que un electrón (o un positrón) emite un fotón. Los fotones de alta energía interaccionan principalmente con la materia mediante la producción de pares, es decir, un fotón se desintegra creando un electrón y un positrón. Hay otros mecanismos de interacción posible, pero en estos rangos de energías son despreciables.

Supongamos entonces que, inicialmente, tenemos un electrón de alta energía en un medio. El electrón producirá un fotón de Bremsstrahlung. Mientras que la energía siga siendo alta, este electrón volverá a crear un electrón y un fotón, mientras que el fotón creará otro electrón y un positrón. En este punto tenemos ya cuatro partículas, y a su vez cada una de estas generará otras dos y así sucesivamente hasta que las partículas no tengan energía para generar otro par de partículas. En el caso de que la partícula inicial fuera un fotón, cambiaría simplemente el primer paso, donde inicialmente tenemos un fotón que genera un electrón y un positrón, iniciando así la cascada. En la Fig. 2.3 se muestra este evento mediante diagramas de Feynman donde las líneas onduladas representan fotones, las flechas hacia la derecha representan electrones y las flechas hacia la izquierda positrones.

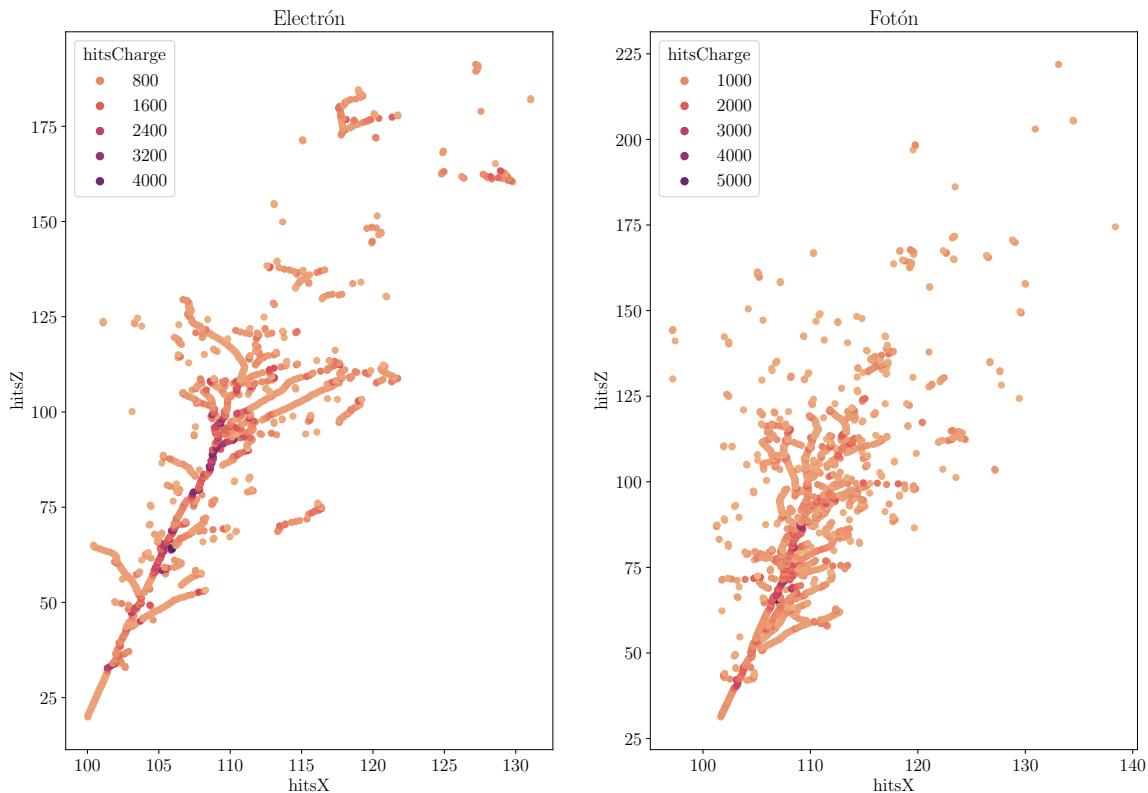


**Figura 2.3:** Esquema de una cascada electromagnética originada por un fotón. El sentido temporal es de izquierda a derecha [19].

Como se mencionó en la motivación física del problema, para detectar los neutrinos, necesitamos detectar las cascadas producidas por los electrones. Según como se han definido las cascadas, parece un problema difícil porque en los detectores solo podemos ver a los electrones y no es posible observar únicamente la primera interacción de la cascada. Por lo tanto, si tuviésemos justo lo que acabamos de explicar, resultaría prácticamente imposible distinguir las cascadas.

Sin embargo, sí existen diferencias entre ambas cascadas. La diferencia principal es que, para cascadas originadas por electrones, al comienzo de la cascada, se observa que el electrón recorre una distancia que podemos ver en el detector. Además, en las cascadas originadas por electrones, tenemos que los electrones de puntos más avanzados tienden a recorrer mayores distancias, no obstante, este efecto es mucho menor que el primero mencionado.

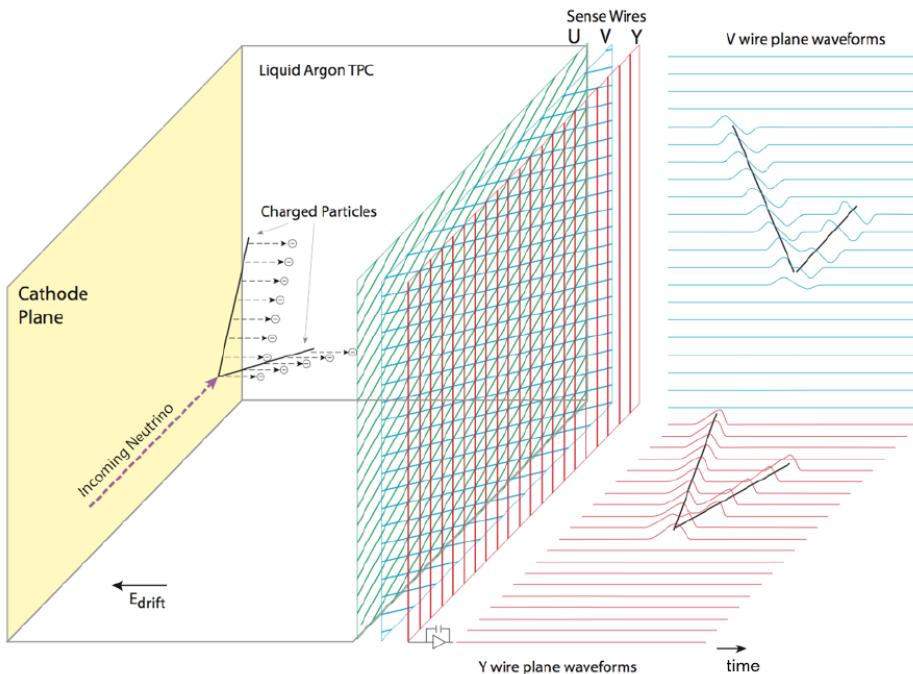
El hecho de saber la diferencia, no hace que sea un proceso simple, en la Fig. 2.4 mostramos dos imágenes obtenidas de los datos dados. Se muestra una cascada originada por un electrón y otra originada por un fotón.



**Figura 2.4:** Medidas simuladas de las trayectorias de las cascadas electromagnéticas. Se muestra una cascada originada por un electrón (izquierda) y otra originada por un fotón (derecha).

### 2.1.2. Detector LArTPC

Como se ha mencionado, los datos con los que vamos a trabajar proceden de simulaciones realizadas sobre un detector LArTPC, que se empleará en programas como SBN y DUNE. Veamos detalladamente cómo funciona y qué se mide en estos detectores [7]. En la Fig. 2.5 se muestra el esquema de funcionamiento de este detector.

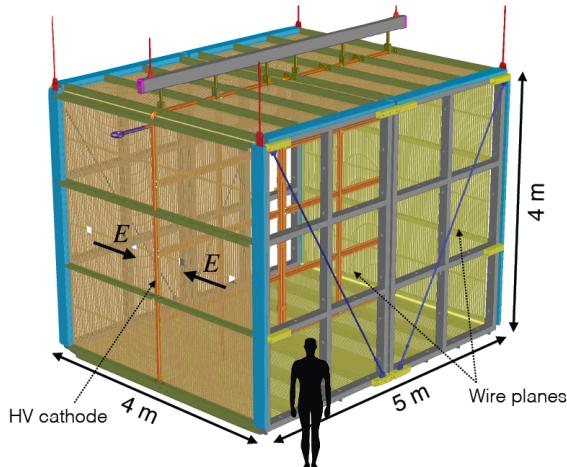


**Figura 2.5:** Esquema de funcionamiento de un TPC [7]. La partículas cargadas en el interior del detector arrancan electrones que son dirigidos hacia los planos de hilos donde se recogen su energía, posición en dicho plano y tiempo de llegada. Con esta información junto con el instante de tiempo inicial del evento se reconstruye la trayectoria de las partículas cargadas.

Estos detectores consisten en un volumen de argón líquido (LAr), donde tiene lugar las interacciones, y una cámara de proyección temporal (TPC) que nos permite trazar la trayectoria de partículas cargadas. El hecho de que se use argón líquido se debe a que es un gas noble y, por tanto, es un gas inerte permitiéndonos mantener el medio lo más puro posible. Además, el Ar posee un potencial de ionización bajo, de manera que es sencillo que una partícula cargada ionice el medio dejando electrones libres. A todo esto se le añade que el Ar es un gas noble muy abundante [20], que lo hace más accesible.

El detector consiste en un criostato<sup>5</sup> que contiene el argón y está dividido en dos partes por un plano que va a actuar como cátodo. Paralelo a este plano, en los extremos del criostato, vamos a tener planos de hilos que nos permiten registrar la señal dejada por los electrones liberados durante la ionización (ver Fig. 2.6). Cada plano consiste en tres planos de cables, que actúan como ánodo del circuito que van a recorrer los electrones libres. De esta forma, los detectores TPC son capaces de trazar las trayectorias de las partículas cargadas que se desplazan en su interior.

En la Fig. 2.6 mostramos un esquema de la estructura del detector. En lo que sigue, vamos a tomar como eje  $x$  la dirección del campo eléctrico, dirección en la que se van a desplazar los electrones libres y se denomina dirección de deriva. Como eje  $z$ , se considera el lado más largo del volumen del detector y esta será la dirección con la que entran los neutrinos. Finalmente, como eje  $y$  tomaremos la dirección restante. Los planos de hilos se encuentran en los extremos del volumen, en el plano YZ.



**Figura 2.6:** Esquema de la estructura del detector LArTPC del programa SBN [7]. En el eje  $x$ , dirección de deriva, se observa el campo eléctrico aplicado para dirigir los electrones libres hacia los planos de hilos. El eje  $z$  es el que contiene el lado más largo del detector y es la dirección con la que inciden los neutrinos. El eje  $y$  es el eje restante. Vemos que perpendicular a la dirección de deriva tenemos un plano que parte el detector en dos regiones y actúa como cátodo. En los extremos se tienen los planos hilos que actúan como ánodos.

Lo primero que vamos a mostrar es cómo, dada una partícula cargada, vamos a poder trazar su trayectoria. Cuando una partícula cargada con energía suficiente se mueve por

---

<sup>5</sup>Un criostato es un dispositivo que nos permite mantener bajas temperaturas.

el argón neutro puede darse que, por fuerza coulombiana, arranque electrones ligados a los átomos de argón, generando electrones libres; de forma que se obtiene una traza de su recorrido. En la Fig. 2.5 se puede ver representado gráficamente este hecho.

Al aplicar un campo eléctrico, los electrones libres son dirigidos hacia los planos de hilos donde son recogidos emitiendo un pulso eléctrico. Hay muchos detectores que usan este principio, como puede ser un contador Geiger. Sin embargo, con el LArTPC se pretende reconstruir la trayectoria seguida por las partículas y para ello se hace lo siguiente. Por un lado, se utilizan fotodetectores para determinar el instante tiempo  $t_0$  en el que comienza la trayectoria. En este instante es cuando el neutrino interacciona con el medio y, por tanto, el momento en el que comienza la trayectoria de las partículas que queremos estudiar.

Cuando se origina la partícula cargada, esta se desplaza arrancando electrones que, con un campo eléctrico perpendicular al plano YZ, dirigimos hasta los planos de hilos en los extremos. Cada plano de hilos consiste en tres planos de cables que se pueden observar en la Fig. 2.5. Estos planos son paralelos, pero los cables de cada plano tienen distintas orientaciones. La clave es que con dos de estos planos podemos determinar las coordenadas de los electrones libres en el plano YZ.

Para entender esto, simplifiquemos el problema. Supongamos que tenemos solo dos planos de hilos, uno vertical y otro horizontal, de manera que son perpendiculares entre sí, es decir, tenemos una rejilla. Cuando llegue uno de los electrones libres al plano se producirá una pulso eléctrico en uno de los cables del plano vertical y en uno del horizontal, por lo que sabiendo la posición de estos cables podemos determinar la posición del electrón en dicho plano. A estos pulsos les denominamos *hits* y, además, recogemos información de la energía del electrón libre, ya que en el cable se recogerá una carga, proporcional a la energía, que es lo que se denomina carga del *hit*. Adicionalmente, se mide el instante de llegada de los electrones al plano.

La situación real es que los planos no son horizontales y verticales, y el tercer plano está para deshacer ambigüedades que se pueden dar al realizar la reconstrucción. De forma que, los planos de hilos nos permiten obtener la posición en el plano YZ de los electrones que hayan sido arrancados por la partícula cargada. Lo que es lo mismo, obtenemos la trayectoria de dicha partícula proyectada en el plano YZ. Para obtener información de la dimensión restante lo que hacemos es usar  $t_0$  y el tiempo de llegada de los electrones libre

al plano, ya que al saber su velocidad de deriva en el medio, podemos obtener la distancia al plano YZ.

En resumen, inicialmente llega un neutrino, que interacciona con el Ar, obteniendo un protón y un electrón (recordemos que esta es una de las opciones posibles). Con los fotodetectores obtenemos el instante de tiempo en el que tiene lugar esta interacción. Al ser ambas partículas cargadas, vamos a tener un rastro de electrones libres por cada partícula que, debido al campo eléctrico en el eje  $x$ , se desplazan hacia el plano de hilos, obteniendo la proyección de cada trayectoria en el plano YZ. La información de la trayectoria en el eje  $x$ , la obtenemos usando el tiempo de origen, así como el tiempo en el que llegan los electrones al plano de hilos.

En definitiva, vamos a poder obtener la trayectoria completa de una partícula cargada, es decir, la región del espacio recorrida por la partícula. Los datos finales, tras procesar las señales, son las coordenadas de los *hits* dejados por la partícula cargada. Estos datos se asemejan mucho a una imagen, que no es más que una matriz cuyos elementos representan los píxeles y su valor la intensidad. En nuestro caso, tenemos las coordenadas y la carga que lo podemos entender como la posición en la matriz y la intensidad, respectivamente.

Nosotros vamos a centrarnos en el estudio de las cascadas electromagnéticas que se van a producir en el interior del detector. Una de las situaciones que podemos tener es que el neutrino llegue, produzca un electrón y un protón, este electrón va a desplazarse por el medio creando electrones libres (y, por tanto, lo observamos) hasta que se desintegra generando un electrón y un fotón iniciándose la cascada electromagnética. La otra opción posible es que un fotón origine la cascada. Nuestro objetivo fundamental en este trabajo es conseguir distinguir ambos tipos de sucesos, ya que nosotros solo tenemos que contar las cascadas electromagnéticas producidas por electrones.

## 2.2. Conceptos de computación

En esta sección recogemos los conceptos relacionados con el Deep Learning y con la computación necesarios para comprender completamente este trabajo. En concreto, vamos a desarrollar los fundamentos del Deep Learning centrándonos en las redes convolucionales y haremos una breve descripción de qué es el MLOps.

### 2.2.1. Deep Learning

El Deep Learning (DL) es un conjunto de técnicas y modelos que están dentro del ámbito del Machine Learning (ML), y que, en los últimos años, ha experimentado unos enormes avances en muchos problemas de ciencia de datos que se habían resistido a los métodos clásicos [21]. Entre estos problemas se encuentran la clasificación de imágenes, donde el uso de redes neuronales convolucionales (CNN) se ha convertido en el estado del arte y ha conseguido obtener mejores resultados que los propios humanos en algunos problemas [22]. Otros ejemplos son el procesamiento de lenguaje natural con los transformers o el reconocimiento del habla [23].

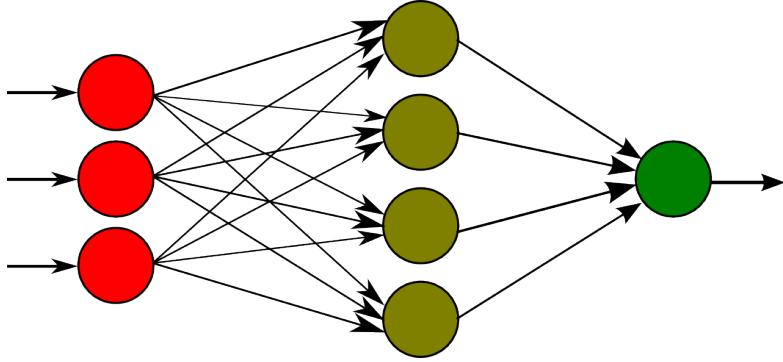
El DL tiene la ventaja de que es capaz de trabajar con los datos en bruto, de hecho, se define como el conjunto de modelos que son capaces de obtener las características directamente de los datos [24]. Estos rápidos avances se deben, en gran medida, a que se dispone de una gran cantidad de datos (gracias al desarrollo del Big Data) que pueden ser procesados automáticamente, así como de distintos avances tecnológicos [23, 21]. De hecho, los avances son tales que, con la combinación de modelos de DL, hemos pasado de ser capaces de clasificar imágenes a lograr generar imágenes nuevas dado un texto [25]. Otro gran avance son los modelos capaces de escribir código dadas las instrucciones de lo que se quiere realizar [26], todo esto en un periodo de solo 10 años.

Con todo esto, vamos a ver los fundamentos necesarios para entender qué es una CNN, sus elementos, su complejidad y mostraremos algunos ejemplos de esta arquitectura de red neuronal. No profundizaremos demasiado, pero en esta sección se pretende mostrar qué es el DL y una CNN, y cuáles son sus principios básicos de funcionamiento.

## Fundamentos

Las redes neuronales (NN) surgen con los perceptrones multicapa (MLP) que consisten en la composición de funciones paramétricas para formar una función más compleja. Este modelo está inspirado en el funcionamiento del cerebro, por analogía con las neuronas y sus activaciones. Esta representación se puede observar en la Fig. 2.7, pero no es más que una manera de expresar una serie de operaciones matemáticas, donde los nodos representan las

entradas y salidas de funciones, y los enlaces representan el valor de los coeficientes de la aplicación, que se denominan pesos.



**Figura 2.7:** Esquema de un modelo MLP con una capa de entrada, una capa oculta y una capa de salida [27].

De hecho, el paso de una capa a otra en un MLP no es más que una aplicación lineal:

$$\mathbf{y} = W\mathbf{x}, \quad (2.2)$$

a la que se le aplica una función no lineal. Donde  $\mathbf{x} \in \mathbb{R}^n$  es la entrada y son los nodos iniciales,  $\mathbf{y} \in \mathbb{R}^m$  son los nodos de salida y la matriz  $W$  se representa con los enlaces entre nodos. Destacar que las dimensiones de entrada y de salida no tienen por qué ser las mismas. Por otro lado, la función no lineal se aplica para que el modelo sea capaz de aprender relaciones más complejas. Hay muchas opciones, normalmente se usa la sigmoide o la RELU entre otras muchas y se suelen denominar funciones de activación [24].

En la ecuación 2.2 lo que hemos mostrado es la función matemática de un MLP, donde solo tenemos una capa de entrada y otra de salida, y donde no se aplica ninguna función no lineal, este ejemplo equivale a un modelo de regresión. Se puede añadir más complejidad introduciendo más capas y estas funciones no lineales. Un ejemplo de la formulación de un MLP con una capa oculta es:

$$\mathbf{z} = \sigma(W\mathbf{x}), \quad \mathbf{y} = \sigma(V\mathbf{z}), \quad (2.3)$$

donde  $\sigma$  representa la función sigmoide,  $\mathbf{x}$  es el vector de entrada,  $\mathbf{z}$  es el de salida de la capa inicial e  $\mathbf{y}$  el vector de salida de la NN. Las matrices  $W$  y  $V$  definen las aplicaciones lineales y representan los enlaces entre capas.

Un detalle importante sobre estos modelos es que se puede demostrar que son aproximadores universales, es decir, que toda función puede aproximarse usando una NN [28, 29], el problema surge con la cantidad de capas y neuronas necesarias, así como en el proceso de aprendizaje de los pesos correctos para representar dicha función.

El punto más importante de los modelos de DL es que son capaces de aprender las características necesarias para resolver el problema. Hasta entonces, si se quería estudiar una imagen, lo que se hacía era aplicar distintos algoritmos, como LBP, para obtener unas características concretas. Los algoritmos se diseñaban para resolver un problema concreto. El problema es que el diseño de estos algoritmos requiere de un esfuerzo muy grande y los resultados no son los mejores [21]. Como ya hemos dicho, esto es posiblemente uno de los factores que han hecho posible que se tengan grandes avances, ya que no tenemos que perder tanto tiempo en la extracción de características [21]. Estos modelos no solo agilizan el proceso, sino que además son capaces de detectar patrones que incluso los expertos no son capaces de visualizar.

Los MLP establecen las bases de todos los modelos actuales de Deep Learning. Con el tiempo, han ido surgiendo distintos tipos de NN, como las CNN, las RNN, los autoencoder o los transformers. En las MLP el paso de una capa a otra no es más que aplicar la función de activación tras realizar una aplicación lineal sobre un vector de entrada. Las distintas arquitecturas lo que hacen es cambiar esta representación y, por tanto, cambiar la función matemática aplicada sobre los datos de entrada. Cómo se definen las redes, influirá en los tipos de datos con los que la red podrá trabajar.

Otro aspecto importante es que las NN, por definición, no son más que la composición de distintas funciones donde cada función la podemos entender como una capa o incluso un conjunto de capas (la composición de funciones es una función). Esto simplifica la construcción de redes en el sentido conceptual porque, simplemente, basta con pensar en la combinación de distintos bloques y cómo interaccionan entre sí estos bloques. Por ejemplo, a las capas que forman un MLP se les denominan densas (todos los nodos están unidos) y un MLP no es más que el uso de varias de estas capas. En las CNN se usan lo que se denominan capas de convolución, que son un conjunto de capas que realizan la operación de convolución. Hay otros muchos bloques y el diseño de una NN consiste en la aplicación de sucesivos bloques que dependen del objetivo final que se tenga.

La dificultad reside en diseñar bloques que sirvan para el tipo de datos que se estén usando y en cómo combinar los bloques para obtener los resultados deseados, pero no en la construcción del modelo, una vez sabemos cuál es su forma.

## CNN

Las CNN son un tipo de NN donde la función matemática que se realiza es la de convolución. Antes de entrar en la parte matemática, mostremos la idea detrás de la capa de convolución. Para ello, supongamos que tenemos una imagen, que se representa como una matriz donde cada elemento describe a un píxel.

Sobre esta imagen podemos aplicar un filtro pequeño, por ejemplo, un bloque  $3 \times 3$  e ir recorriendo la imagen. Este filtro tendrá una forma dada por el valor de los píxeles del filtro. Por ejemplo, podemos definir un filtro que sea una línea horizontal, de manera que se tiene 1 en los píxeles que forman la línea y 0 para el resto de píxeles. Este filtro lo centramos en los distintos píxeles de la imagen y hacemos una operación que involucre al filtro y a los píxeles de la imagen que coincidan espacialmente con el filtro. Esta operación puede consistir únicamente en multiplicar elemento a elemento, los píxeles del filtro y de la imagen, y sumarlos. Es decir, sea una matriz  $I$ , la imagen, y una matriz  $K$ , el filtro, donde las dimensiones de  $I$  son mayores que las de  $K$ . Lo que se hace es centrar la matriz  $K$  en los distintos píxeles de  $I$  y hacer un producto elemento a elemento, entre la matriz  $K$  y la submatriz de  $I$ , y hacer la suma de los valores obtenidos. Si recogemos todos los resultados en una matriz, el resultado de aplicar el filtro sería otra matriz. Esto es lo que se conoce como operación de convolución, pero podemos cambiar la operación para realizar otros efectos sobre la imagen.

Veamos esto de manera formal. Sea  $I$  una matriz de dimensiones  $p \times q$  y  $K$  una matriz de dimensiones  $r \times s$  tal que  $p > r$  y  $q > s$ . Por simplicidad, suponemos que  $r$  y  $s$  son impares. Definimos aplicar un filtro como:

$$S_{i,j} = \sum_{n=1}^r \sum_{m=1}^s I_{i-\lfloor r/2 \rfloor + n, j - \lfloor s/2 \rfloor + m} K_{n,m}, \quad (2.4)$$

con  $i \in \{1, \dots, p-1\}$  y  $j \in \{1, \dots, q-1\}$ . La matriz  $S$  tiene dimensiones  $p-1 \times q-1$ . Hay distintas definiciones según cómo tratemos los bordes de la imagen. Por ejemplo,

podríamos considerar que cuando nos saliéramos de la imagen, se consideren los píxeles del otro extremo, de manera que la matriz resultante tendría las mismas dimensiones que la imagen original. También se puede aumentar la imagen introduciendo píxeles vacíos hasta poder aplicar el filtro en los bordes de la imagen inicial. Otro detalle que podemos destacar es que en la definición dada solo se contempla que el filtro se aplique centrándose en todos los píxeles posibles. Sin embargo, también se puede hacer de manera que el desplazamiento realizado al desplazar el filtro no sea un únicamente píxel.

El uso de estos filtros nos permite determinar qué regiones de la imagen tienen unos patrones que nos pueden interesar. Con las CNN, la clave es que la red aprende los mejores filtros para realizar una tarea concreta.

La operación que acabamos de describir se denomina convolución. Matemáticamente la convolución entre dos funciones  $(x * w)(t)$  se puede escribir, en el caso de una dimensión, como:

$$(x * w)(t) := \int_{-\infty}^{\infty} x(a)w(t - a)da, \quad (2.5)$$

donde  $x$  y  $w$  son funciones que dependen de una variable. En el ámbito del DL a  $x$  se le denomina entrada y a  $w$  el *kernel* [24]. La expresión en el caso discreto quedaría como:

$$(x * w)(t) = \sum_{a=-\infty}^{\infty} x[a]w[t - a]. \quad (2.6)$$

Esta definición se puede generalizar para varias dimensiones, añadiendo tantas integrales como dimensiones. En el caso de las imágenes, tenemos que el espacio es discreto y que tenemos dos dimensiones, entonces, la operación de convolución quedaría:

$$s[i, j] = (I * K)[i, j] = \sum_m \sum_n I[m, n]K[i - m, j - n], \quad (2.7)$$

donde  $I$  es una matriz de dimensiones  $p \times q$ , la entrada, y  $K$  es una matriz de dimensiones  $k \times l$ , el *kernel*. Los límites de los sumatorios realmente serían de menos infinito a infinito, pero en nuestro caso, está determinado por las dimensiones de  $K$  (podemos definir  $K = 0$  si los índices superan las dimensiones). Notar la gran similitud con la ecuación 2.4.

La capa de convolución consiste en aplicar el filtro y luego (en la mayoría de casos) aplicar una capa de reducción o *pooling*, que consiste en reducir las dimensiones de la

salida mediante la agregación de un grupo de píxeles. Por ejemplo, una imagen la podemos reducir si fragmentamos la imagen en bloques  $2 \times 2$  px y hacemos la media a estos bloques. Esta operación ayuda a que el resultado de la aplicación del filtro sea invariante a pequeños desplazamientos, que es fundamental a la hora de estudiar imágenes porque queremos que imágenes que se distingan solo por desplazamientos, se clasifiquen igual (pensar que la imagen puede estar tomada desde puntos de vista distintos, pero contener el mismo objeto) [24].

Estos son los elementos básicos de una CNN y es el punto de partida a la hora de construirlas. En resumen, hay muchas arquitecturas posibles para una NN donde cada una de estas arquitecturas tendrán unas características que hacen que sean útiles para un tipo de problema concreto. Las CNN son muy buenas a la hora de estudiar imágenes, ya que permiten estudiar regiones de la imagen y determinar qué elementos y patrones se tienen en la imagen, desde patrones muy simples, que se obtiene con las primeras capas de convolución, a patrones más complejos cuando aumentamos la complejidad del modelo.

## Aprendizaje

Hasta ahora hemos hablado de cómo es una NN. Sin embargo, no hemos entrado en cómo se determinan los pesos para que sean capaces de resolver un problema determinado. Sin entrar en mucho detalle, lo que se hace es emplear lo que se conoce como el descenso del gradiente [30] y la propagación hacia atrás, conocida por su término en inglés *backpropagation* [31].

El objetivo de un problema de ciencia de datos es, dados unos datos, usar un modelo que sea capaz de realizar una determinada tarea sobre estos con la intención de identificar patrones no triviales. En el caso del aprendizaje supervisado, se parte de que se tienen unas entradas de las cuales conocemos su clase/valor. Con estos valores se entrena el modelo para que realice la tarea concreta. Las NN hacen uso de los datos y sus etiquetas para determinar los pesos que mejor realizan la tarea concreta que se esté realizado. Las tareas clásicas en aprendizaje supervisado son la clasificación y la regresión [24]. En ambos problemas, tenemos un conjunto de instancias de las que tenemos distintas variables de entrada y una (o varias) variables de salida. La diferencia entre clasificación y regresión está en la variable de salida. La variable de salida en regresión toma valores en un rango

continuo y su objetivo es predecir el valor de esta variable, dadas las variables de entrada. En clasificación, la variable de salida es discreta y al conjunto de valores posible se le suele denominar clase. El objetivo de la clasificación es determinar la clase de una instancia, dadas las variables de entrada.

El descenso del gradiente consiste en definir una función pérdida que tome la clase dada por la red y la compare con la clase real. En el caso de que la red se equivoque, la salida de la función pérdida será mayor y será menor a medida que el modelo acierte la clase de las distintas instancias. Esta función va a depender de los pesos de la red, ya que son estos pesos los que determinan la clase que se asigna a la instancia. El descenso del gradiente consiste en obtener el gradiente de esta función pérdida con respecto a los parámetros de la red, obteniendo la dirección de crecimiento de la función. A continuación, se modifican los pesos en la dirección contraria para reducir el valor de la función pérdida y así mejorar los resultados.

Matemáticamente hablando, el problema consiste en la optimización de una función de varias variables, los pesos de red. Los pesos se inicializan de manera aleatoria, se calcula el gradiente y se cambian el valor de estos pesos en función del valor obtenido. El cambio se hace poco a poco y se regula con lo que se conoce como ritmo de aprendizaje ( $lr$ ). Hay distintos algoritmos para este problema, como puede ser SGD o Adam [32, 33].

El proceso de aprendizaje suele realizarse dando un pequeño conjunto de datos, obteniendo el valor de la función pérdida para cada dato, promediando y, finalmente, ajustar los pesos calculando el gradiente. Esto se repite para todos las instancias y, normalmente, este procedimiento se hace varias veces con el mismo conjunto de datos, de manera que durante el aprendizaje el modelo estudia cada dato varias veces. Al conjunto de datos pequeños, se le conoce como *batch size* ( $bs$ ) y al número de veces que se recorre el conjunto de datos como número de épocas ( $n_e$ ).

## Elementos

A modo de resumen final, vamos a recoger los distintos bloques que pueden formar una CNN. Un elemento necesario es el optimizador, algoritmo que realiza el descenso del gradiente, y el  $lr$ . En este punto, se construye la NN usando distintas capas. Las capas más simples son las que hemos ido mencionando. Una CNN muy simple la podemos construir

como la aplicación de varias capas de convolución y tras esto, utilizar un MLP que clasifique los resultados.

Hemos mostrado los bloques más simples necesarios para construir una CNN, pero con el tiempo han ido surgiendo nuevos bloques y modificaciones sobre estos, con la idea de mejorar la optimización o los resultados. También hay distintas formas de unir todos estos elementos para construir la red. No vamos a entrar en estos bloques, pues ha habido una gran cantidad de avances en los últimos años y una revisión de estos elementos es un trabajo en sí. Los elementos y arquitecturas más complejos se irán describiendo a medida que nos vayan surgiendo la necesidad de estudiarlos.

### 2.2.2. MLOps

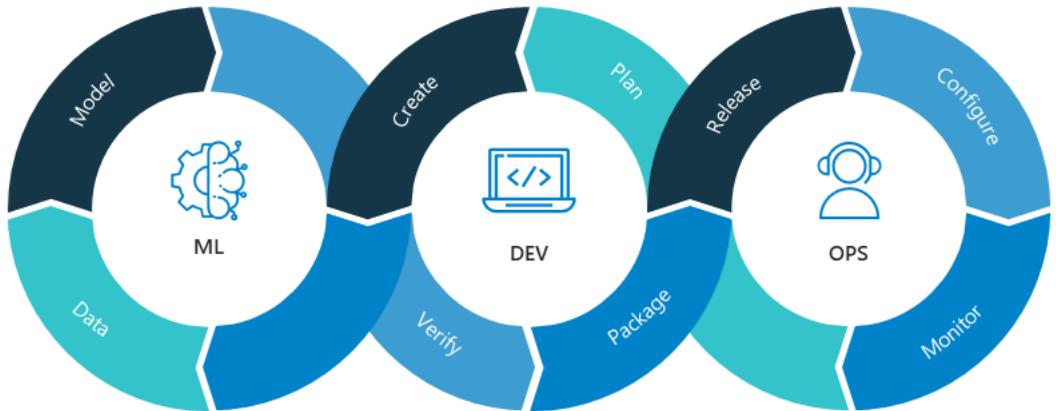
La resolución de un problema de ciencia de datos, es un proceso que puede llegar a ser muy complejo y depender de varias partes e incluso de varios equipos. De forma resumida, la resolución consiste en procesar los datos, analizarlos y entrenar un modelo. Una vez obtenido el modelo, este tiene que ponerse en funcionamiento y realizar la tarea para la que ha sido entrenado. En el momento que el proyecto crece, cada una de estas partes crece, llegando a un punto en el que cualquier modificación que se quiera realizar puede ralentizarse mucho debido a la complejidad del sistema.

El DevOps es un conjunto de prácticas para hacer el ciclo de desarrollo, implementación y ejecución de aplicaciones de forma eficiente [34]. MLOps, no es más que la aplicación de estas prácticas en una aplicación de Machine Learning. La filosofía de estos métodos de desarrollo es que los ciclos de versiones sean los más rápidos posibles, con la idea de aprender del funcionamiento de la aplicación y de la opinión de los usuarios, intentando automatizar lo máximo posible todas las etapas, minimizando los errores, mejorando la escalabilidad y manteniendo la calidad del código.

Veamos algunas de estas prácticas. El primer paso suele ser la integración y entrega continua, que consiste en probar de forma automática el software introducido, así como diseñar en torno a estas pruebas. Por otro lado, también se busca que el despliegue se realice de manera automática [35]. El empleo de sistemas de control de versiones como git

es fundamental, al permitir tener un seguimiento de la evolución del proyecto, así como la colaboración entre personas.

En MLOps, al DevOps, se le tiene que integrar la parte de ciencia de datos donde tenemos, por un lado, el procesamiento de estos datos y, por otro lado, el modelo final que es lo que nos interesa monitorizar y mejorar. Con estas técnicas se pretende automatizar el tratamiento de los datos, así como el despliegue y control sobre los modelos. En la Fig. 2.8 se muestra un esquema de un flujo de trabajo basado en MLOps, donde tenemos primero la parte de ciencia de datos en la que se tratan los datos y el modelo. Luego se tiene la parte de desarrollo donde, de manera automática, se verifica que todo funcione bien. A continuación, o bien se hace un despliegue, o bien se entrega el modelo al cliente, todo automáticamente. Tras el despliegue, entra en juego la parte de operaciones donde se hace uso de la aplicación y se aprende de su funcionamiento. Con esta información se modifican las etapas anteriores y se controla el correcto funcionamiento de los modelos.



**Figura 2.8:** Esquema del flujo de trabajo basado en MLOps [36]. En el centro tenemos el flujo de desarrollo donde tenemos primero la planificación y la creación de la aplicación. Luego, entramos en el flujo de ML donde se procesan los datos y se crea el modelo. Con esto volvemos al flujo de desarrollo donde se verifica y despliega la aplicación. Finalmente, entra en juego el flujo de operación donde se monitoriza el modelo y la aplicación para que, dependiendo de su funcionamiento, se vuelva a comenzar el ciclo.

Nosotros en este trabajo nos vamos a centrar sobre todo en implementar muy bien una estructura automática para los datos y modelos, ya que nos va a permitir trabajar

de manera mucho más ordenada y mantener un control del trabajo realizado y de los modelos obtenidos. En [37] se muestran tres niveles de implementación de MLOps: un primero, donde todo se hace de manera manual, un segundo, donde hay una estructura de flujo donde con el uso de cauces, se automatizan la mayoría de elementos y un tercero, donde se introduce la integración continua con el uso de test automatizados y el código se empaqueta y despliega automáticamente. Nosotros nos vamos a quedar en el segundo nivel donde tenemos un proceso completamente automático en el flujo de datos y del modelo, pero no vamos a entrar en el uso de test automáticos y despliegue automático, al ser un proceso más complejo, sobre todo para alguien sin experiencia previa.

### 2.3. Estado del arte

En física, los neutrinos estériles surgen como una explicación teórica, fuera del Modelo Estándar, de una serie de resultados anómalos en distintos detectores de neutrinos. En los detectores LSND [38] y MiniBooNE [39], se ha observado un exceso del número de neutrinos esperados, compatibles con la existencia del cuarto neutrino [7]. Además, se han observado otras anomalías conocidas como la anomalía del reactor o la anomalía de Galio, que también podrían explicarse con la presencia del neutrino estéril. Los programas SBN y DUNE tienen como objetivo realizar experimentos de seguimiento que traten de confirmar que efectivamente se tienen esas anomalías [7, 13]. El detector microBooNE [40] tenía un objetivo similar y ha publicado resultados recientemente, inconsistentes con los resultados de MiniBooNE [41]. Sin embargo, no dejan de ser unos resultados iniciales y siguen teniendo incertidumbres tales que no descartan completamente los resultados previos. Una de las incertidumbres principales proviene de la selección de los eventos que provienen de neutrinos y, por tanto, para reducir la incertidumbre, una de las cuestiones que hay que mejorar es la detección de cascadas electromagnéticas.

En los últimos años, distintos laboratorios, como el CERN, han aplicado técnicas de DL [6] con la idea de mejorar la capacidad de análisis de los datos que se obtienen. Modelos de DL se han usado para el seguimiento de partículas, la identificación de eventos, la identificación de *Jets* o en simulaciones aplicando redes generativas [42, 43, 44, 45]. De hecho, en microBooNE, un detector similar al nuestro, hace uso de las CNN para la clasificación de las cascadas [6].

Con respecto a la clasificación de imágenes, los mejores resultados en los últimos años vienen dados por CNN, con arquitecturas cada vez más complejas y más profundas. El rendimiento de estos modelos se suele evaluar en una competición anual con el conjunto de datos de ImageNet, que tiene muchas imágenes con muchas clases. Los ganadores de los últimos años han sido redes basadas en EfficientNet [22], que lo que hacen es introducir formas de escalar las redes (aumentar el número de capas y número de filtros) partiendo de un modelo base. Sin embargo, en los dos últimos años, los ganadores no han sido modelos basados en CNN, sino modelos que hacen uso de los transformers [3] que son una nueva arquitectura de NN, que se han convertido en el estado del arte en procesamiento de lenguaje natural. Los modelos que aplican los transformers a la clasificación de imágenes se conocen como Visual Transformers (ViT) [46], donde lo que hacen es proponer un modo de tratar las imágenes para que puedan ser empleadas por los transformers. ViT propone segmentar la imagen en bloques ( $16 \times 16$ ) y, manteniendo un orden, se pasan estos bloques a la red como si fueran una frase.

Otro aspecto fundamental en este trabajo es el uso del paradigma MLOps. Se puede definir como un conjunto de prácticas, conceptos y culturas de desarrollo con la intención de conseguir un rápido desarrollo de productos de Machine Learning y ponerlos en producción, tratando de solucionar el gran reto que supone la automatización y la puesta en funcionamiento de estos productos [47]. Sin embargo, este término es muy reciente y está en pleno proceso de desarrollo, siendo aún muy ambiguo [47]. No obstante, esta filosofía se fundamenta en el DevOps [48] aplicado a los modelos de ML [47]. Actualmente, se tienen distintas herramientas que nos facilitan la aplicación práctica de estas técnicas: MLFlow para el registro de modelos y despliegue automático de modelos [49], Optuna para el ajuste de hiperparámetros [50], Snapper para facilitar la implementación en código de estas técnicas [51, 52] o Amazon SageMaker como una plataforma en la nube con una gran batería de soluciones para facilitar el flujo de trabajo de un producto de ML [53].

## Capítulo 3

# Machine Learning aplicado a la clasificación de cascadas electromagnéticas

En este capítulo vamos a mostrar los datos y cómo los vamos a procesar para poder abordar el problema que se nos presenta, es lo que se conoce normalmente como ingeniería de los datos. Para ello, primero describiremos cuál es el problema y cuáles son los datos con los que partimos.

Posteriormente, se mostrarán las distintas propuestas que hemos diseñado específicamente para la resolución de este problema, tanto en los modelos empleados como en el procesamiento de los datos. En este capítulo nos centraremos únicamente en enumerar las alternativas propuestas, en el Capítulo 5 haremos un desarrollo detallado de su implementación y en el Capítulo 6 estudiaremos cuál de estas representaciones y modelos son los que mejores resultados consiguen.

En ningún momento vamos a entrar en el código desarrollado más allá de un esquema del flujo realizado<sup>1</sup>. No obstante, gran parte de este trabajo ha consistido en la implementación del flujo completo, pues se ha partido desde cero. Además, se ha dedicado una especial atención a hacer una implementación tratando de aplicar buenas prácticas de programación

---

<sup>1</sup>El código desarrollado está disponible en el repositorio: [https://github.com/aponce1509/tfm\\_](https://github.com/aponce1509/tfm_).

[54], así como mantener una estructura de MLOps, teniendo en cuenta que, como físico, no se tenía una formación previa de cómo abordar de manera correcta un proyecto de desarrollo software. Todas estas prácticas son fundamentales a la hora de hacer el desarrollo de un proyecto de ML, tanto a modo de formación personal, como para que futuras personas puedan usar el código y continuar su desarrollo.

### 3.1. Descripción del problema

Como ya se ha mencionado, nuestro problema consiste en la clasificación de un tipo de evento que puede darse en un detector LArTPC, este evento se conoce como cascada electromagnética y puede originarse por un electrón o por un fotón. Nuestro objetivo es determinar si una cascada es de origen fotónico o electrónico. En la Sección 2.1 se ha descrito de manera detallada cómo se producen dichas cascadas y cómo el detector transforma pulsos eléctricos en los datos con los que nosotros vamos a trabajar.

En nuestro caso, los datos están simplificados para contener únicamente cascadas electromagnéticas, por lo tanto, estos eventos son las trayectorias dejadas por estas cascadas en el interior del detector. Estos eventos los tenemos clasificados en función de qué tipo de partícula origina el evento.

Los datos que se nos han proporcionado consisten en datos tabulares etiquetados por el tipo de cascada. Tenemos 7 columnas, tres de ellas hacen referencia a un identificador del evento, a su energía en GeV<sup>2</sup> y al tipo de partícula, respectivamente<sup>3</sup>. También se tienen otras tres columnas que describen la posición tridimensional del *hit* dando las coordenadas  $x$ ,  $y$ ,  $z$ . La última columna recoge la cantidad de carga recogida del *hit*, que es una magnitud proporcional a la energía depositada por la partícula cargada en ese punto de la trayectoria<sup>4</sup>.

---

<sup>2</sup>Un eV es una unidad de medida de energía que se define como la energía que tiene un electrón al aplicarle un campo eléctrico con una diferencia de potencial de 1V.

<sup>3</sup>El Particle Data Group, con la idea de tener un identificador único y estandarizado para todas las partículas, asignó un número natural a cada una de las partículas para su identificación [55]. Las partículas que pueden originar los eventos que estamos estudiando son un electrón y un fotón, cuyos identificadores son el 11 y 22, respectivamente.

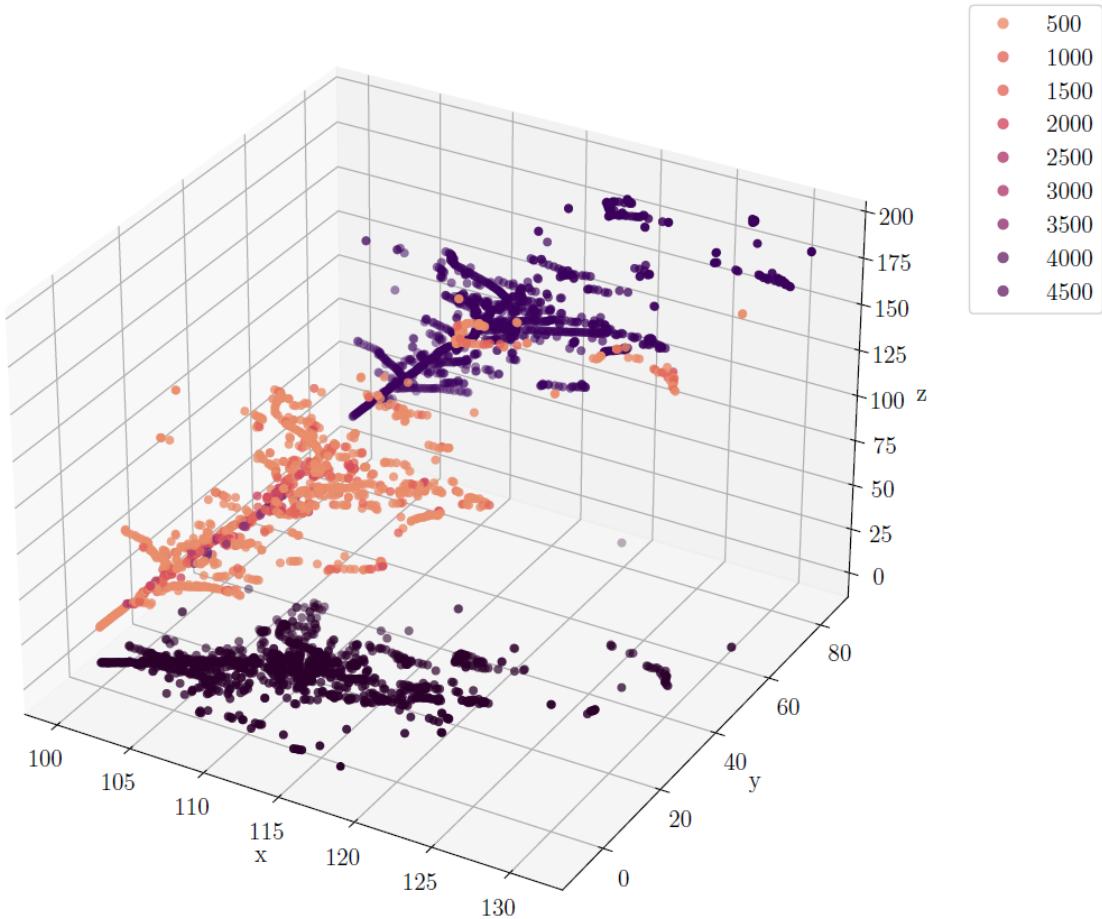
<sup>4</sup>Las unidades de la carga están dadas en tick · ADC, unidades que surgen del proceso de digitalización de la señal continua originada por los electrones libres en los planos de hilos del detector. La señal se digitaliza con una frecuencia de muestreo de 5 µs que son los ticks y la cuantización de la señal se hace de manera que un electrón equivale a  $6.3 \times 10^{-3}$  ADC.

Por otro lado, es importante indicar que en los datos no tenemos una única fila por evento, para cada evento tenemos varias filas que describen la trayectoria completa de la cascada.

Los datos con los que trabajamos provienen de simulaciones de Montecarlo que simulan con mucha exactitud el comportamiento real del detector. El uso de simulaciones supone una serie de ventajas. Por un lado, tenemos una gran cantidad de datos y se podrían obtener más en caso de ser necesario. Además, al tener control total de la simulación, tenemos información privilegiada, en el sentido de que es información que no se tiene en el detector real y se tiene que inferir. En concreto, disponemos del valor de la energía del evento, que realmente se tiene que obtener de distintas medidas en el detector. Por esto mismo, a la hora de hacer la clasificación no vamos a hacer uso de esta variable, pero sí la usaremos para validar el modelo y saber si nuestro modelo funciona correctamente para todos los rangos de energía.

Como hemos dicho, las trayectorias recogidas en los datos han sido filtradas para que solo contengan cascadas electromagnéticas. Realmente, un evento en el detector LArTPC está formado por la trayectoria de las distintas partículas que intervienen en una interacción. Por ejemplo, en la corriente cargada tendremos la traza dejada por el protón y el leptón. Por lo tanto, en el problema real sería necesario aplicar detección de objetos y quedarnos únicamente con la región que contenga a la cascadas, sin embargo, para esto necesitamos un buen clasificador, que es lo que se ha abordado en este trabajo. En la Fig. 3.1 se muestra una representación tridimensional de los datos tal cual se nos han proporcionado. En la Fig. 3.2 se muestran las tres vistas de esta misma trayectoria.

Nuestro objetivo principal en este trabajo consiste en distinguir las cascadas originadas por los fotones de las producidas por los electrones y tenemos que ver cómo abordar este problema desde el punto de vista de la ciencia de datos. Este detector nos permite reconstruir los eventos como imágenes tridimensionales, por lo que podemos abordar este problema como un problema de clasificación de imágenes. Tratar los datos como imágenes es una selección que hacemos, ya que queremos ver cómo actúan los clasificadores de imágenes del estado del arte en este problema concreto. Esta decisión la hemos tomado, pues la forma natural representar los datos proporcionados por este detector es mediante



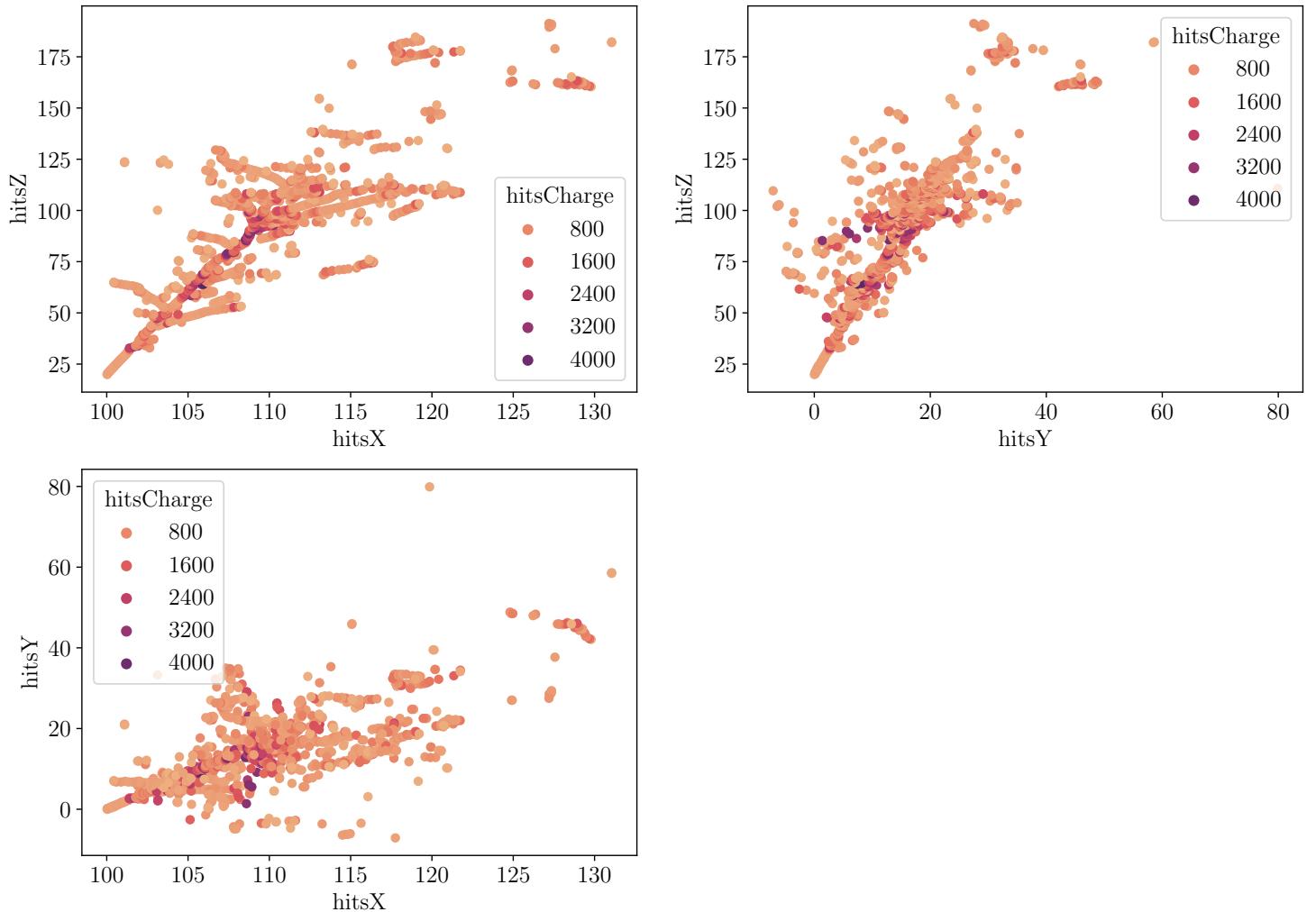
**Figura 3.1:** Representación tridimensional de una cascada electromagnética originada por un electrón con los datos en bruto. El color de los puntos viene dado en función de la carga del *hit*. Se muestran también las proyecciones sobre el plano XZ y sobre el plano XY con colores fijos.

imágenes<sup>5</sup> y, además, nos permite entender y visualizar los eventos. Por otro lado, en [56] se ha llevado a cabo un enfoque con una representación numérica y nosotros queremos ver si una representación con imágenes mejora los resultados.

Las redes convolucionales son el estado del arte en clasificación de imágenes, por lo que todos los enfoques que vamos a desarrollar van a hacer uso de estas arquitecturas. Sin embargo, se pueden abordar mediante otros procedimientos, como puede ser hacer la clasificación mediante la definición de nuevas variables a partir de las imágenes o el uso

---

<sup>5</sup>En la Sección 2.1.2, hemos descrito el funcionamiento del detector simulado y vemos cómo recoge la trayectoria de los eventos con la idea de visualizar las partículas como si estuvieran en una cámara de niebla, queremos ver las partículas y con este detector se pretende, entre otras cosas, obtener reconstrucciones tridimensionales de las trayectorias dejadas por las partículas.



**Figura 3.2:** Representación de las tres vistas de una cascada electromagnética originada por un electrón con los datos en bruto. El color de los puntos viene dado en función de la carga del *hit*.

de métodos clásicos de extracción de características. Estos enfoques son más clásicos en el sentido que no se hace uso del Deep Learning.

## 3.2. Preprocesamiento de los datos

Antes de entrar en los modelos que vamos a usar para hacer la clasificación, tenemos que ver cómo se tratan los datos iniciales para que podamos utilizar un modelo de aprendizaje. Lo primero que tenemos que hacer es pasar los datos a imágenes. En nuestro caso, al trabajar con imágenes tridimensionales, tendremos que trabajar con matrices con cuatro dimensiones, tres de posición y una para el canal de color. El color de la imagen lo asociaremos con la cantidad de carga depositada por el *hit*, por lo que, en principio, trabajamos con imágenes en blanco y negro.

### 3.2.1. Formas de representación

Una vez que hemos construido las imágenes, vamos a plantear distintas formas de tratarlas que consideramos que pueden ayudar a la hora de discriminar los eventos. Estas son los que enumeramos a continuación:

- Trabajar directamente con las imágenes tridimensionales. Las CNN no están limitadas a trabajar con imágenes bidimensionales, pues la operación de convolución se puede aplicar sobre matrices de cualquier dimensión.
- Trabajar con imágenes bidimensionales donde nos quedamos únicamente con una proyección. El problema con este enfoque es que se pierde información.
- Codificar la información perdida a la hora de hacer la proyección en el color de la imagen, de manera que mantenemos toda la información usando imágenes bidimensionales pero con tres canales de color.
- Entrenar distintos modelos para cada proyección y agregar los resultados. Podemos usar un ensemble o agregar las características obtenidas en los tres modelos.

Independientemente del modelo o enfoque empleado, es necesario pasar los datos tabulares a imágenes, pero surgen una serie de problemas al realizar esta conversión, en concreto, surge con la resolución de la imagen, pues si queremos mantener todos los *hits* vamos a tener que imágenes muy grandes. Esto se debe a que la posición viene dada en

centímetros y tenemos una precisión de varias décimas de milímetro, por lo que necesitaremos de muchos píxeles, ya que las dimensiones del detector SBND (el detector particular que hemos simulado en el estudio) son de  $400 \times 400 \times 500$  cm<sup>3</sup>. Notar que la precisión del detector está relacionada con la resolución de la imagen, entendiendo resolución de la imagen como el número de píxeles que usamos para representar el volumen completo del detector. El problema surge si queremos mantener toda esta información, ya que necesitaríamos muchos píxeles y a más píxeles, mayor número de pesos vamos a tener en el modelo.

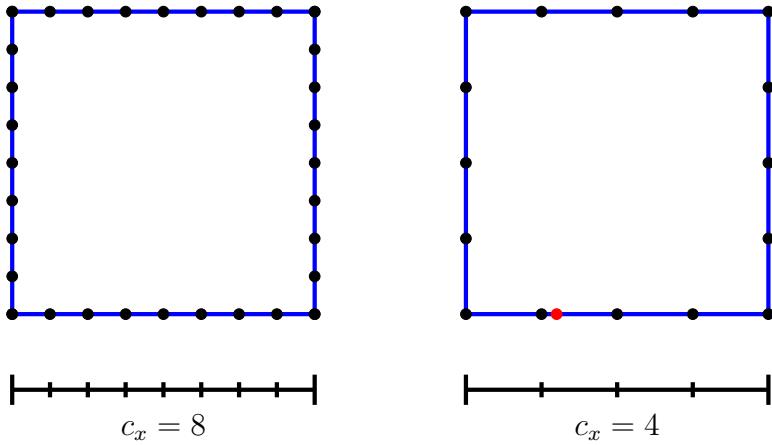
En el diseño de la implementación, es importante parametrizar las variables que influyen en el procesamiento de las imágenes, no solo para tener un seguimiento de las mismas, sino también para poder probar con distintas configuraciones. En concreto, es importante tener como parámetro la resolución final, así como el tamaño de la imagen, ya que son parámetros que pueden influir mucho en los resultados y la eficiencia del modelo.

### 3.2.2. Transformación de los datos a imágenes

En esta sección se va a mostrar cómo vamos a implementar la generación de imágenes, que ha sido realizada desde cero. Se podría haber hecho uso de la librería *matplotlib* simplificando mucho el problema, pero solo serviría en caso de trabajar con las proyecciones. Debido a que vamos a probar distintos enfoques, hemos realizado la implementación desde el principio. Además, los distintos procesamientos parten de la misma idea y modificar la representación es simple en cuanto a implementación. Asimismo, de esta forma tenemos control total sobre las imágenes y todas van a compartir los parámetros.

Los datos proporcionados están en un sistema de referencia donde el eje *z* va de 0 cm a 500 cm y los ejes *x* e *y* van de -200 cm a 200 cm. Para regular la resolución y pasar los datos a una imagen, lo que vamos a hacer es aplicar un cambio de sistema de referencia, así como de unidades. Con estos cambios lo que pretendemos es regular la resolución, así como facilitar la creación de las imágenes.

Para entender la idea que hay detrás de la creación de las imágenes pongamos el siguiente ejemplo. Supongamos que se tiene una regla que solo tiene marcados los centímetros. Si se emplea para medir el lado de un cuadrado que mide 1.2 cm, al colocar la regla ve-



**Figura 3.3:** Esquema gráfico de la discretización de una trayectoria y de la influencia de resolución a la hora de describir una trayectoria. Se muestran como se describirían dos cuadrados cuando se toma una resolución de  $c_x = 8$  y  $c_x = 4$  asumiendo que los cuadrados ocupan todo el volumen posible. El parámetro  $c_x$  denota el número de marcas necesario para el eje  $x$  del volumen completo (ver discusión entorno a la Ec. 3.1).

ríamos que el cuadrado acaba entre la marca con el 1 y con el 2, por lo que podemos decir que mide 1cm con un cierto error, pues vemos que el cuadrado acaba más cerca del 1 que del 2, pero se pasa un poco (punto rojo de la Fig. 3.3). Con las transformaciones que planteamos, lo que hacemos es crear una regla donde el punto más pequeño lo colocamos en el cero y vamos variando el número de marcas ( $c_x$ ) entre los extremos en función de la resolución deseada. De forma que si queremos aumentar la resolución, aumentamos el número de marcas y viceversa. El cambio de sistema de referencia lo realizamos para hacer que el mínimo sea 0 (colocamos la regla en el origen) y el cambio de unidades para fijar el valor del máximo al número de marcas de la regla ( $c_x$ ). De esta forma, tenemos una regla entre los extremos del detector cuyas marcas corresponden a los píxeles de la imagen. En la Fig. 3.3, mostramos gráficamente cómo se describe un cuadrado, que lo podemos entender como el detector, para dos valores de  $c_x$ . Vemos cómo a mayor número de marcas, tenemos una mayor definición de su volumen, pero a cambio se necesita de una mayor cantidad de memoria para recoger toda la información. En la figura también se muestra un punto rojo que hace referencia a un posible *hit* y nos permite visualizar que tras la transformación tenemos que aproximar estos puntos a las marcas.

Otro modo de ver todo esto es mediante la conversión de una señal de analógico a digital, donde se tiene una señal continua y la queremos discretizar. Para realizar la conversión se establece una frecuencia de muestreo (marcas por centímetro). Si la frecuencia con la

que se muestra la señal es alta, tendremos una mayor resolución de la señal, mientras que si establecemos una frecuencia menor, tendremos una peor resolución, ya que se toman menos puntos de la señal analógica, es decir, estamos usando menos marcas. El número total de marcas en un eje lo hemos denotado como  $c_x$ .

Para fijar ideas, supongamos que tenemos solo dos dimensiones y queremos acabar con imágenes  $100 \times 100$  px. Para esto lo que hacemos es normalizar las posiciones al intervalo  $[0, 100]$ . Físicamente, es equivalente a realizar un cambio de referencia, de manera que el valor más pequeño en cada eje pase a ser cero, y luego hacer cambio de unidades para que el valor más grande en los eje pase a valer 100. Finalmente, redondeamos a un entero el resultado obtenido tras la transformación, obteniendo las marcas más cercanas a los puntos de la trayectoria que, además, nos sirve como índice para colocar el píxel en la matriz que representa a la imagen.

De manera formal, tenemos lo siguiente. Sea  $r = (r_x, r_y, r_z)$  un punto cualquiera de la trayectoria de un evento cualquiera. Sean  $t_{min} = (x_{min}, y_{min}, z_{min})$  y  $t_{max} = (x_{max}, y_{max}, z_{max})$  tuplas que recogen los valores más pequeños y más grandes en cada eje, respectivamente. Estas tuplas se obtienen de todos los eventos de entrenamiento. Sea  $c = (c_x, c_y, c_z)$  una tupla con la resolución final de la imagen que contiene al detector completo. Las componentes de  $c$  verifican:

$$c_x = c_y = \frac{5}{4}c_z, \quad (3.1)$$

para que los tres ejes tengan las mismas unidades<sup>6</sup>.

Lo primero que hacemos es aplicar el cambio de referencia (sumar a cada eje una constante a todos los puntos de todos los eventos) con la idea que el punto más pequeño esté en origen:

$$r' = r - t_{min}. \quad (3.2)$$

Ahora, aplicamos el cambio de unidades (multiplicar a cada eje por una constante):

$$r'' = \frac{r'}{t_{max} - t_{min}} \cdot c. \quad (3.3)$$

---

<sup>6</sup>Los valores de las posiciones en los tres ejes están dados en centímetros, pero los rangos son distintos. Por lo tanto, para mantener las dimensiones en los tres ejes tenemos que multiplicar por la misma constante en las tres dimensiones y en la Ec. 3.3 vemos cómo multiplicamos por  $c/(t_{max} - t_{min})$  y, en consecuencia, depende del rango. El factor  $5/4$  surge de que en los ejes  $x$  e  $y$  se divide entre 400 y en el eje  $z$  entre 500.

Finalmente, redondeamos  $r''$  a un entero. Aplicamos esta transformación a todos los eventos y a todos los puntos se obtiene, para cada evento, las posiciones dadas como números enteros cuyo mínimo es el cero y el máximo es  $c$ , obteniendo así los índices de los *hits* en la imagen.

Las imágenes que obtenemos corresponden al detector completo, pero los eventos no ocupan todo el detector, por lo que debemos dar una parte de este volumen para evitar tener regiones vacías, es decir, nos quedamos con una ventana del detector. Además, estas ventanas tienen que ser de igual tamaño para todos los eventos, pero su posición sí depende del evento. Hay varias opciones para colocar la ventana:

- En el “centro de masa” de la cascada que se obtiene usando la carga de los *hits* del evento.
- En el comienzo de las cascadas, ya que sabemos que ahí es donde más diferencias hay entre los dos tipos de cascadas.

Nosotros hemos centrado las ventanas en el origen y el tamaño de esta ventana corresponde al tamaño de las imágenes con las que entrenaremos los modelos de clasificación. Este tamaño será otro parámetro que vamos a poder controlar a la hora de realizar la clasificación.

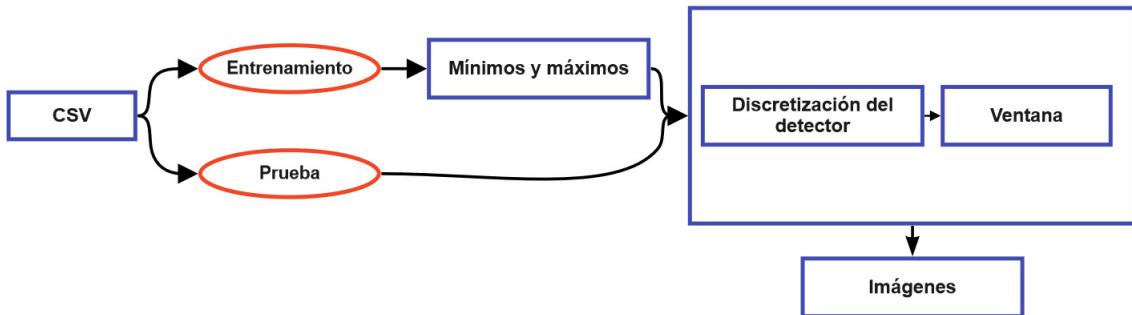
A la hora de colocar la ventana en el origen hay que tener en cuenta que, dependiendo de si el evento ocurre a la izquierda o a la derecha del cátodo, el evento se desarrolla en la dirección positiva del eje  $x$  o en la dirección negativa. Lo que hemos hecho ha sido colocar el origen de la cascada en una de las esquinas inferiores de la imagen, pero la esquina se determina en función de la dirección de avance en el eje  $x$ . Para los otros ejes no se tiene este problema, pues en el eje  $z$  se tiene que siempre se avanza en la dirección positiva y para el eje  $y$  se tienen desplazamientos en ambas direcciones, por lo que colocamos el origen en un punto medio.

### 3.2.3. Comentarios finales

Como comentario final, es posible que, a la hora de hacer el paso de datos tabulares a imágenes, al estar cambiando la resolución, tengamos para el mismo punto del espacio dos

valores distintos de carga (antes eran puntos distintos), por lo que tendremos que elegir un valor para estos puntos donde hay conflicto. Hay distintas opciones como puede ser tomar el máximo o la media, estas opciones también las introducimos como parámetros.

A modo de resumen, la creación de las imágenes consiste en primero obtener los máximos y mínimos de las coordenadas espaciales y de la carga. Con estos valores hacemos las transformaciones descritas obteniendo los índices para colocar los píxeles en la imagen. Tras la transformación, lo que se hace es utilizar alguna de las representaciones propuestas, como quitar uno de los ejes, quedarnos con la imagen tridimensional o hacer la codificación de la 3º dimensión en forma de color. Una vez se tienen los índices colocamos la ventana manteniendo siempre el origen de la cascada. En la Fig. 3.4 se muestra de forma esquemática el flujo que acabamos de describir.



**Figura 3.4:** Esquema del flujo realizado para realizar la transformación de los datos. Se parte de los datos simulados, que los dividimos en un conjunto de entrenamiento y otro de pruebas. Sobre el conjunto de entrenamiento obtenemos los máximos y mínimos de distintas variables necesarios para el procesamiento. Finalmente, creamos las imágenes, para ello discretizamos primero el detector y luego colocamos la ventana.

### 3.3. Modelos propuestos

Una vez se han procesado los datos y tenemos las imágenes, el siguiente paso es aplicar un modelo de clasificación. Vamos a hacer uso de las CNN que, aunque no son la única opción para la clasificación de imágenes, han mostrado muy buenos resultados. En esta sección simplemente vamos a enumerar los distintos modelos propuestos para realizar la clasificación:

- Una CNN simple, con pocas capas que nos sirva de punto de partida.
- Una CNN más compleja, similar a la simple pero más densa y profundas es decir, con más filtros, neuronas, y capas ocultas.
- Hacer uso de arquitectura del estado del arte como pueden ser ResNet o EfficientNet. El entrenamiento lo hacemos usando los pesos aprendidos con el conjunto de datos de ImageNet reentrenando las últimas capas.

Hemos seguido un enfoque incremental donde hemos partido de modelos simples, como pueden ser redes basadas en la arquitectura LeNet, hasta usar redes más profundas y densas con la idea aumentar la complejidad hasta que no se observen mejoras en los resultados.

Uno de los problemas fundamentales que se tienen con el DL es que a medida que se tienen redes complejas, el número de pesos aumenta y esto hace que sea relativamente simple tener sobreajuste. De hecho, una de las limitaciones del DL es la gran cantidad de datos que se necesitan para entrenar los modelos. Este problema ha hecho que hayan surgido métodos de regulación, como el *dropout*, que tratan de lidiar con el sobreajuste. El *dropout* consiste en eliminar aleatoriamente un porcentaje de los enlaces entre dos capas y ha mostrado buenos resultados para reducir el sobreajuste [57].

Otras alternativas para reducir el sobreajuste consisten en introducir pequeñas modificaciones sobre las imágenes para aumentar la variabilidad de los datos, así como la cantidad de imágenes. Estas transformaciones suelen ser rotaciones, desplazamientos, cambios de brillo, reflexiones, etc. Esta técnica se conocen como *data augmentation* y ha mostrado dar muy buenos resultados [58]. Sin embargo, no vamos a utilizarla, ya que trabajamos con simulaciones y podemos, hasta cierto punto, obtener más datos en caso de ser necesario. Además, transformaciones como rotaciones y reflexiones modifican la geometría del detector que es algo no deseado.

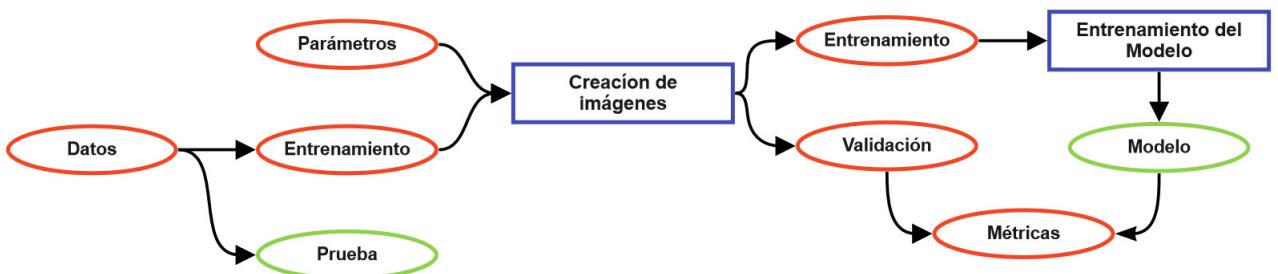
Otra opción, cuando se tienen pocos datos, es hacer uso de modelos que han sido entrenados con conjuntos de datos con muchas imágenes y muchas clases. Los pesos aprendidos, los modificamos ligeramente utilizando nuestro conjunto de datos [59]. La idea detrás de este enfoque consiste en que estas redes han aprendido muchos patrones que se pueden aplicar en otras situaciones. Las CNN se pueden ver como dos grandes bloques: un extractor de características y un MLP para la clasificación. De forma que, al reentrenar modelos, podemos

usar los dos bloques o mantener únicamente el extractor y usar un MLP completamente distinto. Por otro lado, los pesos que se modifican pueden ser todos, o simplemente los pesos de las capas más profundas, al ser las capas que contienen los patrones más complejos y, por tanto, son más específicos, que es posible que no funcionen bien para otro problema.

En definitiva, en este trabajo vamos a proponer varias representaciones de los datos, así como distintos modelos para la clasificación. En el Capítulo 5 describiremos en detalle las arquitecturas usadas, así como las representaciones de los datos que hemos propuesto.

### 3.4. Flujo de trabajo

Sin entrar, todavía, en los detalles de la implementación, en esta sección se pretende mostrar, de forma esquemática, el flujo de trabajo del proceso de entrenamiento de los modelo, desde que se leen los datos hasta que se entrena el modelo. Este flujo está muy simplificado y pretende mostrar de forma general el diseño de esta parte del código. En la Fig. 3.5 se muestra el flujo mencionado.



**Figura 3.5:** Esquema del flujo de trabajo implementado para resolver el problema de clasificación de cascadas electromagnéticas. Los cuadrados son bloques de código mientras que las elipses representan datos, parámetros y modelos. En rojo se indica que forma parte del proceso de entrenamiento y validación mientras que en verde se recoge lo que se despliega tras el entrenamiento.

Inicialmente se parte de los datos, que consisten en 20000 cascadas, la mitad originada por electrones y la otra mitad originada por fotones. Lo primero que hacemos es separar el conjunto completo en un 90 % para entrenamiento y en un 10 % prueba, de manera que nos aseguramos que no se utiliza el conjunto de prueba en ningún momento. Además, es necesario fijar distintos parámetros que definen el tratamiento de los datos y el entrenamiento. En este punto, pasamos los datos tabulares a imágenes. Estas imágenes las

volvemos a separar en un conjunto de entrenamiento y otro de validación (90 %-10 %). Con las imágenes de entrenamiento, entrenamos un modelo de clasificación definido en función de los parámetros establecidos y evaluamos el modelo usando el conjunto de validación. El modelo final es guardado junto con los parámetros para su uso futuro.

Este flujo mencionado está muy simplificado, veamos algunas cosas que son importantes mencionar. A la hora de entrenar el modelo, puesto que estamos usando redes neuronales, se dan varias vueltas por el conjunto de entrenamiento. En cada vuelta, validamos y continuamos entrenando hasta que no se observan mejoras en los resultados del conjunto de validación. Por otro lado, en el procesamiento de los datos necesitamos obtener el mínimo y máximo de varias magnitudes, por lo que, al aprenderlas con el conjunto de entrenamiento, estas tienen que guardarse con el modelo para poder procesar imágenes futuras.

Hay que tener en cuenta que tenemos un conjunto de parámetros que vamos a tener que explorar para mejorar los resultados. Para esto, utilizamos la librería Optuna [50] que nos permite realizar esta búsqueda de una forma más eficiente que haciendo una búsqueda de rejilla, al tratar de optimizar una función objetivo. La implementación consiste en una pequeña modificación del flujo, donde hay que definir una nueva función objetivo y en función de los resultados obtenidos tras la validación se modifican los parámetros iterativamente.

En la Fig. 3.6 mostramos, en código, el flujo de entrenamiento, es decir, todo menos la evaluación del modelo sobre el conjunto de prueba, que se hace aparte. Lo primero que hacemos es fijar la semilla aleatoria y configurar MLFlow fijando el lugar donde se guardan los modelos y el nombre del experimento. Luego obtenemos los datos y los elementos necesarios para el modelo y su optimización. Finalmente, entrenamos y guardamos los resultados en MLFlow. Este es el cauce principal y él más simple. Tenemos otros cauces, en los que no vamos entrar, que implementan la búsqueda de hiperparámetros o el estudio de los modelos usando en los distintos conjuntos de datos entre otros. Todo el código está disponible en GitHub<sup>7</sup>.

Por otro lado, como ya se mencionó, hemos hecho uso de técnicas de MLOps que no se recogen completamente en el flujo descrito. Las prácticas MLOps que hemos llevado a cabo se pueden resumir en:

---

<sup>7</sup>[https://github.com/aponce1509/tfm\\_](https://github.com/aponce1509/tfm_)

- Encapsulación de las distintas partes del flujo de manera que cualquier modificación realizada no afecte al flujo completo, teniendo así un sistema de cauces que nos permite un desarrollo cómodo. En el código observamos cómo, por un lado se obtienen los conjuntos de entrenamiento, y por otro lado, se entrena el modelo. Esta filosofía se ha mantenido durante todo el desarrollo, pues simplifica mucho el diseño y la implementación de nuevas funcionalidades.
- Todos los modelos entrenados, junto con sus resultados y parámetros, han sido guardados haciendo uso de MLFlow. Una herramienta que facilita mucho tanto el despliegue como el seguimiento de los resultados.
- Con respecto al despliegue, se ha construido un flujo que parta de datos nuevos (en nuestro caso el conjunto de prueba) y el identificador del modelo aprendido, y clasifique los datos. También se han añadido funciones que nos permiten estudiar los modelos, así como sus resultados.
- Aunque el trabajo no ha sido desarrollado por un equipo, donde es obligatorio establecer un repositorio central para recoger las métricas y modelos, hemos tenido que establecerlo y configurar MLFlow al trabajar con clusters de ordenadores, que han sido necesarios para realizar todas las pruebas llevadas a cabo en este trabajo.
- Se ha hecho la implementación para que al efectuar el entrenamiento, solo sea necesario indicar los parámetros de los modelos y del preprocesamiento. Para la evaluación de nuevos datos solo se requiere indicar un identificador del modelo que se quiere usar.



The screenshot shows a Jupyter Notebook cell with three colored status indicators (red, yellow, green) at the top. The cell contains the following Python code:

```
1 def basic_pipeline(params, is_optuna=False, optuna_step=None,
2                     verbose=True, trial_params=None, experiment_name="Default",
3                     opt_id=None, trial=None):
4     fix_seed(params)
5     mlflow.set_tracking_uri(MODEL_PATH)
6     mlflow.set_experiment(experiment_name)
7     with mlflow.start_run(nested=True):
8         model, device, optimizer, scheduler, criterion = get_gradient_elements(
9             params
10        )
11        train_data, val_data = get_data_validation(params)
12
13        val_loss, metrics, _id = train_loop(
14            train_data=train_data,
15            val_data=val_data,
16            model=model,
17            optimizer=optimizer,
18            criterion=criterion,
19            scheduler=scheduler,
20            device=device,
21            params=params,
22            verbose=verbose,
23            is_optuna=is_optuna,
24            optuna_id=opt_id,
25            trial=trial
26        )
27        mlflow_log(params, is_optuna, metrics, _id)
28
29    mlflow_log_optuna(is_optuna, trial_params, val_loss, optuna_step)
30
31    return val_loss
```

**Figura 3.6:** Función con el flujo de entrenamiento del modelo. Hemos quitado los comentarios para mostrar el código de manera más clara.

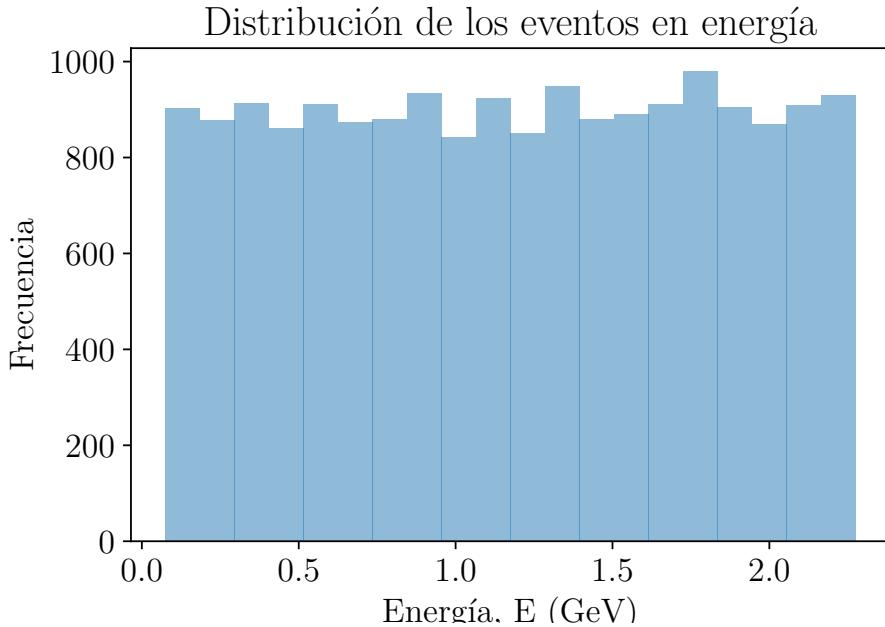
# Capítulo 4

## Análisis de los datos

En este capítulo vamos a analizar los datos con la idea de poder entenderlos mejor y justificar las decisiones tomadas sobre cómo procesar los datos y cómo crear las imágenes según la metodología comentada en el capítulo anterior. Asimismo, nos permite mostrar el efecto de la resolución y el tamaño de las imágenes en cuanto a qué cantidad de cascada se pierde. En concreto, nos centraremos en estudiar el rango y distribución de las distintas variables.

Como ya se ha mencionado, una de las variables que tenemos para cada evento es la energía de la partícula que origina la cascada. Recordemos que esta variable se tiene gracias a que trabajamos con simulaciones y solo se usará para hacer el estudio de los resultados obtenidos. En la Fig. 4.1 se muestra un histograma de la distribución en energía donde vemos que se tiene una distribución uniforme, es decir, no hay rangos de energía donde se tengan más eventos. Los datos se han generado con esta distribución intencionadamente porque nos permite entrenar con la misma cantidad de eventos todo el espectro de energías. La dependencia con la energía es crucial para investigar la física que ocurre en las cascadas, por lo que será necesario evaluar las métricas de los modelos en función de la energía para estudiar cómo afectan al rendimiento de los modelos de clasificación.

Otra variable que resulta importante comprender es la posición. En concreto, nos interesa saber el volumen del detector ocupado por los eventos, ya que nos permite entender el efecto que tienen la resolución y el tamaño de la ventana en la imagen. Para obtener el volumen agrupamos los datos por eventos y calculamos el rango en los distintos ejes



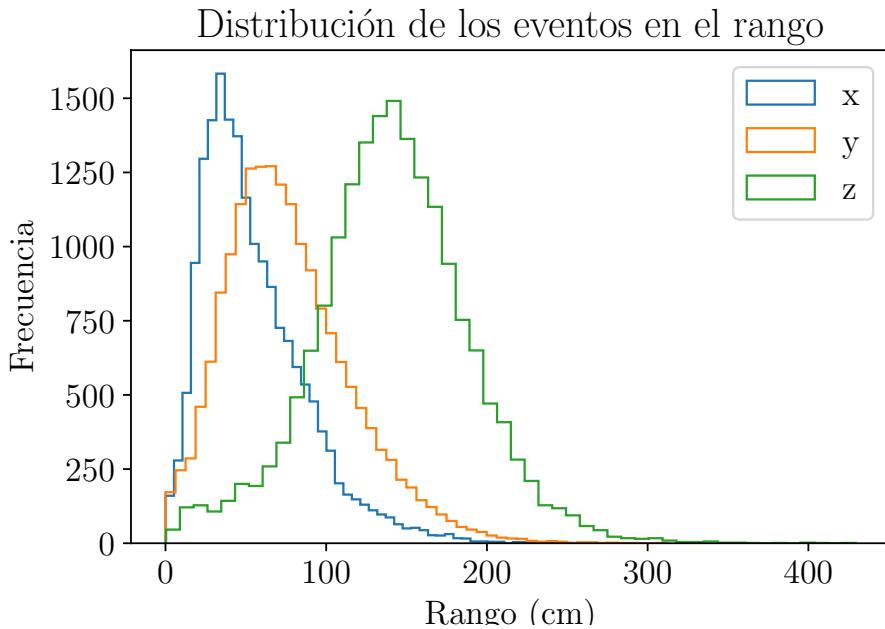
**Figura 4.1:** Histograma de la energía de los distintos eventos.

para cada evento. Es decir, para cada evento restamos el máximo y el mínimo en cada eje, obteniendo el rango para los tres ejes y para cada evento. A estas nuevas variables las vamos a denominar simplemente rangos. En la Fig. 4.2 se muestran los histogramas para estas nuevas variables y en la Tabla 4.1 mostramos los cuartiles.

	hitsX (cm)	hitsY (cm)	hitsZ (cm)
min	0.04	0.00	0.40
25 %	31.26	48.79	111.90
50 %	47.47	71.12	140.50
75 %	72.23	98.97	170.15
max	263.51	312.29	429.40

**Tabla 4.1:** Cuartiles de los rangos que alcanzan las cascadas en los distintos eventos y ejes.

Antes de entrar a discutir estas gráficas, recordemos que el parámetro  $c$  nos permite regular la resolución con la que se describe el volumen completo del detector. Supongamos que queremos tener una precisión de 0.1 cm, es decir, construir una imagen donde la distancia entre dos píxeles consecutivos equivalga a 0.1 cm. En este caso, teniendo en cuenta las dimensiones del detector:  $x \in [-200, 200]$  cm,  $y \in [-200, 200]$  cm y  $z \in [0, 500]$  cm el valor de  $c$  tendría que ser  $(4000, 4000, 5000)$  px. Básicamente, lo que tenemos que hacer es fijar una precisión y dividir el rango de los ejes por esta precisión. Este valor de  $c$  es la

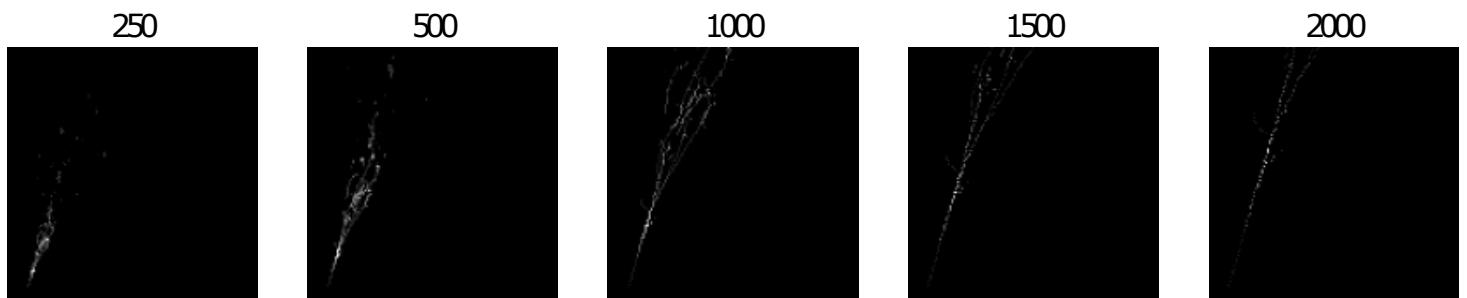


**Figura 4.2:** Histograma de los rangos. Los rangos hacen referencia a la distancia entre el *hit* que ocurre en el valor más pequeño y él que ocurre en el valor más grande. Se obtienen para los tres ejes espaciales y para cada evento.

resolución y significa que representamos el volumen completo del detector con una malla de  $4000 \times 4000 \times 5000$  px. Un evento dentro de este volumen ocupara solo una parte y lo que hacemos es coger una ventana del volumen completo.

Si nos queremos asegurar de que no se pierde nada de información al tomar la ventana, tenemos que ver cuál es el rango máximo que puede darse en un evento, que lo podemos ver en la Tabla 4.1. En el caso de que la precisión sea de 0.1 cm, tendríamos que usar imágenes de  $2636 \times 3123 \times 4294$  px que son imágenes muy grandes. Sin embargo, si nos conformamos con mantener la información completa del 75 % de los eventos, el tamaño de la ventana sería de  $723 \times 990 \times 1701$  px que siguen siendo imágenes muy grandes, pero son considerablemente más pequeñas. A la hora de entrenar los modelos, vamos a tener que jugar con los parámetros de resolución y de tamaño de la ventana para regular la cantidad de información que se mantiene al crear las imágenes. Es importante mencionar

que la información que se pierde es, en su mayoría, la parte final de la cascada<sup>1</sup>, mientras que el origen se mantiene siempre. Lo hacemos de esta forma, ya que en origen de la cascada es donde se tienen las principales diferencias entre las cascadas electromagnéticas que queremos clasificar. En la Fig. 4.3 se muestra el mismo evento para distintos valores de resolución, donde vemos como a para valores muy bajos tenemos toda la cascada pero con muy poca definición de la cascada. Sin embargo, a medida que se aumenta la resolución, vemos cómo se va definiendo la cascada, pero vamos perdiendo información. El tamaño de las imágenes mostradas es de  $128 \times 128$  px.



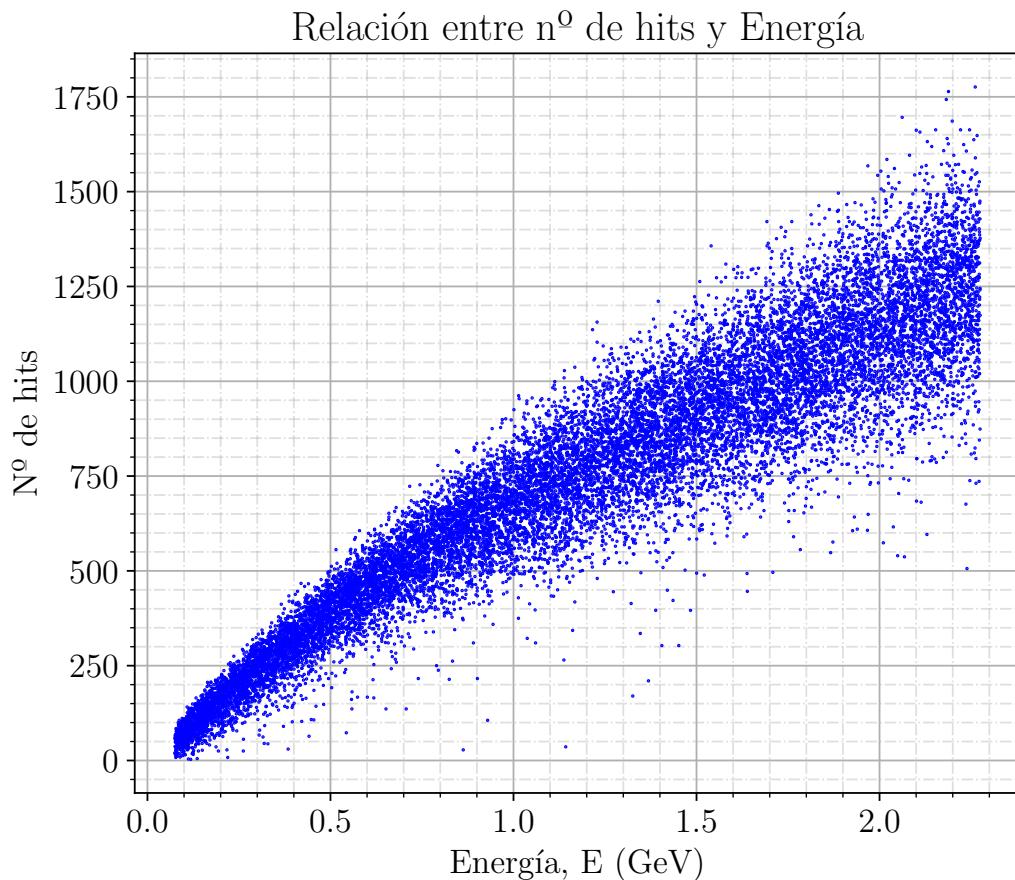
**Figura 4.3:** Mismos eventos representados para distintas resoluciones (número de píxeles que describen el detector completo). El tamaño de las imágenes es de  $128 \times 128$  px.

Por último, vamos a estudiar los *hits* y su energía. La cantidad de *hits* que se tienen por evento depende de la energía del suceso. En la Fig. 4.4 se muestra la relación que hay entre el número de *hits* de un evento y la energía del mismo donde observamos que hay una tendencia positiva, es decir, a más energía mayor número de hits. El problema se tiene sobre todo cuando se tienen pocos *hits*. Hay ocasiones donde un evento está dado por entre 5 y 10 *hits*, siendo el evento con menos *hits* uno que viene dado por 3 *hits*. Es interesante ver cómo se comportan nuestros modelos en rangos de energía bajos.

En la Fig. 4.5 se muestra la distribución de la carga de los *hits*. Se muestran dos distribuciones, una donde lo que hemos hecho ha sido normalizar dividiendo por el valor máximo de la carga, así como la distribución tras aplicar una transformación logarítmica. En esta gráfica vemos que parece que, a simple vista, la distribución de los *hits* sigue

---

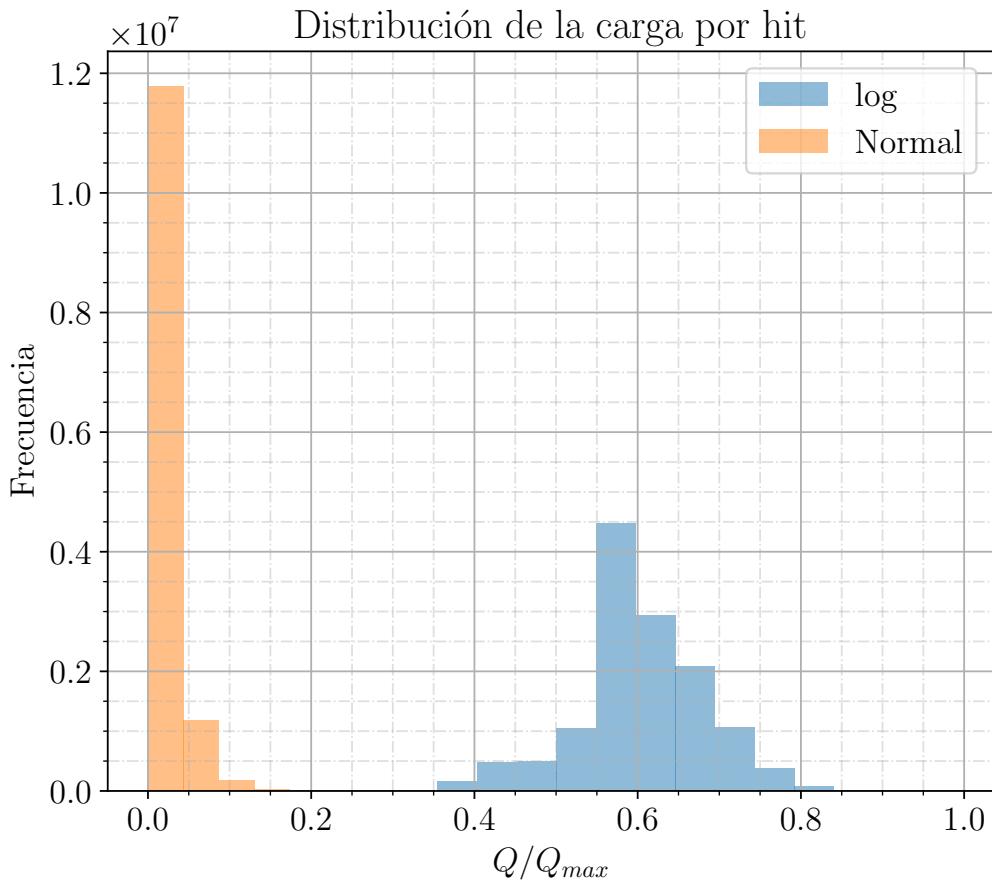
<sup>1</sup>Debido a la geometría del problema, tras originarse las cascadas, estas se desarrollan de menor  $z$  a mayor  $z$  y en el eje  $x$  depende de si ocurre a la derecha o la izquierda del cátodo porque siempre se dirigen hacia el ánodo (recordar Fig. 2.5 y 2.6). Al generar las imágenes, las centramos en el origen colocando el origen en una esquina, de manera que se recoja la dirección de avance. Sin embargo, puede darse que, en algunas ocasiones, las partículas emitidas durante la interacción se dirijan en la dirección contraria, dando situaciones en las que se puede perder un poco de información no correspondiente al final de la cascada.



**Figura 4.4:** Diagrama de dispersión donde se muestra la relación entre el número de *hits* de un evento y la energía. El número mínimo de *hits* en un evento es de 3 *hits*.

una distribución exponencial, donde tenemos que la mayoría de *hits* depositan poca carga, mientras que muy pocos tienen valores de carga mayores. Notar que, al haber dividido por el máximo, hay al menos un *hit* con valor 1 en la carga normalizada. Sin embargo, vemos que tras aplicar la transformación se pierde esta distribución exponencial y tenemos una distribución que en ciertas ocasiones puede ser deseable al estar los *hits* más distribuidos.

Este comportamiento es importante tenerlo en mente, ya que en algunos casos puede que los *hits* más energéticos tengan valores tan grandes que, en comparación, los menos energéticos no tengan nada de importancia. Una forma de solucionar este problema es aplicar la transformación logarítmica mostrada. A la hora de entrenar los modelos, vamos a tener que considerar esta transformación porque pueden llevar a una mejora de los re-

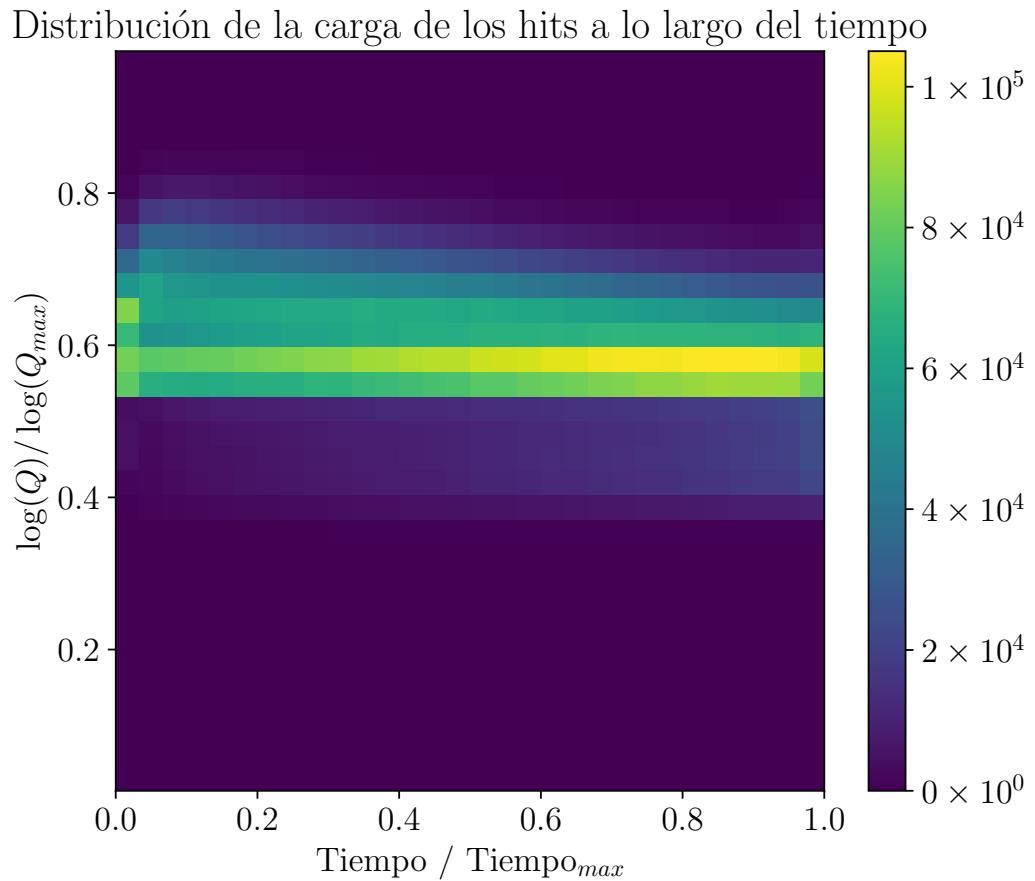


**Figura 4.5:** Histogramas de la carga de los *hits* que se observan en los distintos eventos. Se muestran los datos sin aplicar ninguna transformación y los datos tras aplicar una transformación logarítmica.

sultados. Además, esta distribución exponencial puede dar lugar a que, al tener mucha diferencia entre los valores donde se encuentran la mayoría de *hits* y el máximo, se tengan las imágenes prácticamente negras.

Otro factor que es interesante observar en la carga, es su distribución en función del tiempo, ya que es conocido que al comienzo del evento se deben observar valores más altos para la carga. Aunque no se dispone de la variable tiempo, se puede inferir al tener los datos ordenados temporalmente. En la Fig. 4.6 se muestra un histograma bidimensional donde en el eje  $x$  se tiene el tiempo y en el eje  $y$  se muestra el logaritmo de la carga. Hay que tener en cuenta que el tiempo se ha normalizado para cada evento, es decir, un valor de

tiempo igual a 0 equivale al comienzo del evento y un valor de 1 equivale al final. De igual forma, la carga se ha normalizado usando el valor máximo observado en el conjunto de entrenamiento. Vemos que la distribución en el tiempo es prácticamente constante, salvo al principio, donde observamos *hits* con energías superiores.



**Figura 4.6:** Histogramas bidimensional donde se muestra la distribución de hits en función de la carga y el tiempo. Ambas variables las hemos normalizado para que tomen valores entre cero y uno.

# Capítulo 5

## Desarrollo de los modelos para la clasificación

En este capítulo vamos a recoger y desarrollar todas las propuestas que se plantean en este trabajo en lo referente a la representación de los datos y la construcción del modelo de clasificación. La construcción de las imágenes datos ha sido discutida en los Capítulos 3 y 4. Primero haremos una descripción detallada de las distintas representaciones diseñadas necesarias para que los modelos entiendan los datos. Asimismo, mostraremos las arquitecturas de los distintos modelos que vamos a emplear. También, se muestran las métricas que emplearemos para realizar el entrenamiento y validar el modelo. Finalmente, se listarán todos los parámetros que se tienen en el código desarrollado y permiten modificar el preprocesamiento de los datos y el entrenamiento del modelo.

### 5.1. Representaciones de los datos

Como ya hemos visto, hay muchas opciones a la hora de crear las imágenes desde los datos tabulares. En esta sección vamos a describir las distintas representaciones que hemos propuesto. Todas estas representaciones parten de hacer el cambio de unidades y de sistema de referencia que se ha descrito en la Sección 3.2. En general, todas las propuestas buscan

una forma de tratar con la tercera dimensión espacial, que no se puede recoger en una imagen típica.

## Representación tridimensional

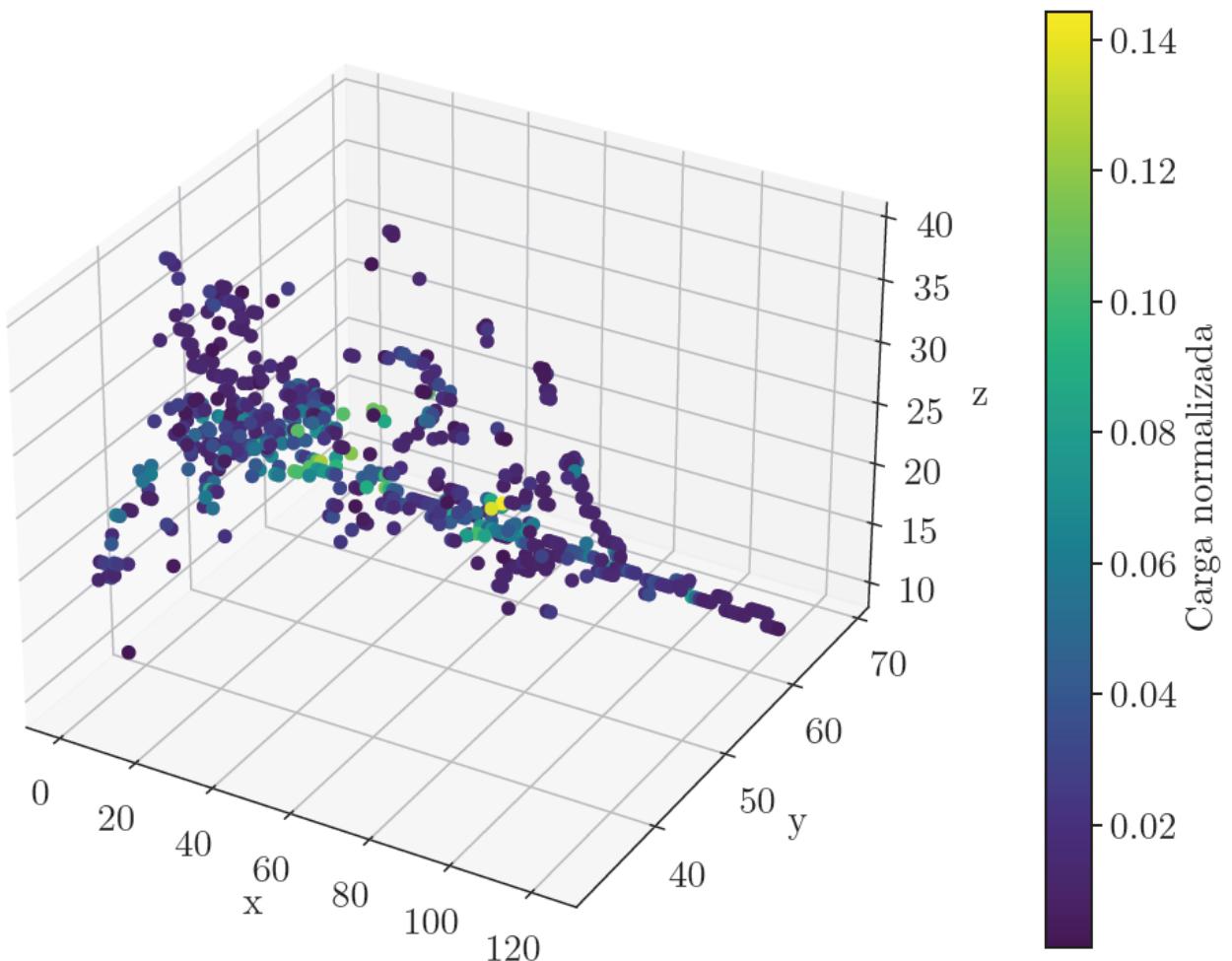
Una forma de abordar el problema es quedarnos con toda la información y no descartar ninguna dimensión porque la operación de convolución no está limitada a trabajar con dos dimensiones espaciales. Esta representación es conceptualmente muy simple, pero surgen una serie de problemas relacionados con el tamaño de las imágenes y que no se tienen modelos complejos previamente entrenados, como sí ocurre para las imágenes bidimensionales. En la Fig. 5.1 se muestra gráficamente el resultado de usar esta representación sobre uno de los eventos.

## Representación bidimensional, una proyección

Otra opción es proyectar la imagen tridimensional y quedarnos con una de las vistas obteniendo una imagen bidimensional con un solo canal de color. Si recordamos, el eje  $x$  es el eje de deriva, el haz de neutrinos está dirigido en la dirección del eje  $z$  (el momento de la partícula tiene la dirección del eje  $z$ ) y eje  $y$  simplemente la altura con la que llega el neutrino al detector.

La implementación de esta representación consiste en quitar la columna del eje que se quiera proyectar. Notar que, entonces, puede darse que se tengan *hits* que tengan la misma posición tras realizar la proyección. Cuando se tienen conflictos, nos quedamos con la media o con el máximo de la carga. Por lo demás, se aplican los cambios de unidades y de sistema de referencia que se describieron en la Sección 3.2.2. En la Fig. 5.2 se muestra gráficamente un evento que ha sido obtenido usando los datos tabulares directamente, así como la imagen con la representación que se acaba de describir. Los parámetros que hemos utilizado para la creación de la imagen procesada es de  $c_x = 1000$  px para la resolución, un tamaño de ventana de  $128 \times 128$  px y no se ha aplicado la transformación logarítmica. En la Fig. 5.3 se muestran varios eventos con esta representación, tanto para cascadas electrónicas como fotónicas.

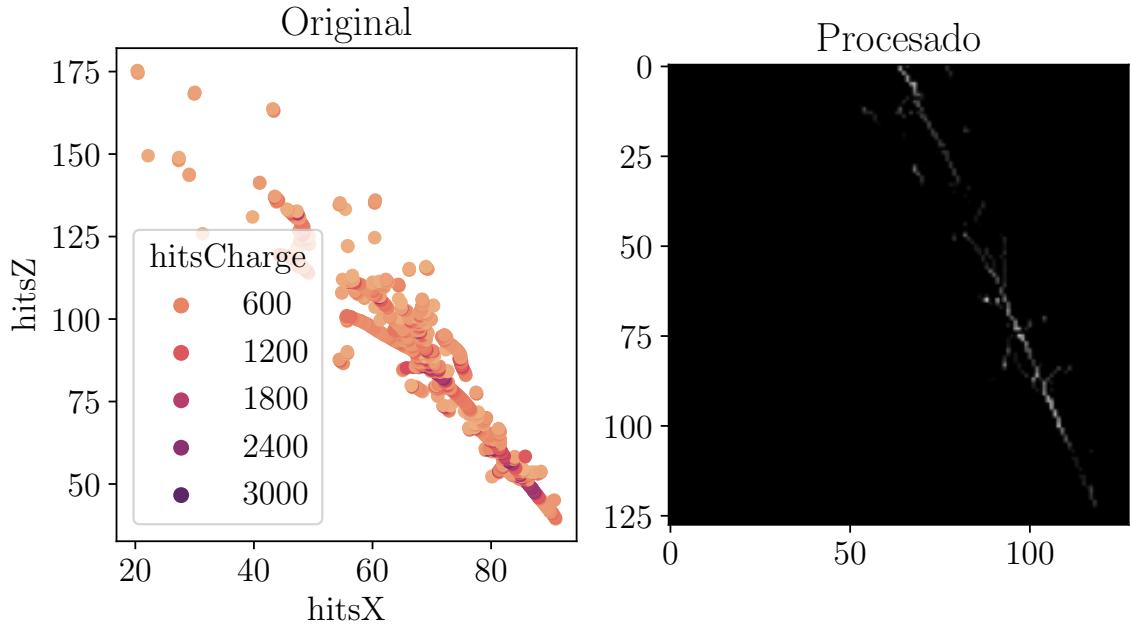
## Representación tridimensional



**Figura 5.1:** Evento procesado manteniendo las tres dimensiones espaciales. Se ha tomado una resolución  $c_x$  de 500 px y tamaño  $128 \times 128 \times 128$  px.

## Representación bidimensional, codificación del color

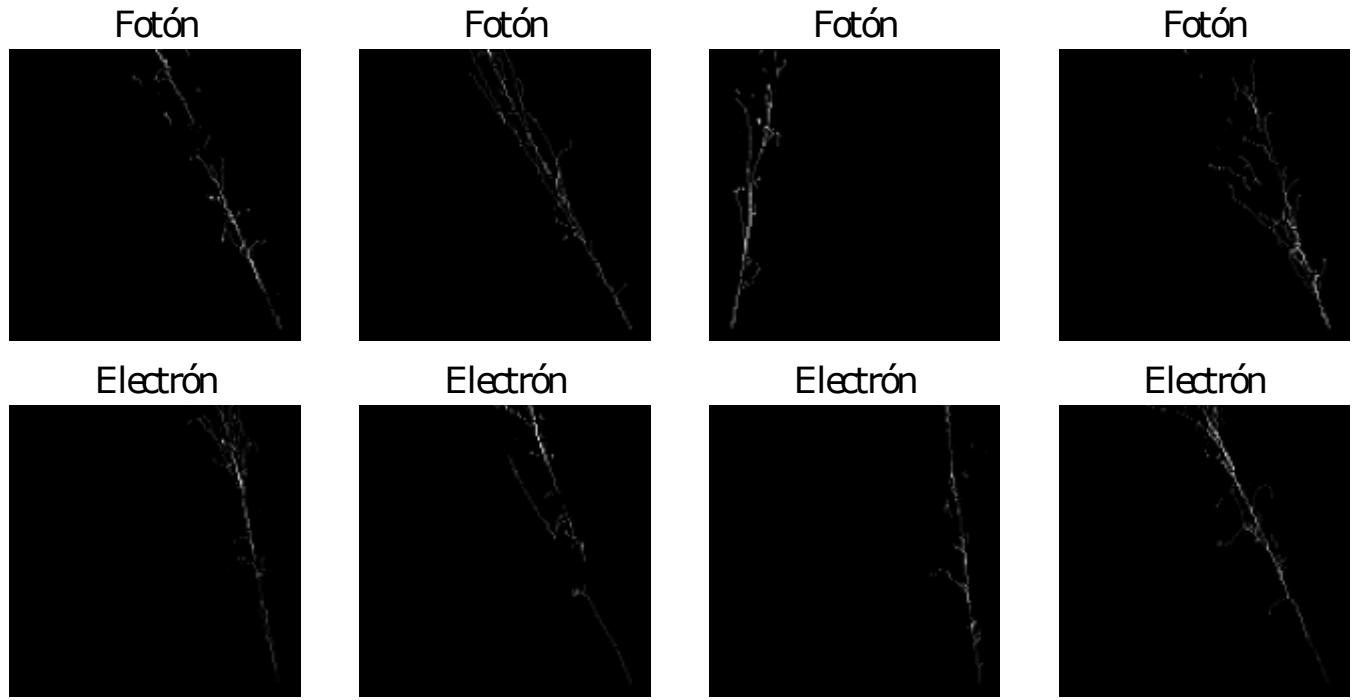
Independientemente del eje que proyectemos vamos a perder información, ya que las cascadas se desarrollan en las tres dimensiones espaciales, incluso al proyectar el eje  $y$  que es con el que se pierde menos información. Una forma de mantener esta información es codificar de alguna forma el color de la imagen para que contenga esta información.



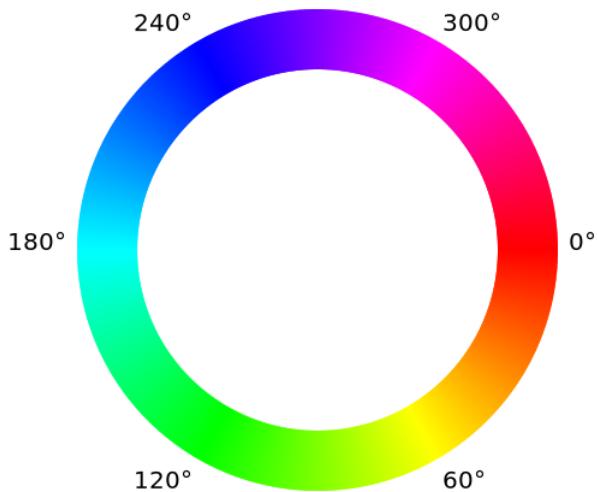
**Figura 5.2:** En la izquierda se muestra un evento proyectado obtenido de los datos tabulares directamente sin hacer uso del procesamiento que presentamos. En la derecha se muestra la imagen procesada con una resolución de 1000 px y tamaño  $128 \times 128$  px.

Hasta ahora, hemos estado trabajando con imágenes en blanco y negro donde solo se tiene un canal. En el caso de trabajar con imágenes que tengan color es necesario trabajar con los tres canales de color. Hay distintos espacios para recoger el color de una imagen, el más común es el RGB donde los canales recogen la cantidad de rojo, verde y azul en la imagen. Hay otros muchos espacios y en función de lo que se quiera hacer con la imagen será más adecuado usar uno u otro. Nosotros vamos a trabajar en el espacio de color HSL, donde la H hace referencia al *hue* (el color), la S a la saturación y la L a la luminosidad. En la Fig. 5.4 se los colores para los distintos valores de *hue*.

Para mantener toda la información a la hora de hacer la proyección, tenemos que codificar la posición de los *hits* en el eje proyectado, así como agregar la carga y posición de los distintos *hits* que se superpongan. Lo que se propone es fijar la saturación, y codificar la posición y la carga en el color y la en luminosidad, respectivamente. En caso superposición, la agregación la realizamos haciendo la suma, tanto para la posición como para la carga. De esta forma, tenemos las dos variables espaciales no proyectadas y las sumas de la posición y carga de los *hits* que, en el mismo evento, tengan la misma posición en la proyección. Estas dos nuevas variables nos darán el *hue* y la luminosidad de la imagen.



**Figura 5.3:** Imágenes obtenidas tras procesar y quedarnos con la proyecciones en el eje  $y$  para distintos eventos. Las imágenes han sido procesadas con una resolución de 1000 px y tamaño  $128 \times 128$  px.



**Figura 5.4:** Rueda con los colores asociados al valor de *hue* fijados la saturación y la luminosidad [60].

Veamos la implementación de esta representación formalmente. Sea  $X$  el conjunto de datos de partida donde cada instancia hace referencia a un *hit*. El conjunto de datos tiene

distintas variables:  $X_e$ , determina el evento del hit,  $X_x$ ,  $X_y$  y  $X_z$ , indican la posición, y  $X_c$  recoge la carga. Para fijar ideas, supongamos que queremos proyectar el eje  $y$ . Para ello, agrupamos<sup>1</sup> por evento, posición en  $x$  y posición en  $z$ , y, tras agrupar, sumamos tanto la carga como la posición en el eje  $y$ . A continuación, reescalamos las nuevas variables obtenidas para que el máximo de las nuevas posiciones sea  $180^2$ , al representar el *hue*, y el nuevo valor de la carga sea 1, al representar la luminosidad.

Sea  $X^i (e = e_n, x = x_m, z = z_p)$  un *hit* que pertenece al evento  $e_n$  y está en la posición  $(x_m, z_p)$ . Definimos  $\hat{X}_k$  como:

$$\hat{X}_k (e = e_n, x = x_m, z = z_p) = \sum_i X_k^i (e = e_n, x = x_m, z = z_p) \quad \forall n, m, p. \quad (5.1)$$

Esto es una forma de describir matemáticamente la operación de agrupar por evento y por las posiciones no proyectadas. De esta forma, transformamos los datos a  $\hat{X}$  donde las variables por la que hemos agrupado se mantienen iguales y las variables  $X_c$  y  $X_y$  se han modificado al sumarse aquellas instancias con los mismos valores en las variables agrupadas. Una vez hecho el agrupamiento, se cambian los rangos de  $X_c$  y  $X_y$  para que coincida con el *hue* y con la luminosidad.

Sin embargo, surgen un par de problemas. Por un lado, si hacemos justo lo que se acaba de mostrar, si tenemos dos eventos idénticos pero desplazados en el eje de proyección, vamos a tener imágenes con colores distintos y esto es algo que no queremos que ocurra. Para solucionarlo, cambiamos el sistema de referencia para cada evento para que el mínimo pase a ser cero.

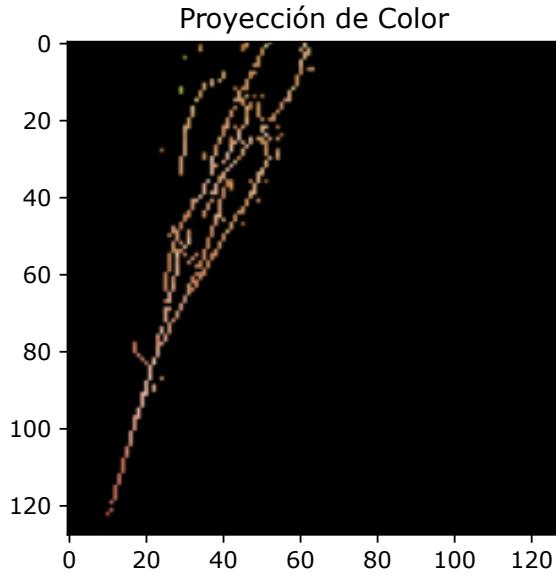
El otro problema está relacionado con la distribución exponencial que se observaba en la carga (ver Fig. 4.5) y que se agrava al realizar la suma. Esto hace que el valor de la luminosidad para la mayoría de eventos sea tan pequeño que, al realizar la imagen, se tenga que los *hits* no se observan. Para solucionar esto lo que hacemos es tomar logaritmo tras realizar la suma.

---

<sup>1</sup>Agrupar consiste en coger todas las instancias que comparten valor en una o varias variables y hacer una operación sobre las variables restantes como puede ser tomar el máximo.

<sup>2</sup>Normalmente, el rango del *hue* va de  $0^\circ$  a  $360^\circ$ , sin embargo, nosotros vamos a usar Open CV que fija el máximo en  $180^\circ$ .

Con esta representación esperamos que el origen de la cascada tome valores pequeños de *hue* y a medida que se desarrolla la cascada y avanza hacia valores mayores del eje *y* vaya aumentando su valor, pasando de rojo a naranja. Cuando se superpongan *hits* observaremos saltos en el color al estar sumando las posiciones. La luminosidad de la trayectoria la determina la carga y cuando se tenga superposición, también se deben observar saltos en la luminosidad, ya que corresponde a la suma de varios valores. En la Fig. 5.5 mostramos un ejemplo de un evento tratado con esta representación.



**Figura 5.5:** Evento procesado codificando la tercera dimensión en el color de la imagen. Las imágenes han sido procesadas con una resolución de 1000 px y tamaño 128 × 128 px.

## Representación para Transformers

En este trabajo nos vamos a centrar sobre todo en el tratamiento de imágenes y la clasificación se ha realizado utilizando imágenes. Sin embargo, también vamos a plantear una representación distinta, cuya aplicación dejamos como trabajo futuro. Las cascadas tienen un marcado carácter temporal, ya que tenemos un punto inicial donde se origina y, a partir de este punto, esta se va desarrollando. Además, sabemos que al principio es donde se tiene la mayor parte de la información que nos permite hacer la discriminación. Los transformers son una arquitectura que han mostrado muy buenos resultados con textos, que son una estructura de datos donde el orden de las palabras es fundamental.

Estas redes se basan en hacer uso de mecanismos de atención y en codificar la posición de las palabras. Estos modelos se han convertido en el estado del arte del procesamiento de lenguaje natural y nuestra propuesta consiste en utilizar los datos tabulares directamente. Para ello codificamos las instancias temporalmente. En la Fig. 5.6 mostramos gráficamente como proponemos tratar los datos. Si hacemos una analogía con un texto, la idea es que cada evento sea una frase que se le pase al transformer y cada *hit* una palabra. La “frase” de *hits* la construimos ordenándolos temporalmente. Los *hits* los representamos con vectores de cuatro elementos, las tres coordenadas espaciales y la carga, al igual que las palabras se representan como vectores.

Una alternativa, a medio camino entre los métodos presentados, es hacer uso de los Visual Transformers (ViT), que hacen uso de los transformers para trabajar con imágenes. Este modelo particiona la imagen en bloques que se usan como entrada para el transformer. Sin embargo, nuestros datos admiten ser tratados directamente por transformers sin necesidad de pasar por la construcción de la imagen. Por lo tanto, esta forma de funcionar sería mucho más eficiente que el tratamiento con imágenes:

- La memoria requerida para almacenar los eventos es muy baja, pues solo necesitamos guardar tantos vectores de cuatro elementos como *hits* tenga el evento.
- Con esta representación no se descarta nada de información, puesto que, al contrario que para la generación de imágenes, no necesitamos realizar la proyección ni tomar una ventana del detector.

## 5.2. Arquitecturas

En esta sección vamos a mostrar las distintas arquitecturas que vamos a emplear para realizar la clasificación de las imágenes. Para esto vamos a usar redes diseñadas desde cero, así como arquitecturas del estado del arte ya entrenadas, donde se modifican tanto los pesos como el clasificador. Las arquitecturas que usaremos serán: EfficientNet, ResNet y GoogleNet. Además, nos hemos basado en LeNet para diseñar una red simple y una más compleja.

Evento	tantas columnas como <i>hits</i> tenga el evento y ordenados temporalmente				
1	x	x <sub>1</sub>	x <sub>2</sub>	x <sub>3</sub>	
	y	y <sub>1</sub>	y <sub>2</sub>	y <sub>3</sub>	
	z	z <sub>1</sub>	z <sub>2</sub>	z <sub>3</sub>	
	q	q <sub>1</sub>	q <sub>2</sub>	q <sub>3</sub>	
2	x	x <sub>1</sub>	x <sub>2</sub>	x <sub>3</sub>	
	y	y <sub>1</sub>	y <sub>2</sub>	y <sub>3</sub>	
	z	z <sub>1</sub>	z <sub>2</sub>	z <sub>3</sub>	
	q	q <sub>1</sub>	q <sub>2</sub>	q <sub>3</sub>	
3	x	x <sub>1</sub>	x <sub>2</sub>	x <sub>3</sub>	
	y	y <sub>1</sub>	y <sub>2</sub>	y <sub>3</sub>	
	z	z <sub>1</sub>	z <sub>2</sub>	z <sub>3</sub>	
	q	q <sub>1</sub>	q <sub>2</sub>	q <sub>3</sub>	

**Figura 5.6:** Representación gráfica de los datos preparados para el uso de transformers en clasificación de cascadas electromagnéticas. Para cada evento se tienen tantos vectores como *hits* y cada vector contiene las coordenadas espaciales (*x*, *y*, *z*) y la carga del *hit* *q*.

Independientemente de la arquitectura que se use, es necesario fijar distintos hiperparámetros del algoritmo de aprendizaje, como el ritmo de aprendizaje (*lr*), que pueden ser cruciales de cara a la convergencia y mejora de los resultados.

Como ya hemos dicho, vamos a emplear Pytorch para el diseño y entrenamiento de los modelos. Para esto, primero hay que definir un optimizador, como puede ser SGD o Adam, y, como vamos a considerar un *lr* dinámico, necesitamos establecer cómo lo vamos a modificar a medida que avance el entrenamiento. Con estos elementos, se implementa el ciclo de aprendizaje del modelo, donde primero se usan todos los *batchs* y se va repitiendo el proceso varias veces (es lo que se conoce como épocas). Cada vez que se completa una época, se validan los resultados con un conjunto de datos no visto por el modelo durante el entrenamiento y, a medida que se avanza en el entrenamiento, se van guardando los pesos aprendidos. Solo se guardan si se mejoran los resultados en validación, es decir, que los pesos del modelo guardado tras el entrenamiento son aquellos con los que mejores resultados se han obtenido en validación.

Con la intención de no tener que fijar un número de épocas, lo que hemos hecho ha sido implementar lo que se conoce como parada temprana. Consiste en que si, tras un número consecutivo de épocas, hay mucho sobreajuste o el modelo no mejora en validación, se para el ciclo de aprendizaje y se guardan los pesos con los que mejores resultados se hayan

obtenido. De esta forma, no es necesario fijar el número de épocas, pues el entrenamiento se interrumpirá en el momento que ya no se estén mejorando los resultados.

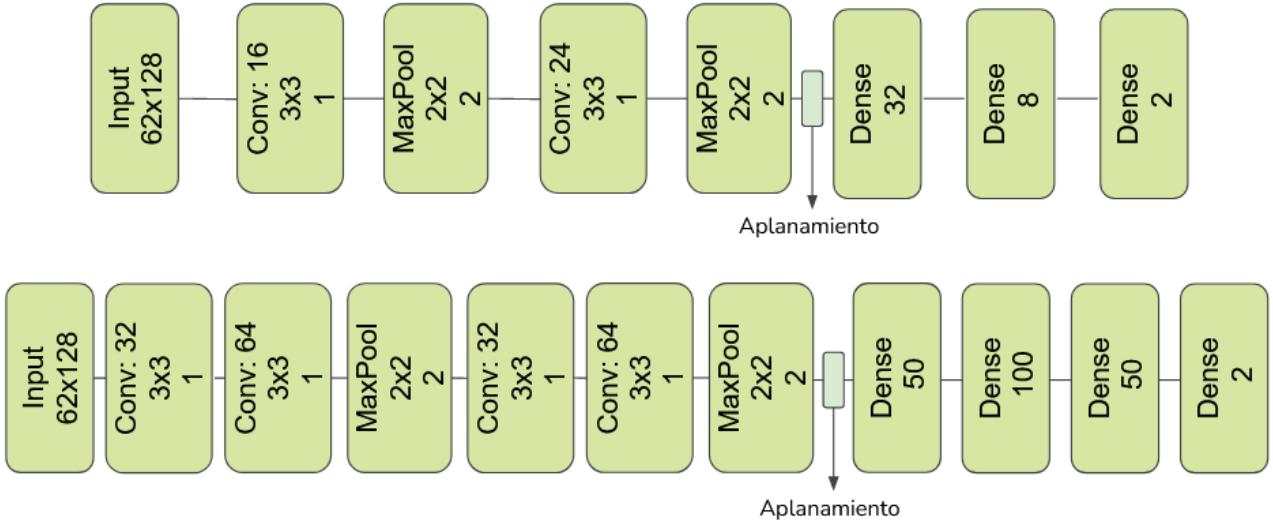
A continuación, vamos a mostrar como funcionan las distintas arquitecturas que hemos empleado.

## LeNet

LeNet es una de las primeras propuestas de redes convolucionales [61] que consiste únicamente en aplicar varias veces el bloque convolucional que se mostró en la Sección 2.2.1. Una vez se extraen las características se realiza la clasificación haciendo uso de un MLP.

Con estas redes, nuestro objetivo es construir modelos simples para establecer un punto de partida que tratar de mejorar añadiendo complejidad o ajustando los hiperparámetros. Hemos diseñado dos modelos basados en esta arquitectura, uno simple y otro más complejo, donde se aumenta tanto la anchura como la profundidad del modelo simple. Para hacer su diseño, hemos utilizado la representación más simple planteada y probado a modificar el número de filtros por nivel, el número de capas, así como el clasificador.

Las pruebas realizadas serán descritas en la Sección 6.2, pero mencionar que la construcción de la red nos ha dado muchos problemas, puesto que en muchos casos las redes no superan el 50% de acierto y pequeños cambios en la estructura de la red hacen que modelo pase de aprender a no aprender. En la Fig. 5.7 se muestran las arquitecturas de los modelos finales que hemos empleado. Tenemos el modelo simple que consiste en dos bloques convolucionales con pocos filtros y un MLP con dos capas ocultas y pocas neuronas. El modelo más complejo es similar al simple, pero hemos modificado el bloque convolucional de manera que se aplican dos capas convolucionales consecutivas sin aplicar la capa de reducción. Además, hemos aumentado la anchura y profundidad de la red, tanto en la extracción de características como en el clasificador.



**Figura 5.7:** Arquitectura basada en LeNet usada para la clasificación de las imágenes dadas. En las capas de convolución se muestran el número de filtros, el tamaño y desplazamiento de los filtros. En las capas de reducción se muestra el tamaño y desplazamiento, y en las capas densas el número de neuronas.

## EfficientNet

EfficientNet es una red cuyo objetivo es que sea eficiente y escalable, y que ha mostrado dar muy buenos resultados [62]. Uno de los problemas fundamentales que se observó tras las primeras CNN es que se tenían problemas para realizar el entrenamiento al aumentar la complejidad de los modelos [63]. Con el paso del tiempo, se fueron introduciendo mecanismos para lidiar con este problema y EfficientNet establece una familia con 8 modelos que parten de un modelo base y van aumentando su complejidad.

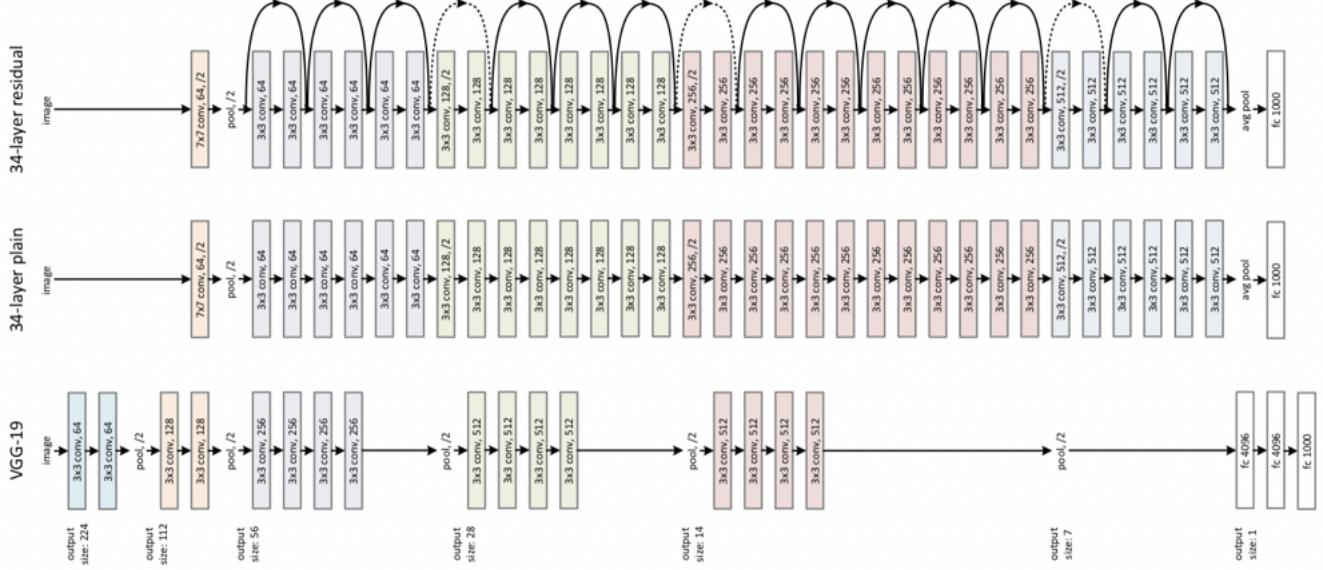
Una red neuronal se puede escalar aumentando la profundidad, la anchura o la resolución (tamaño de las imágenes), EfficientNet lo que hace es introducir un método de escalado que aumenta la complejidad aumentando las tres opciones simultáneamente. Parten de la premisa que las distintas formas de escalar están relacionadas, por ejemplo, si tenemos imágenes más grandes vamos a necesitar más filtros y más capas para conseguir realizar la clasificación [62].

Nosotros vamos a usar estas arquitecturas no solo porque han mostrado dar muy buenos resultados en conjuntos de datos complejos, sino porque, además, nos permite probar una misma arquitectura con distintos niveles de complejidad.

## ResNet

ResNet es una arquitectura que surge antes que EfficientNet, que introduce también un mecanismo para solucionar problemas a la hora de realizar la optimización cuando se tienen redes muy profundas [63]. Está basada en VGG [64] que se basa en aplicar sucesivamente bloques convolucionales, creando un modelo muy profundo. ResNet introduce lo que denominan celdas residuales que consisten en agregar la entrada del bloque convolucional a la salida del bloque. A esto le denominan aprendizaje residual y muestran mejoras en la optimización del modelo [63].

La arquitectura consiste en aplicar sucesivamente bloques convolucionales y solo se aplica la reducción tras la primera capa convolucional. A esto se le añade el aprendizaje residual. Por otro lado, al igual que para EfficientNet, se tiene una familia de modelos donde varían la profundidad de los modelos. Nosotros usaremos únicamente ResNet 50, donde el 50 hace referencia al número de capas. En la Fig. 5.8 podemos ver la arquitectura de este modelo comparada con la de VGG y la arquitectura de ResNet sin el aprendizaje residual.



**Figura 5.8:** Arquitectura de la red ResNet donde se muestra VGG y cómo se modifica para introducir el aprendizaje residual [63].

## Visual Transformer

Los transformers [3] son modelos que se basan únicamente en mecanismos de atención y, actualmente, se han convertido en el estado del arte al trabajar con texto. Los Visual transformers hacen uso de esta arquitectura y la aplican a imágenes. La idea fundamental es que dividen la imagen en bloques y tratan cada bloque como una palabra que el transformers procesa [46].

### 5.3. Métricas

A la hora de realizar un problema de ciencia de datos, es necesario establecer una métrica que nos diga cómo de bueno es nuestro modelo. Como ya se mencionó, en las redes neuronales se define una función pérdida, cuyo gradiente en función de los pesos de la red nos permite modificar estos pesos para mejorar los resultados.

Sin embargo, una vez realizado el ajuste, hay distintas métricas que nos permiten evaluar el rendimiento de los modelos, como puede ser el porcentaje de acierto o la curva ROC. Estas métricas nos dan información sobre distintos aspectos, pero en física, para problemas como el que estamos tratando, se suele usar la eficiencia y la pureza.

En un problema de clasificación binaria, se tienen dos clases que se suelen denominar como clase positiva y clase negativa. Con el clasificador, se asigna a cada instancia un valor de la clase y pueden darse 4 situaciones:

- La instancia sea positiva y el clasificador le asigne la clase positiva. A esto se le denomina positivo real (TP).
- La instancia sea negativa y el clasificador le asigne la clase negativa. A esto se le denomina negativo real (TN).
- La instancia sea positiva y el clasificador le asigne la clase negativa. A esto se le denomina falso negativo (FN).
- La instancia sea negativa y el clasificador le asigne la clase positiva. A esto se le denomina falso positivo (FP).

La eficiencia se define como:

$$\text{eff} = \frac{\text{TP}}{\text{FN} + \text{TP}}, \quad (5.2)$$

es decir, el cociente entre el número de instancias positivas que hemos clasificado bien entre el número de instancias que son realmente positivas.

La pureza se define como:

$$\text{pureza} = \frac{\text{TP}}{\text{FP} + \text{TP}}, \quad (5.3)$$

es decir, el cociente entre las instancias positivas correctamente clasificadas entre las instancias que hemos dicho que son positivas.

De todos los eventos positivos que se dan en el detector, el porcentaje que hemos sido capaces de clasificar correctamente es la eficiencia. Sin embargo, la pureza nos dice qué porcentaje de los eventos que hemos dicho que son positivos son realmente positivos.

En física, dependiendo de la tarea que estemos tratando, nos va a interesar maximizar una u otra. Por ejemplo, en los experimentos donde se pretende descubrir una partícula, se prefiere tener una pureza alta, de manera que si se dice que se tiene dicha partícula realmente, es muy probable que esa afirmación sea cierta. La eficiencia se busca cuando no se quiere perder eventos que realmente sean positivos. Por ejemplo, en un detector Geiger, que determina las partículas radiactivas, nos interesa que tengamos una alta eficiencia para saber cuándo es peligroso un sitio. También es interesante esta última medida cuando la producción de los eventos es cara y descartar eventos tiene un gran coste.

## 5.4. Hiperparámetros

Por último, vamos a recoger la mayoría de parámetros que tenemos a la hora realizar el flujo de trabajo completo, denominados hiperparámetros. Vamos a clasificar los hiperparámetros en función de la parte del flujo en la que influyen: El tratamiento de los datos, la arquitectura de la red o el entrenamiento de la red.

Antes de entrar en estos bloques, tenemos algunos parámetros generales, como puede ser la semilla aleatoria, que nos permite tener una reproducibilidad de los resultados<sup>3</sup>. Por otro lado, es importante tener en cuenta que el entrenamiento de una red neuronal es algo costoso, tanto en recursos como en tiempo, por lo que no nos podemos permitir hacer búsquedas en rejillas de gran tamaño. Por esto mismo, es fundamental seleccionar aquellos hiperparámetros que más pueden influir en los resultados.

## Tratamiento de los datos

A la hora de hacer la creación de las imágenes, hay un conjunto de parámetros que tenemos que establecer y que, en algunos casos, pueden afectar mucho al resultado obtenido. Los parámetros que tenemos son los siguientes:

- **cube\_shape\_x**: Fija el valor de  $c$  en el eje  $x$ . A partir de este obtenemos los valores en el eje  $y$  y  $z$ .
- **win\_shape**: Tamaño de la ventana cogida del detector.
- **projection**: Tipo de proyección empleada. Como opciones tenemos: las imágenes tridimensionales, la proyección en cualquiera de los ejes o el tratamiento de color.
- **projection\_pool** y **cube\_pool**: Tratamiento de las situaciones en las tengamos superposición de puntos al realizar el tratamiento. Podemos elegir el máximo o la media.
- **log\_trans**: Si hacemos o no una transformación logarítmica sobre la carga de los *hits*.
- **transform**: Si queremos aplicar una transformación sobre la imagen final. Consiste en hacer uso de la librería *torchvision* que introduce un conjunto de transformaciones como puede ser normalizar los datos.

---

<sup>3</sup>Hay algunas funciones que al usar la GPU no son completamente reproducibles, sin embargo, nuestros modelos no hacen usos de estas funciones.

## Arquitectura

Por otro lado, se tienen distintos parámetros que nos permiten controlar la red que estamos usando para realizar la clasificación. Estos parámetros dependen de si el modelo está ya entrenado, donde solo podemos controlar el clasificador y el número de capas a reentrenar, o si es un modelo que se construye de cero, donde con los hiperparámetros podemos definir tanto la estructura del extractor como la del clasificador. Los parámetros que tenemos para este apartado son:

- `model_name`: Nombre del modelo que vamos a usar.
- `conv_filters`, `conv_sizes`, `conv_strides`: Tupla que definen el número de capas convolucionales y como son estas capas: número de filtros, tamaño de los filtros y desplazamiento de los filtros.
- `pool_sizes` y `pool_strides`: Define el tamaño y el desplazamiento al realizar el *pool*. Son tuplas y tienen que tener el mismo tamaño que los parámetros que definen la convolución.
- `clf_neurons`: Tupla con el número de neuronas por cada capa. Cada elemento de la tupla establece una capa.
- `clf_no_linear_fun` y `conv_no_linear_fun`: funciones no lineales aplicadas tras las capas densas y las capas de convolución respectivamente.
- `bn`: si aplicamos capas de normalización por *batch*
- `dropout`: Porcentaje de *dropout*.

## Entrenamiento de la red

Por último, tenemos una serie de parámetros que nos permiten regular como se realizar el entrenamiento como puede ser el número de épocas o el tipo de optimizador:

- `batch_size`: Tamaño del *batch*.

- `n_epochs`: número de épocas.
- `optim_name`: tipo de optimizador.
- `optim_lr`: ritmo de aprendizaje.
- `scheduler_name`: En caso de querer tener un ritmo de aprendizaje que cambie con las épocas cual queremos aplicar.
- `scheduler_step_size`: cada cuanto se modifica el ritmo de aprendizaje.
- `scheduler_gamma`: cuanto se modifica el ritmo de aprendizaje.

Podemos ver que tenemos una gran cantidad de parámetros que podemos modificar y, en muchos casos, pequeñas modificaciones hacen que pasemos de tener modelos que no pasen del 50 % de acierto a modelos que lleguen al 80 %. Por estos motivos, el ajuste de estos hiperparámetros es uno de los aspectos más importantes para obtener un modelo que dé buenos resultados. Para esto, trabajaremos con una metodología concreta que nos permita llegar a ciertas conclusiones y luego ajustar solo los parámetros más relevantes.

## 5.5. Entorno de trabajo

Por último, vamos a describir algunas peculiaridades del entorno donde hemos realizado los experimentos, que es un aspecto importante al trabajar con redes neuronales, debido a la gran cantidad de recursos que necesitan. Por un lado, vamos a trabajar con redes neuronales y su entrenamiento se beneficia de la gran paralelización de las GPU, sin embargo, el entrenamiento consume muchos recursos, haciendo que en algunos casos un ordenador personal no sea suficiente. La pruebas iniciales se han realizado en un ordenador personal, pero para las pruebas finales hemos recurrido a un cluster de ordenadores que cuenta con 2 procesadores Xeon Silver 4110 (con un total de 32 hebras de procesamiento), 132GB de memoria RAM y 2 GPUs NVidia GeForce RTX 2080 Super. Inicialmente empezamos usando únicamente un portátil<sup>4</sup>, pero fue necesario hacer uso de un máquina más potente debido al volumen de pruebas, así como por la cantidad de memoria VRAM necesitada.

---

<sup>4</sup>El portátil cuenta con un procesador Intel i7-7700HQ, 16GB de memoria RAM y una GPU NVidia GTX 1060.

En el Anexo B se muestran tanto el la planificación que se ha seguido para resolver este trabajo como el presupuesto necesario para llevarlo a cabo.

Otro aspecto fundamental ha sido el control de todos los resultados obtenidos, ya que se han realizado muchas pruebas y ha sido necesario mantener un seguimiento, no solo por conocer los resultados de pruebas antiguas, sino para poder recuperar los modelos que han mostrado buenos resultados y saber los hiperparámetros de estos modelos. Como ya hemos mencionado, vamos hemos utilizado MLFlow para hacer este seguimiento, así como herramienta para hacer el despliegue automático de modelos. Además, nos ha permitido establecer un repositorio central donde poder recoger las métricas y modelos obtenidos en las distintas máquinas usadas<sup>5</sup>.

---

<sup>5</sup>Para usar el cluster fue necesario hacer una migración de la base de datos que se tenía en local a la nueva base de datos que se creó en el cluster. Esto ha supuesto un reto a nivel técnico, pues fue necesario modificar toda la base de datos porque esta guardaba información de las rutas de los modelos. Por otro lado, también ha habido que configurar MLFlow correctamente para que el repositorio funcionara correctamente.

# Capítulo 6

## Resultados y discusión

En este capítulo, vamos a mostrar los resultados de hacer uso de los distintos modelos que se han presentado junto con las distintas representaciones. En primer lugar vamos a mostrar la metodología seguida, así como una descripción de las pruebas realizadas. A continuación, se mostrarán todas las pruebas junto con sus resultados y comentarios sobre estos.

### 6.1. Metodología de experimentación

Al final del capítulo anterior se mostró la gran cantidad de hiperparámetros que tenemos, por lo que hacer una búsqueda por todos los parámetros es completamente prohibitiva. Teniendo esto en mente, vamos a definir una metodología de experimentación que nos permita discernir qué representación y resolución de las que se plantean es mejor. Con las mejores representaciones y resoluciones, haremos distintas pruebas sobre los modelos que nos permitan mejorar los resultados.

En primer lugar, hemos realizado un conjunto de prueba no estructuradas que nos han permitido entender el problema y comprobar que el tratamiento diseñado es correcto. Asimismo, usamos una de las representaciones para diseñar modelos que funcionen para usarlos en las pruebas futuras.

El siguiente punto será determinar cuál de las representaciones que presentamos es mejor. Para esto, probamos las distintas representaciones presentadas para los distintos modelos. En concreto, haremos uso de un modelo simple y uno complejo diseñado por nosotros, distintos modelos EfficientNet, ResNet y GoogleNet. Las pruebas las haremos de manera que se compartan todos los parámetros y que lo único que vamos a modificar son los parámetros del aprendizaje, como el ritmo de aprendizaje, en caso de ser necesario. Una vez hecho este análisis, se pretende obtener una representación que sea mejor que el resto que usaremos para el resto de pruebas.

El siguiente paso consiste en escoger una resolución  $c_x$ , para ello repetiremos el estudio anterior, pero cambiando la resolución. Finalmente, ajustamos los hiperparámetros para intentar superar los resultados obtenidos. Para esto se define una rejilla sobre la que se buscan qué parámetros funcionan mejor. Aplicamos esta búsqueda para los distintos modelos.

Una vez hemos obtenido un buen modelo, hemos tratado de mejorar los resultados introduciendo modelos más complejos como pueden ser concatenación de redes (en la Fig. 6.9 se muestra el esquema de la arquitectura que mencionamos), o ensembles que nos permitan mejorar los resultados.

## 6.2. Pruebas iniciales

Tras obtener las representaciones y las imágenes, vamos a realizar unos primeros estudios para ver que el procesamiento realizado es correcto y tratar de establecer redes capaces de realizar la clasificación. A continuación, se muestran los experimentos preliminares que han permitido delimitar las pruebas futuras, así como afianzar algunas hipótesis de partida para, posteriormente, hacer su comprobación de manera exhaustiva. Con estas pruebas, queremos estudiar tanto los datos como los modelos para poder establecer una línea de trabajo.

Como punto de partida se ha usado la representación más simple y una resolución y tamaño que permitan recoger siempre el comienzo de las cascadas, al ser la zona donde más diferencias se encuentran. Además, hemos tratado de utilizar imágenes pequeñas que nos permitieran construir un modelo con pocos parámetros y establecer un modelo base.

Por todo esto, para estas pruebas iniciales, en la mayoría de los casos, hemos usado imágenes donde hemos proyectado el eje  $y$  y usado una resolución de  $c_x = 1000$  px y un de tamaño de ventana de  $62 \times 128$  px (62 píxeles para el eje  $x$  y 128 para el  $z$ )

En primer lugar, hemos diseñado el modelo LeNet simple que se mostró en la Sección 5.2. Establecimos un modelo que no fuera especialmente complejo y con pocos filtros, ya que las diferencias fundamentales entre las cascadas son patrones simples como líneas. Además, un modelo simple consume menos recursos y el aprendizaje es más rápido, permitiéndonos hacer un mayor número de pruebas. En definitiva, queremos establecer un modelo base capaz de extraer la máxima información del origen de la cascada.

El primer resultado que se obtuvo fue de 67 % de acierto en validación, donde la proyección se hizo sobre el eje  $z$ . Sin embargo, el máximo se obtuvo con un 88.5 %, tras realizar cambios en la arquitectura, en cómo se creaban las imágenes y en algunos parámetros de interés, así como el eje proyectado escogido, observando que el eje  $y$  es el que mejor resultados obtiene. Entrar a comentar todas estas pruebas realizadas no es especialmente interesante al ser más de 100 pruebas, donde muchos de los cambios consisten en realizar ajustes en el tratamiento de las imágenes, tamaño de las imágenes y cambios en la arquitectura de los modelos LeNet y del ciclo de aprendizaje.

Sin embargo, sí hay algunas cosas que se han observado que son especialmente interesantes de comentar. Una de las primeras cosas que tratamos de hacer, tras obtener un modelo que mostraba buenos resultados, fue tratar de escalarlo aumentando tanto el número de filtros como el número de capas. No obstante, nos encontramos con que todos los modelos tenían mucha sensibilidad con la semilla de números aleatorios y con la arquitectura de la red, haciendo que en algunos casos se obtuviesen buenos resultados (en torno al 80 % de acierto) mientras que en otros casos no se conseguía superar el 50 %, que es lo mismo que usar una moneda para realizar la clasificación.

El hecho que los resultados dependan de la semilla es un problema, puesto que nos incapacita para decir que un conjunto de parámetros es malo, ya que podría ser que simplemente estemos en una semilla mala. Para solucionar este problema, probamos a modificar el tamaño del *batch* ( $bs$ ) y cada cuánto se modifica el ritmo de aprendizaje ( $lr$ ). También probamos a modificar el optimizador (*optim.*), aunque vimos que Adam mostraba mejores resultados siempre. El motivo por el que pensábamos que tendría que ser un parámetro relacionado con el aprendizaje, se debe a que el tratamiento de las imágenes no depende

de la semilla, por lo que al cambiar la semilla lo que modificamos es: el orden con el que llegan las imágenes al modelo, el conjunto de validación y cómo se realiza el descenso del gradiente. Con distintas pruebas observamos que al aumentar tanto el *bs* como la frecuencia de cambio del *lr* se solucionaba este problema.

Otra gran cantidad de pruebas fueron hechas con la idea de obtener modelos basados en LeNet, pero más complejos para ver si éramos capaces de mejorar los resultados al obtener patrones más complejos que el modelo simple no fuera capaz de determinar. Sin embargo, nos encontramos con que muchos de los modelos probados no conseguían clasificar el conjunto de datos. Debido a esto, solo hemos conseguido establecer dos modelos de distinta complejidad que consigan clasificar los datos. Las arquitecturas de estos modelos se mostraban en la Fig. 5.7. A la hora de realizar los estudios posteriores, vamos a usar estos modelos, así como los modelos EfficientNet que tienen distintos niveles de complejidad para ver si modelos más complejos mejoran o empeoran los resultados.

BN	log	Normalización	dropout	Acierto Mejor Val	Acierto Entrenamiento
No	No	No	0.0	0.885	0.892
No	No	Sí	0.0	0.649	0.638
No	Sí	No	0.2	0.865	0.840
No	Sí	Sí	0.2	0.879	0.888
Sí	No	No	0.35	0.884	0.875
Sí	No	Sí	0.35	0.638	0.87
Sí	Sí	No	0.35	0.863	0.879
Sí	Sí	Sí	0.35	0.866	0.865

**Tabla 6.1:** Estudio del efecto de la normalización, capas de normalización por *batch* (BN) y la transformación logarítmica sobre una misma representación. La representación empleada es la proyección en *y* y el modelo usado es el LeNet simple. El *dropout* se ha introducido en aquellos casos en los que en los que observaba mucho sobreajuste.

Antes de entrar a realizar los distintos estudios, se probó a aplicar la transformación logarítmica, así como a introducir capas de normalización por *batch* (BN) [65]. La introducción de estas modificaciones en la red no mostraron mejoras en los resultados y se observó un aumento en el sobreajuste llegando a obtener cerca de un 100 % de acierto en el conjunto de entrenamiento. Con el uso del *dropout*, se consiguió reducir el sobreajuste llegando a obtener resultados prácticamente idénticos a los obtenidos sin introducir estas modificaciones.

En la Tabla 6.1 se muestra el estudio realizado sobre la proyección simple con una resolución de 1000 px y una ventana de  $62 \times 128$  px. En este estudio, hemos probado todas las combinaciones de introducir o no las capas de BN, la transformación logarítmica y realizar una normalización sobre los píxeles<sup>1</sup>. También hemos modificado el *dropout* en aquellos casos donde se tenía sobreajuste, más detalles sobre los hiperparámetros se muestran en el Anexo A.1. Los mejores resultados se han obtenido cuando no se ha introducido ninguna de las modificaciones mencionadas. Se han obtenido resultados similares introduciendo la capa de BN. Sin embargo, para obtener estos resultados, es necesario introducir mucho *dropout* debido al sobreajuste. Por lo tanto, se observa que estas técnicas no suponen un mejorar en los resultados, por lo que no vamos a aplicarlas.

### 6.3. Selección de una representación

Las pruebas iniciales nos han servido para entender mejor el conjunto de datos y establecer un punto de partida. Además, la sección anterior nos ha mostrado la cantidad de pruebas que se pueden hacer para mejorar los resultados, poniendo de manifiesto la necesidad de establecer una metodología que nos permita descartar opciones.

Lo primero que vamos a hacer es establecer qué representación da mejores resultados de manera general. Para ello, probamos distintos modelos, con distintas complejidades, y determinaremos cuál de ellas es mejor. Para todas las pruebas, hemos usado los mismos parámetros y redes, salvo el ritmo de aprendizaje que se ha modificado solo si el modelo no conseguía superar el 50 %. Las pruebas se han realizado a 10 épocas y con una resolución  $c_x = 1000$  px.

En la Tabla 6.2 se recogen los resultados para las tres representaciones que hemos presentado, donde para la representación tridimensional no se han realizado pruebas con los modelos preentrenados al no tenerlos accesibles. Por otro lado, para todas las pruebas con modelos preentrenados, hemos usado el mismo clasificador que para el modelo LeNet simple. En el Anexo A.2 se pueden consultar en detalle los hiperparámetros empleados.

---

<sup>1</sup>Esta normalización se suele aplicar a las imágenes y consiste en aplicar una transformación z-score sobre los distintos canales una vez la imagen está en el rango 0 y 1. Para aplicar esta transformación, se toman como media: (0.485, 0.456, 0.406) y como desviación estándar: (0.229, 0.224, 0.225). En nuestro caso, las imágenes están en este rango, por lo que solo aplicamos la transformación descrita.

% de acierto	Proyección			Color			3D		
	train	val.	mejor val.	train	val.	mejor val.	train	val.	mejor val.
Simple	0.892	0.885	0.885	0.763	0.772	0.772	0.637	0.627	0.627
Complejo	0.852	0.866	0.866	0.856	0.836	0.852	—	—	—
EfficientNet B0	0.762	0.756	0.768	0.686	0.641	0.646	—	—	—
EfficientNet B2	0.688	0.665	0.665	0.675	0.601	0.610	—	—	—
EfficientNet B5	0.671	0.613	0.623	0.659	0.587	0.597	—	—	—
EfficientNet B7	0.663	0.638	0.643	0.681	0.601	0.601	—	—	—
ResNet 50	0.972	0.690	0.731	0.953	0.607	0.651	—	—	—
GoogleNet	0.788	0.779	0.779	0.706	0.659	0.667	—	—	—

**Tabla 6.2:** Porcentajes de acierto para el conjunto de entrenamiento y validación en la última época y el mejor porcentaje de acierto obtenido en validación. Se muestran los resultados obtenidos para distintas representaciones y modelos.

En la tabla se muestran tanto el mejor porcentaje de acierto en validación (que corresponde al modelo guardado), como los porcentajes en entrenamiento y en validación obtenidos en la última época. Lo más importante que notamos es que, en todos los modelos, los mejores resultados se tienen cuando nos quedamos únicamente con la proyección del eje  $y$ . Por lo tanto, para el resto de estudios vamos a hacer uso de la proyección simple en este eje. No obstante, es posible que estas diferencias se deban a que no hemos conseguido un modelo capaz de aprender la complejidad añadida al introducir el color. Sin embargo, al no tener ninguna evidencia de esto, no vamos a seguir realizando más pruebas con la proyección de color.

Antes de pasar al resto de pruebas, veamos en detalle los resultados que se recogen en esta tabla. Por un lado, mencionar que el estudio está sesgado con los modelos LeNet, pues estos modelos se han obtenido haciendo uso de la proyección simple, por lo que es posible que se tengan mejores resultados únicamente por esto. Sin embargo, si nos fijamos únicamente en los modelos preentrenados vemos que la proyección simple es mejor en todos los casos.

En la representación tridimensional, vemos cómo solo hemos hecho pruebas con el modelo LeNet simple, puel el complejo superaba la memoria de la GPU y no se tiene versión tridimensional de los modelos preentrenados. LeNet si lo podemos usar al haber sido diseñado por nosotros y ser muy simple. Con esta representación, vemos que el resultado obtenidos es considerablemente inferior a los obtenidos con las otras representaciones. Aunque con esta metodología hemos obtenido peores resultados a los obtenidos con las otras

representaciones, cuando se hicieron unas pruebas iniciales con este modelo se obtuvo un porcentaje de acierto superior al 70 %. Por lo tanto, es posible que si se probara a modificar parámetros como el  $lr$ , se podrían mejorar los resultados. Sin embargo, no se han realizado más pruebas debido a la cantidad de recursos que consume, haciendo que el aprendizaje sea muy lento. Por todo esto, consideramos que esta representación no es especialmente interesante de abordar, aunque sí es interesante pensar formas alternativas para abordar el problema. Una opción es utilizar matrices dispersas (*sparse matrix*), que no guardan información de los espacios vacíos. En la literatura se han abordado estos problemas de distintas formas: En [66] se han probado a hacer uso de las arquitecturas del estado del arte y extenderlas a imágenes 3D de manera eficiente, en [67] se hace uso de las matrices dispersas en redes convolucionales y en [68] se aplican estas matrices para la detección de los distintos eventos que tienen lugar dentro del detector que estamos usando nosotros.

Aunque, vemos que hay muchos estudios que abordan los problemas que hemos observado con las imágenes tridimensionales, no vamos a entrar en ellos y lo dejamos como un posible área de estudio futuro.

Por último, podemos ver como en los modelos EfficientNet, al aumentar la complejidad, los resultados empeoran. Esto puede deberse simplemente a que a mayor complejidad peores resultados, pero también puede ser que, al estar usando en los 4 casos el mismo clasificador, los patrones más complejos no se estén aprendiendo. También puede deberse a que hemos fijado a 10 el número de capas a reentrenar y puede que al tener modelos más complejos sea necesario reentrenar un número mayor de capas. Para los otros dos modelos, observamos que se tienen mejores resultados, aunque para ResNet tenemos mucho sobreajuste. Por todo esto, es interesante probar distintas configuraciones de hiperparámetros con idea de mejorar los resultados o reducir el sobreajuste.

## 6.4. Selección de la resolución

Antes de entrar en los ajustes de los modelos, vamos a repetir el estudio anterior, pero variando la resolución en vez de la representación. Cuando se hace referencia a la resolución, estamos haciendo mención al número de píxeles con el que se describe el eje  $x$  del detector. Las resoluciones para los otros ejes están relacionadas por la ecuación 3.1. Al igual que

para el estudio anterior, mostramos los resultados obtenidos en la Tabla 6.3 y en el Anexo A.3 se pueden consultar en detalle los hiperparámetros de las distintas pruebas realizadas.

Por un lado, se observa que para los modelos EfficientNet si aumentamos la resolución se tiene una mejora de los resultados, obteniendo los mejores resultados con el modelo B0, que es el más simple. Sin embargo, por el mismo motivo que se comentó en la discusión anterior, no podemos descartar que los modelos más complejos presenten mejores resultados al modificar los hiperparámetros, ya que estamos escalando la complejidad del modelo sin modificar el resto de elementos, por ejemplo el clasificador. Lo que sí parece muy claro es que, si aumentamos la resolución, obtenemos mejores resultados. Esto es muy interesante porque al aumentarla, tenemos una mayor definición en la trayectoria, pero a cambio se reduce la cantidad de cascada que hay en la imagen, puesto que se mantiene el tamaño de la imagen. De esta forma, al aumentar la resolución, nos centramos más en el origen de la cascada, ignorando la parte final.

Por otro lado, para LeNet se observa que una vez se supera la resolución de 1000 px, los resultados son similares. Vemos que para resoluciones menores tenemos resultados ligeramente inferiores y que para el modelo simple con la máxima resolución se observa que el modelo no es capaz de superar el 50 %. Esto se puede deber a los hiperparámetros del modelo, aunque hemos hecho un par de pruebas modificándolos, pero no se han mejorado. Otra opción posible es que hemos aumentado tanto las imágenes, que se pierde mucha cascada como para que el modelo sea capaz de aprender. Sin embargo, viendo que los resultados del modelo simple y complejo son similares para el resto de resoluciones, y que para esta resolución el modelo complejo sí obtiene buenos resultados, es posible que la red no aprenda debido a la primera cuestión mencionada.

Es importante notar que la información que se muestra en los modelos LeNet no es comparable a la información dada a los modelos preentrenados para una misma resolución. Esto se debe a que el tamaño de las imágenes es distinto. En la Fig. 4.3 se mostraba el mismo evento para distintas resoluciones, viendo que, a medida que se aumentaba la resolución, se perdía también información. Estas imágenes tenían un tamaño de  $128 \times 128$  px y nosotros estábamos usando imágenes  $62 \times 128$  px y  $224 \times 224$  px para los modelos LeNet y preentrenados, respectivamente. Esto quiere decir que, si tenemos una resolución de 1000 px y un tamaño  $112 \times 112$  px, la cantidad de cascada que se recoge es más o menos la

misma que se recoge en una imagen con resolución 2000 px pero de tamaño  $224 \times 224$  px<sup>2</sup>. Por ende, no debemos de pensar que, a una misma resolución, la información que están recibiendo los modelos es la misma, pues depende también de la ventana.

A modo de resumen, vemos que para los modelos LeNet observamos cómo el resultado satura en una resolución de 1000 px. Observamos que los resultados con resoluciones de 1000 px y de 1500 px son similares, siendo algo mejores para la resolución de 1000 px. No obstante, para el resto de pruebas se ha usado la resolución de 1500 px debido a que, inicialmente, se observan mejores resultados con esta resolución, pero fue tras solucionar un problema en los experimentos, cuando se observó que la resolución de 1000 px era mejor. Este fallo se detectó después de realizar las pruebas de la siguiente sección, por lo que habría que repetir todas las pruebas, que consisten en entrenar 50 modelos. Por este motivo y debido a que las diferencias que se observan entre las resoluciones son muy pequeñas, vamos a quedarnos con la resolución de 1500 px. Además, tener una mayor resolución nos permite definir mejor el origen de la cascada, que es la región que se quería describir con estos modelos LeNet.

Para los modelos EfficientNet, no se observa que alcancemos un máximo en el rendimiento del modelo al aumentar la resolución, obteniendo los mejores resultados con la última probada, 3000 px. No vamos a probar resoluciones mayores para mantener la máxima información posible, ya que con estos modelos no pretendemos centrarnos únicamente en el origen de la cascada.

Resolución (px)	250	500	1000	1500	2000	2250	2500	3000
Simple	0.834	0.855	0.885	0.881	0.502	—	—	—
Complejo	0.516	0.838	0.879	0.872	0.8864	—	—	—
EfficientNet B0	0.611	0.655	0.768	0.782	0.803	0.821	0.826	0.842
EfficientNet B2	0.599	0.592	0.665	0.683	0.717	0.720	0.746	0.760
EfficientNet B5	0.582	0.579	0.623	0.666	0.693	0.719	0.716	0.738

**Tabla 6.3:** Porcentaje de acierto para distintos modelos y distintas resoluciones (píxeles usados para describir el eje  $x$  del detector).

<sup>2</sup>No es exactamente la misma por el factor 1.25 que hay cuando se obtiene el  $c_z$  dado el  $c_x$  que es el parámetro de resolución del que hemos hablado en toda la discusión

## 6.5. Ajuste de hiperparámetros

Con todos los resultados observados, surgen algunos aspectos importantes. Vemos que los modelos LeNet obtienen resultados superiores usando imágenes más pequeñas ( $62 \times 128$ ) mientras que para los modelos EfficientNet se obtienen siempre peores resultados incluso cuando se usan imágenes de mayor resolución y tamaño. Sin embargo, cabría esperar que los modelos preentrenados obtuviesen mejores resultados que los modelos simples al tener mayor información, resolución y ser modelos más complejos. Por tanto, deberían de ser capaces de extraer patrones más complejos. Además, al haber sido entrenados con muchas imágenes, el entrenamiento debe ser más simple.

Por otro lado, los modelos LeNet se diseñaron con la idea de centrarse únicamente en el origen de la cascada al ser el punto donde teóricamente hay más diferencias. Observamos cómo se obtienen muy buenos resultados con una red muy simple. Teniendo todo esto en cuenta, es posible que, para los modelos LeNet, estemos muy cerca del máximo que se puede conseguir teniendo en cuenta que estamos usando una representación desprecia una dimensión completa y que se pierde mucha información al generar las imágenes debido al tamaño y resolución que estamos usando. Por lo mismo, los modelos más complejos deberían al menos igualar a LeNet.

Por todo esto, ahora vamos a ajustar los hiperparámetros de los distintos modelos, centrándonos en los modelos preentrenados donde si es esperable una gran mejora. Para los modelos LeNet lo que hemos hecho ha sido ajustar algunos hiperparámetros, pero no esperamos que se consigan mejoras significativas.

### LeNet

Como ya hemos puesto de manifiesto en varias ocasiones, tenemos una gran cantidad de parámetros y no nos podemos permitir hacer una búsqueda sobre todos ellos. Por todo esto, vamos a probar a modificar 4 parámetros: el ritmo de aprendizaje ( $lr$ ), lo que se modifica el ritmo de aprendizaje cada época ( $\gamma$ ), el tamaño de los filtros ( $ks$ ) y el desplazamiento ( $kd$ ). Las pruebas las vamos a realizar para ambos modelos y en las Tablas 6.4 y 6.5, se recogen los resultados. En el Anexo A.4 se muestran los hiperparámetros en detalle.

		ks	kd	ks	kd	ks	kd	ks	kd
		3	1	5	1	3	2	5	2
lr	$10^{-3}$	0.883	0.891	0.771	0.718				
$\gamma$	1.0								
lr	$10^{-3}$	0.884	0.892	0.768	0.727				
$\gamma$	0.8								
lr	$10^{-3}$	0.881	0.887	0.763	0.728				
$\gamma$	0.7								
lr	$10^{-3}$	0.875	0.886	0.762	0.718				
$\gamma$	0.5								
lr	$10^{-4}$	0.818	0.862	0.627	0.593				
$\gamma$	1.0								
lr	$10^{-4}$	0.758	0.836	0.597	0.573				
$\gamma$	0.8								
lr	$10^{-4}$	0.713	0.803	0.612	0.584				
$\gamma$	0.7								
lr	$10^{-4}$	0.657	0.764	0.609	0.578				
$\gamma$	0.5								

**Tabla 6.4:** Porcentajes de aciertos en validación de los modelos obtenidos en la búsqueda de rejilla realizada sobre el modelo LeNet simple. Se han modificado el ritmo de aprendizaje ( $lr$ ), la cantidad que se modifica el  $lr$  ( $\gamma$ ), el tamaño de los filtros ( $ks$ ) y su desplazamiento ( $kd$ ).

		ks	kd	ks	kd	ks	kd	ks	kd
		3	1	5	1	3	2	5	2
$\gamma$	1.0	0.875	0.857	0.516	0.858				
$\gamma$	0.8	0.879	0.857	0.516	0.526				
$\gamma$	0.5	0.870	0.767	0.516	0.484				

**Tabla 6.5:** Porcentajes de aciertos en validación de los modelos obtenidos en la búsqueda de rejilla realizada sobre el modelo LeNet complejo. Se han modificado la cantidad que se modifica el  $lr$  ( $\gamma$ ), el tamaño de los filtros ( $ks$ ) y su desplazamiento ( $kd$ ).

En la rejilla definida con estos parámetros, solo hemos probado un par de valores para cada parámetro. Aun así, el número de pruebas realizadas al final, ha sido en torno a 50, ya que se han realizado algunas más y descartado otras. En concreto, hemos hecho 32 pruebas para el modelo simple y 15 para el complejo y la rejilla consiste en:

- `optim_lr`  $\in [0.0001, 0.001]$ .
- `scheduler_gamma`  $\in [0.5, 0.7, 0.8, 1]$ .

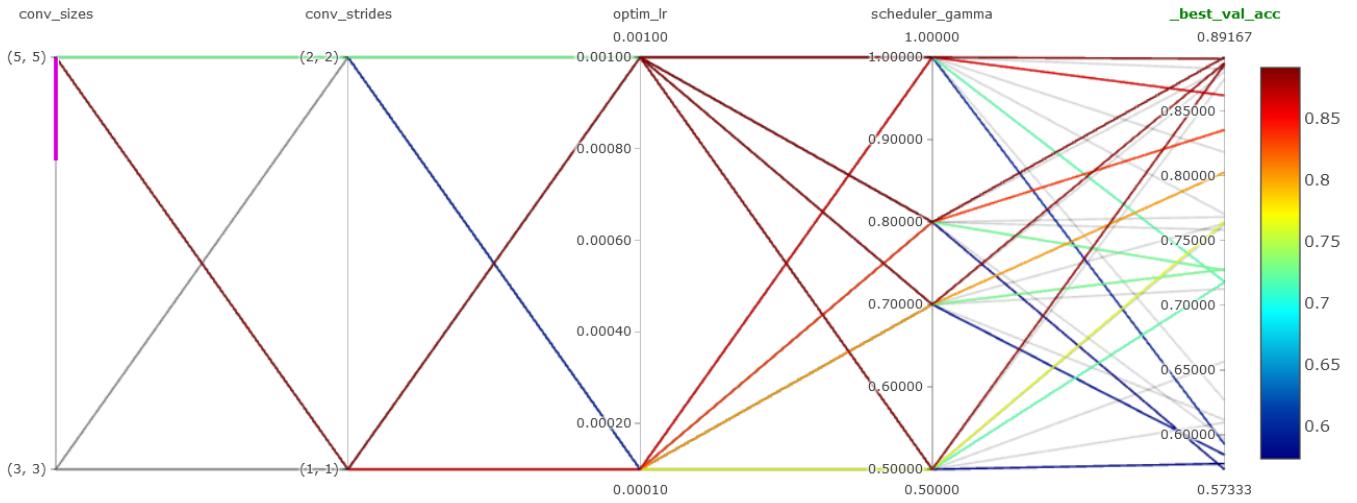
- `kernel_stride`  $\in [1, 2]$ .
- `kernel_size`  $\in [3, 5]$ .

Para estudiar los resultados hemos usado MLFlow que, además, nos permite comparar experimentos y hacer gráficas mostrando los distintos hiperparámetros y los resultados de esos experimentos. En la Fig. 6.1 se muestra un ejemplo de la gráfica obtenida y, además, se pueden seleccionar rangos de parámetros para visualizar correctamente modelos concretos. Las conclusiones que podemos llegar de los resultados del modelo simple son:

- Siempre se observa que los resultados son peores cuando se usa un desplazamiento de los filtros de 2 px frente a usar un  $kd = 1$  px.
- De igual manera, se observa con el  $lr$ , donde el  $lr = 0.0001$  siempre da peores resultados frente a usar  $lr = 0.001$ . El resto de conclusiones las vamos a limitar estos casos.
- También vemos que, en general, a medida que disminuimos el  $\gamma$  vemos que los resultados empeoran pero el máximo se encuentra en 0.8 siendo ligeramente peor cuando el gamma es 1.0.
- Siempre se observan mejores resultados con los filtros 5 px de tamaño.
- El mejor resultado se tiene con un 0.892 para  $lr = 0.001$ ,  $\gamma = 0.8$ ,  $ks = 5$  px y  $kd = 1$  px.

Para el modelo complejo, hemos usado un valor de  $\gamma$  menos y vimos que, cuando se tenía  $lr = 0.001$ , el modelo no aprendía, por lo que no se realizaron todas las pruebas con este  $lr$ . Además, hemos tenido que modificar las imágenes introduciendo lo que se denomina *padding* para que, al aplicar los filtros, se puedan utilizar los píxeles en los bordes y no se reduzca mucho la imagen. El método de *padding* empleado consiste en ampliar la imagen añadiendo píxeles negros fuera de la imagen. Con estas pruebas se observó que:

- Siempre se obtienen mejores resultados cuando el desplazamiento es de 1 px.
- Con respecto al  $\gamma$  vemos que los mejores resultados se tienen con  $\gamma = 0.8$  y parece que lo mejor es cuando es 0.8 ó 1.0.



**Figura 6.1:** Gráfica obtenida de MLFlow con los resultados de la búsqueda de rejilla aplicada sobre el modelo simple con una resolución de 1500 px.

- A diferencia que lo que se observaba con el modelo simple, se obtiene mejores resultados con el  $ks = 3$  px.
- El mejor resultado se tiene con un 0.879 para  $lr = 0.0001$ ,  $gamma = 0.7$ ,  $ks = 3$  px y  $kd = 1$  px.

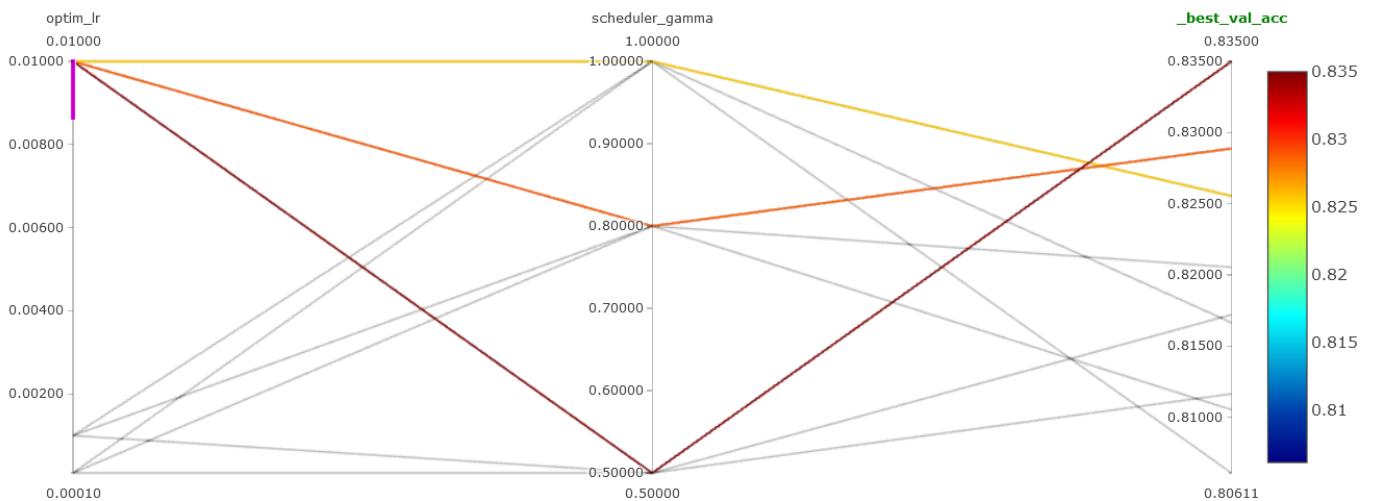
En resumen, vemos cómo el uso de filtros de mayor tamaño mejora ligeramente los resultados obtenidos con el modelo simple. Notar que la mejora obtenida es de un 1 % que es considerablemente bajo, teniendo en cuenta que hemos hecho más de 50 pruebas para llegar a este modelo.

Por otro lado, aunque es posible que se mejoren aún más los resultados si buscamos en otros parámetros, sería necesario realizar muchas más pruebas con el gran coste de energético y de tiempo que eso conlleva. Además, previamente a este trabajo, el grupo de investigación con el que se está realizado este trabajo, ha abordado la resolución del mismo problema que hemos presentado nosotros, usando otros algoritmos de ML. Para ello se han utilizando los datos tabulares codificados de manera que se mantiene toda la información de la cascada. Tras el ajuste que acabamos de presentar, nuestros resultados ya son similares a los obtenidos en este estudio utilizando modelos como Random Forest [56]. Esto nos muestra que nuestro modelo más simple ya obtiene muy buenos resultados

y cómo el origen de las cascadas ya nos permite obtener resultados satisfactorios desde el punto de vista físico.

## Modelos preentrenados

Una vez hemos ajustado los modelos LeNet, veamos los modelos preentrenados. Lo primero que vamos a hacer es tratar de establecer un valor de  $\gamma$  y  $lr$ . Como no podemos hacer esto para todos los modelos, hacemos primero una prueba inicial con un clasificador complejo, un extractor de complejidad intermedia, como EfficientNet B2, y usaremos un *dropout* de 0.2. El clasificador utilizado tiene una única capa oculta con 256 neuronas. Además, se han descongelado 50 capas y se han probado 3 valores de  $\gamma$ : [1.0, 0.8, 0.5] y 3 valores de  $lr$ : [0.0001, 0.001, 0.01]. Los mejores resultados se han obtenido con el  $\gamma$  más bajo y el  $lr$  más alto. Estos resultados se muestran en la Fig. 6.2 y se pueden consultar detenidamente en el Anexo A.6.



**Figura 6.2:** Gráfica obtenida de MLFlow con los resultados de la búsqueda inicial del  $lr$  y de  $\gamma$ .

Con estos valores de  $\gamma$  y  $lr$ , definimos una rejilla donde se modifica el clasificador y el número de capas descongeladas del extractor de características. Los clasificadores tienen una única capa oculta con 256 y 128 neuronas ( $n_n$ ) y el número de capas a descongelar ( $n_u$ ) probados son [5, 50, 100, 300, 1000]. Hemos fijado el *dropout* ( $dr$ ) a 0.3, ya que, en general, hemos visto que se tiene sobreajuste con estos modelos. Por otro lado, en algunas ocasiones

las pruebas de  $n_u = 1000$  no se han realizado por no tener memoria suficiente, pues los modelos que estamos usando llegan a tener muchas capas y pesos. Hemos introducido  $n_u = 1000$  como caso en el que se descongelan todas las capas.

Estas pruebas se hicieron en primer lugar usando únicamente los modelos EfficientNet B0, B2 y B5 para ver si de esta forma podríamos reducir el número de pruebas al observar algún patrón. En la Tabla 6.6 se recogen los resultados en validación de la búsqueda de rejilla sobre estos modelos, así como las pruebas realizadas utilizando ResNet y GoogleNet. En el Anexo A.7, se pueden consultar los detalles de los hiperparámetros.

	B0		B2		B5		ResNet	GoogleNet
	256	128	256	128	256	128	256	256
5	0.737	0.838	0.749	0.756	0.729	0.728	0.779	0.837
50	0.884	0.878	0.826	0.819	0.808	0.811	0.878	0.869
100	0.889	0.516	0.872	0.870	0.516	0.516	0.516	0.880
300	0.898	0.839	<b>0.913</b>	0.904	0.859*	0.859	0.569	0.516
1000	0.898	0.839	0.854	0.516	—	—	0.569	0.516

**Tabla 6.6:** Porcentajes de acierto en validación de los modelos obtenido en una búsqueda de rejilla donde se ha modificado el número de capas a reentrenar y el clasificador. Se muestran los resultados para distintos modelos preentrenados. La primera columna hace referencia al número de capas descongeladas  $n_u$ , la primera fila al modelo y la segunda al número de neuronas del clasificador  $n_n$ . El asterisco significa que se ha entrenado con 15 épocas.

En primer lugar, se observa es que los resultados son muy superiores a los obtenidos sin realizar la modificación de los parámetros, mostrando cómo de importante puede ser el ajuste. Por otro lado, el mejor resultado se obtiene con el modelo B2 con  $n_u = 300$  y con  $n_n = 256$  obteniendo un 91.3 % de acierto en validación que es una mejora considerable a los resultados obtenidos con los modelos LeNet.

Podemos sacar algunas conclusiones de estos resultados. Vemos que para los modelos EfficientNet, en general, se tienen mejores resultados cuando se usa  $n_n = 256$ , por lo que para los otros modelos se decidió usar solo este valor. Con respecto a  $n_u$ , observamos que los resultados mejoran al aumentar este parámetro hasta que se alcanza un máximo, a partir del cual los resultados empeoran. Una posible explicación es que, al ir aumentando el número de pesos que se pueden ajustar, se llegue a un punto donde haya tantos pesos por ajustar que la red tenga problemas para hacerlo, dando lugar a peores resultados.

Por otro lado, vemos que los resultados obtenidos con los modelos B5, ResNet y GoogleNet son ligeramente inferiores a los obtenidos con otros modelos, pero esto puede deberse únicamente al espacio de búsqueda establecido. Sobre estos modelos hemos hecho algunas pruebas adicionales para ver si se conseguían mejorar un poco más los resultados, especialmente para aquellos en los que se han obtenido malos resultados.

- ResNet 50: Hemos probado a usar  $dr = 0.4$  para  $n_u = 50, 100$  obteniendo 0.883, 0.891 respectivamente. Obteniendo una mejora considerable y similar a los mejores resultados obtenidos con el resto de modelos.
- GoogleNet: También hemos probado a usar el  $dr = 0.4$  para  $n_u = 100, 300$  obteniendo 0.892 y 0.516. Donde vemos que para  $n_u = 100$  hemos mejorado el porcentaje de acierto llegando también a igualar al resto de modelos. También hemos probado a usar  $n_n = 128$  obteniendo 0.808.
- B5: Hemos usado  $n_n = 512$  para el caso que peores resultados tenía obteniendo 0.819.
- B0: Hemos probado a introducir  $dr = 0.4$  para los mejores resultados pero se han empeorado los resultados.

A modo de conclusión de las pruebas que hemos realizado, vemos que en la mayoría de pruebas se supera el 0.89, de hecho, solo el B5 no lo ha conseguido. Por otro lado, vemos que el B0 y B2 son los que mejores resultados han dado, sobre todo el modelo B2 que supera en un 2-3 % al resto de modelos. En general, vemos que las diferencias entre los modelos no es muy alta, siendo posible que las diferencias se deban a los hiperparámetros, es decir, parece que todos los modelos son más o menos similares. Esto es coherente con las imágenes que estamos usando, pues, aunque a simple vista es muy difícil determinar el origen de la cascada, los patrones que las diferencian son en su mayoría simples.

Como ya hemos repetido varias veces, no podemos afirmar que este sea el mejor modelo, pues no hemos hecho suficientes pruebas y puede que ajustar más algunos parámetros hagan que los otros modelos lo igualen o incluso lo superen. Sin embargo, el número de pruebas necesario para esto es muy alto y, con las pruebas ya realizadas, vemos cómo hemos conseguido unos resultados muy buenos. En este punto, es más interesante modificar las arquitecturas con la idea de tratar de recoger la información que estamos perdiendo al

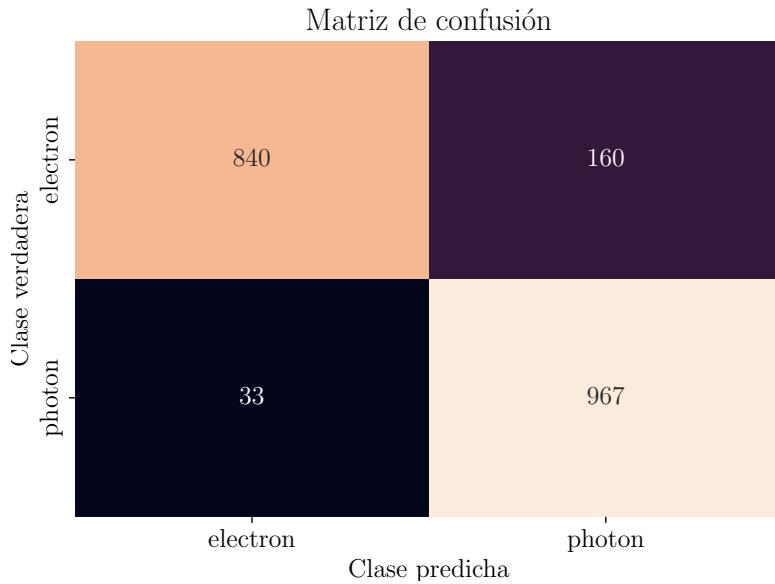
construir las imágenes que seguir con la búsqueda de hiperparámetros. Para estas modificaciones haremos uso de EfficientNet B2 al ser el que mejores resultados nos ha dado. En caso de tener problemas por la complejidad del modelo, usaremos alguno más simple.

## 6.6. Breve estudio del modelo B2

Antes de hacer las modificaciones sobre la arquitectura, vamos a tratar de elaborar un breve estudio del modelo obtenido. Recordemos que estos modelos son “cajas negras” cuya interpretación es muy compleja. En esta sección, vamos a ver algunas métricas que nos digan cómo de bueno es el modelo y veremos las imágenes en las que el modelo se está equivocando. Además, obtendremos la eficiencia y pureza del modelo en función de la energía y veremos cómo se modifican los resultados si quitamos *hits* de la imagen.

En la Tabla 6.7, se muestran distintas métricas para evaluar el modelo para los distintos conjuntos de datos, donde se recogen el porcentaje de acierto, el F1, el AUC, la eficiencia y la pureza. Por un lado, vemos cómo prácticamente no tenemos sobreajuste, hay una diferencia en torno a un 1% entre los resultados de entrenamiento y de prueba. Por otro lado, tenemos una pureza muy superior a la eficiencia, es decir, tenemos muchos eventos que clasificamos como fotones, pero son electrones (muchos falsos positivos que reducen la eficiencia). Sin embargo, tenemos muy pocos eventos que clasificamos como electrones si son fotones (pocos falsos negativos por los que aumentamos la pureza). Esto último lo podemos ver en la matriz de confusión del conjunto de prueba que se muestra en la Fig. 6.3. Con todo esto podemos ver cómo hemos obtenido un modelo que obtiene muy buenos resultados, sobre todo en la pureza, que es especialmente interesante al estar trabajando en un problema de descubrimiento de una nueva partícula.

Por otro lado, en la Fig. 6.4 se muestra cómo evolucionan la eficiencia y la pureza en función de la energía, así como con el producto de estas dos métricas. Por un lado, podemos ver cómo la pureza se mantiene más o menos constante con la energía, mientras que la eficiencia es ligeramente peor a bajas energías. Podemos ver cómo, aunque es más o menos constante con la energía, si se observa que el rendimiento es ligeramente inferior a bajas energías, sobre todo en la eficiencia. No obstante, tenemos que tener en cuenta que,



**Figura 6.3:** Matriz de confusión obtenida usando el mejor modelo EfficientNet B2 obtenido sobre el conjunto de prueba.

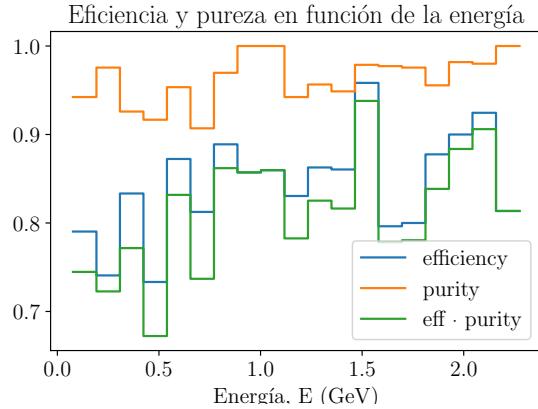
	Entrenamiento	Validación	Prueba
Porcentaje de acierto	91.8	91.3	90.4
F1	0.921	0.920	0.909
AUC	0.979	0.968	0.967
Eficiencia	0.865	0.854	0.840
Pureza	0.967	0.961	0.962

**Tabla 6.7:** Resultados para distintas métricas del mejor modelo obtenido usando la proyección del eje  $y$  y con el model EfficientNet B2.

en estos casos, los eventos están dados por 5-10 *hits* y que, con esta poca información, el modelo es capaz de determinar bastante bien el tipo de cascada observada.

A continuación, vamos a ver cómo se comporta el modelo si lo entorpecemos quitando *hits* de los eventos con la idea de interpretar, dentro de lo posible, el modelo obtenido. Para esto, hemos quitado de los *hits* de tres formas distintas: aleatoriamente, por energía y temporalmente.

En la Fig. 6.5 se muestra cómo influye en el modelo que se pierdan un porcentaje aleatorio de los *hits*. En esta gráfica, se observa cómo tenemos una disminución progresiva de los resultados al ir reduciendo la cantidad de píxeles que el modelo recibe hasta que

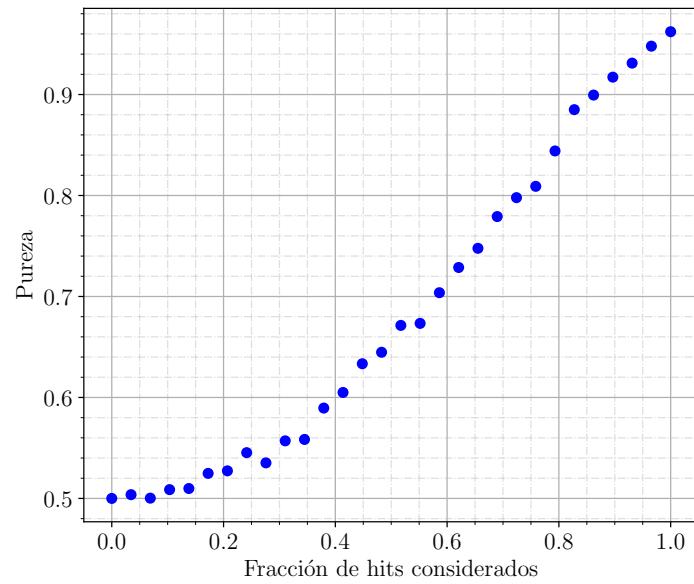


**Figura 6.4:** Eficiencia y pureza del mejor modelo EfficientNet B2 obtenidas sobre el conjunto de prueba al variar la energía.

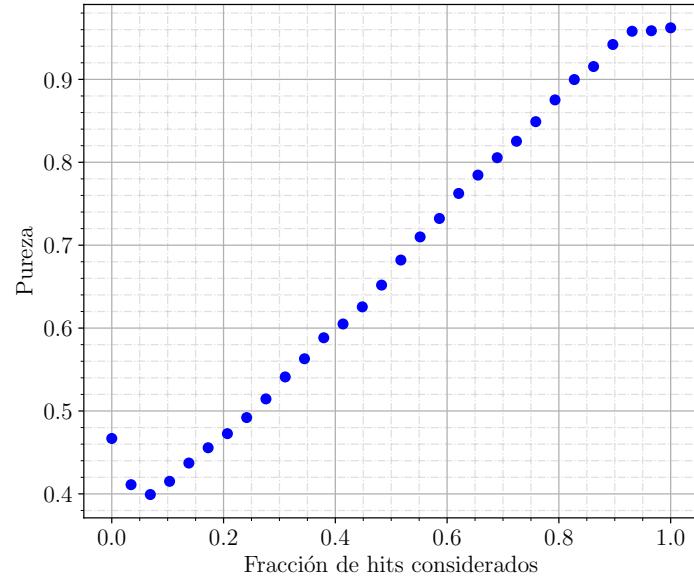
llega un punto, a partir del cual el modelo predice aleatoriamente que tipo de partícula origina el evento. En la Fig. 6.6 se muestra una gráfica equivalente, pero ahora estamos quitando gradualmente los *hits* más energéticos. Vemos cómo si quitamos solo los más energéticos la diferencia es mínima, no obstante, a medida que aumenta la cantidad de *hits* que quitamos, vemos cómo la pureza comienza a disminuir hasta que llega a tener una pureza inferior al 50 %. Esto que observamos es interesante, pues parece que los *hits* menos energéticos son más importantes para clasificar fotones. Por esto, cuando el modelo recibe únicamente estos, se observa un aumento de los falsos negativos, es decir, muchos eventos que son electrones se predicen como fotones porque le estamos dando los rasgos que usa el modelo para determinar los fotones. También lo podemos ver al contrario, la red usa los más energéticos para identificar los electrones, pero al no verlos en la imagen, esta asume que es un fotón.

Por último, en la Fig. 6.7 se muestra cómo influye en el rendimiento del modelo que quitemos los *hits* ordenados temporalmente, es decir, para cada imagen quitamos una fracción de los *hits* empezando por los que se produjeron más tarde<sup>3</sup>. Vemos cómo el comportamiento es muy distinto al observado en las gráficas anteriores. Si nos fijamos en el eje *x* de la Fig. 6.7, no se observa una caída en el rendimiento hasta que se reduce la fracción por debajo del 0.05, es decir, nos quedamos solo con unos pocos *hits* del principio de la cascada. Estos resultados nos están mostrando cómo el modelo está realizando la

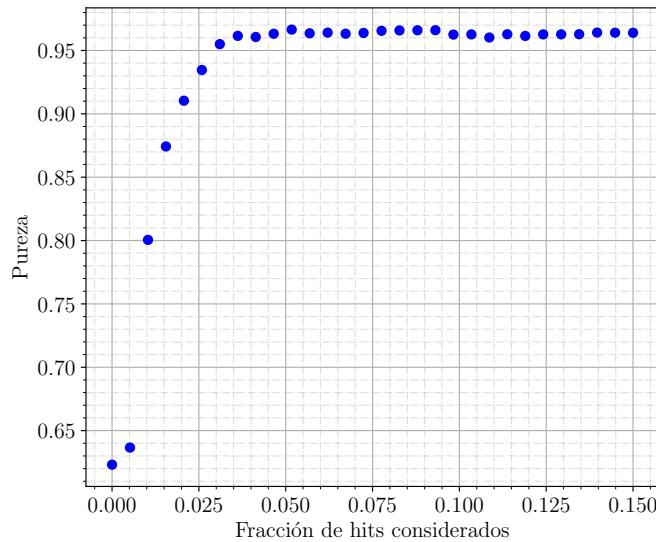
<sup>3</sup>Concretamente, lo que hacemos es crear una variable que tenga los *hits* y el tiempo, obtenemos los cuantiles y nos quedamos solo con la fracción más temprana. Es decir, obtenemos un valor de tiempo que hace que mantengamos un porcentaje concreto de *hits*.



**Figura 6.5:** Evolución de la pureza al reducir de forma aleatoria en número de *hits* que recibe el modelo para realizar la clasificación.



**Figura 6.6:** Evolución de la pureza al reducir de forma progresiva los *hits* que recibe el modelo para realizar la clasificación de más energéticos a menos.



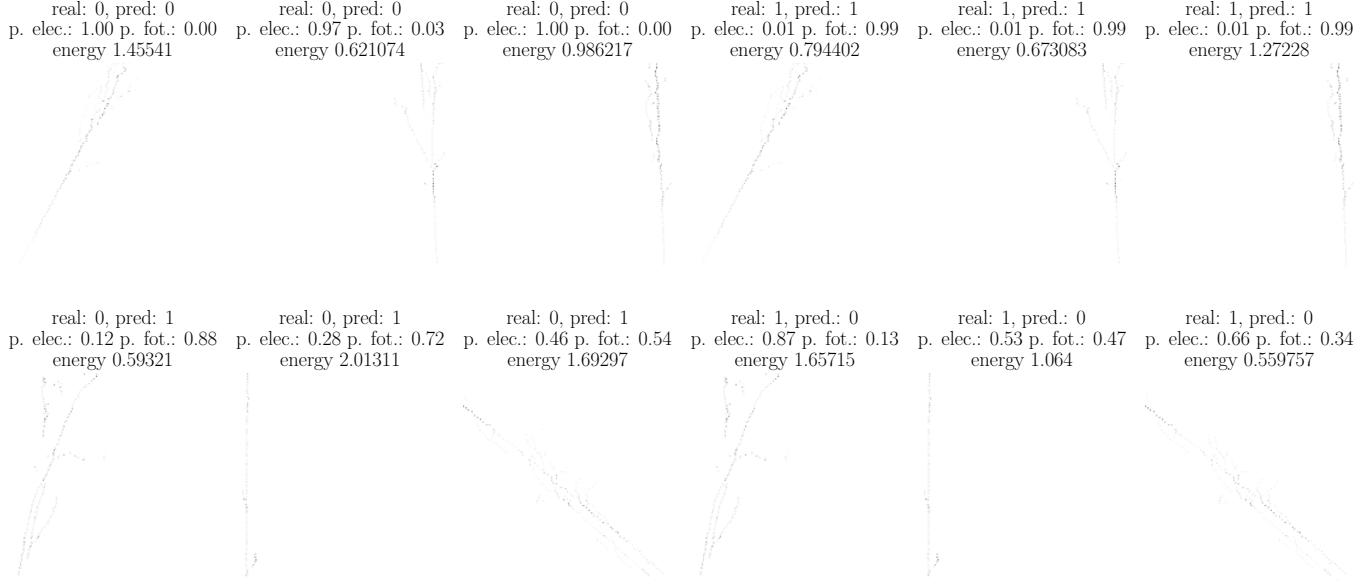
**Figura 6.7:** Evolución de la pureza al reducir de forma progresiva los *hits* que recibe el modelo para realizar la clasificación. Primero se eliminan los *hits* producidos más tarde hasta llegar a los más nuevos.

clasificación utilizando únicamente los primeros *hits*, siendo el resto innecesarios para la discriminación.

Cómo ya se ha mencionado anteriormente, el grupo de investigación con el que se está realizando este trabajo, hizo uso de métodos de ML no basados en DL para resolver el mismo problema que se trata aquí. Los resultados que se obtuvieron con estos métodos mostraban un comportamiento similar al que se observa con el modelo aquí presentado: usar la cascada completa no mejora los resultados si se usa únicamente el origen. Este resultado es coherente con la física del problema, pues sabemos que la diferencia entre las cascadas se encuentra en su mayoría en el origen. No obstante, hay información en otras regiones pero nuestro modelo no está siendo capaz de aprenderla dejando un posible margen de mejora.

Finalmente, podemos ver en qué imágenes el modelo se ha equivocado y en cuáles el modelo ha clasificado correctamente. Para esto hemos hecho uso de Jupyter Notebook, donde se muestran eventos aleatorios ordenados por clase. Además, lo hemos hecho de manera que se muestren por separado imágenes donde el modelo acierta y donde se equivoca para cada clase. En la Fig. 6.8 mostramos la gráfica generada por el código y nos permite

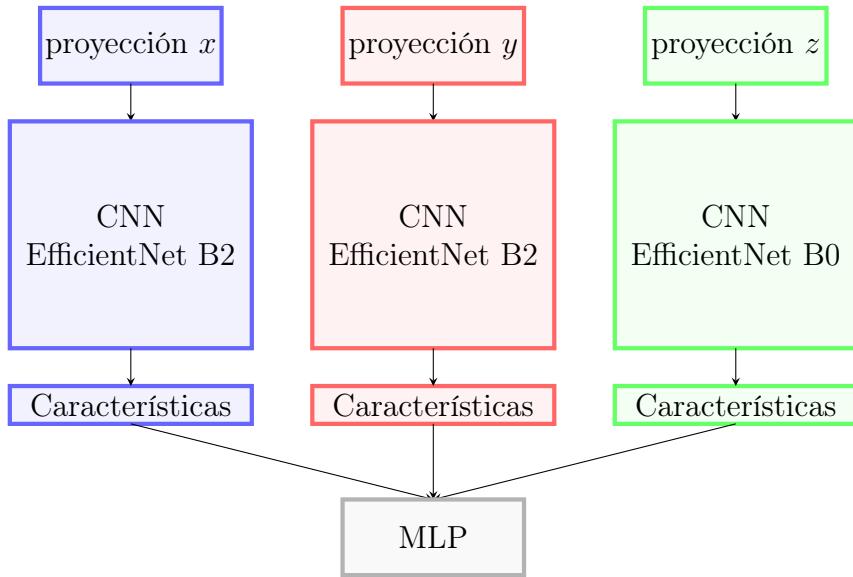
visualizar las imágenes con la idea de visualizar si hay algún problema o ver si podemos observar algún patrón en las imágenes que el modelo confunde.



**Figura 6.8:** Distintos eventos clasificados por el mejor modelo EfficientNet B2 obtenido. Se muestran eventos de electrones y fotones correctamente clasificados (primera fila) y eventos de electrones y fotones que no se han clasificado bien.

## 6.7. Concatenación de redes, 3 proyecciones

Hasta ahora hemos usado únicamente una de las proyecciones para realizar la clasificación, pues con esta representación se obtenían los mejores resultados. En esta sección, vamos a tratar de mejorar los resultados entrenando un modelo para cada una de las proyecciones y concatenando las características extraídas por cada uno de los modelos. Con estas características, entrenamos un MLP para efectuar la clasificación. Podemos entender este modelo como una red que mira las tres vistas de la trayectoria y, a partir de estas tres, toma la decisión. En la Fig. 6.9 mostramos esquemáticamente la arquitectura del modelo



**Figura 6.9:** Representación gráfica de un modelo que concatena las características extraídas de las tres proyecciones de la trayectoria para realizar la clasificación.

presentando, donde usamos como extractores de características modelos ya entrenados que obtengan buenos resultados.

Para entrenar esta red, primero entrenamos los modelos por separado con el objetivo de realizar la clasificación en cada una de las proyecciones. El modelo que hemos usado, ha sido EfficientNet B2 con los mismos hiperparámetros que se usaron para obtener el mejor modelo con la proyección en el eje  $y$ . Los resultados que se obtuvieron fueron 90.1%, 91.3% y 51.6% en validación para las proyecciones en los ejes  $x$ ,  $y$  y  $z$ , respectivamente. Indicar que no se observaba sobreajuste. Vemos como para la proyección  $z$  se tienen resultados cercanos al 50%, por lo que probamos a reducir la complejidad utilizando redes más simples y una mayor resolución para centrarnos más en el origen de la cascada. Con estos cambios se obtuvo un 82.2% usando EfficientNet B0 con una resolución  $c_x = 6000$  y un MLP de una capa oculta con 128 neuronas.

Si a estos modelos les quitamos los clasificadores, tenemos modelos que nos extraen las características necesarias para clasificar las cascadas. A continuación, se creó un modelo que las concatena y las usa para la clasificación. En la Tabla 6.8, se muestran distintas métricas para evaluar el modelo obtenido tras realizar el entrenamiento, tanto en el conjunto de entrenamiento como en los conjuntos de validación y de prueba. Si comparamos con los resultados obtenidos con el modelo con una sola proyección, vemos que los resultados

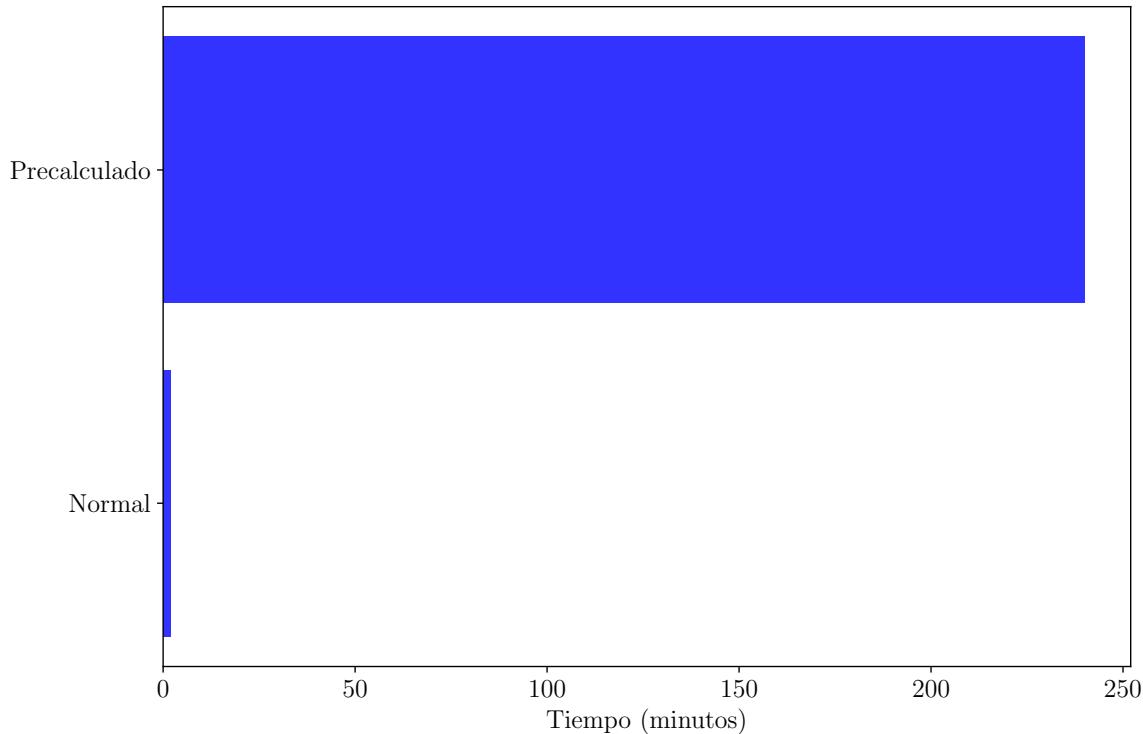
han empeorado en validación y prueba, pero en entrenamiento sí hemos observado una mejora. Sin embargo, los resultados obtenidos en conjuntos vistos durante el entrenamiento son mucho mejores que los que no han sido vistos, es decir, parece que el modelo está sobreajustando.

La implementación de este nuevo modelo es bastante compleja, pues requiere de cargar las imágenes de las tres proyecciones distintas. Además, la construcción del modelo también se dificulta, pues requiere de tres modelos en paralelo, concatenar una parte intermedia de estos modelos y, finalmente, entrenar el nuevo MLP. El principal problema de este modelo es que es bastante lento de entrenar, tarda unas cuatro horas. Sin embargo, una alternativa a esta implementación es precalcular primero las características extraídas por los modelos entrenados y crear un nuevo conjunto de datos que tenga, en vez de imágenes, la concatenación de estas características. Sobre este nuevo conjunto de entradas, creamos un modelo MLP, que en Pytorch es muy simple, y lo entrenamos. El entrenamiento usando esta implementación es mucho más rápido, disminuyendo el tiempo aprendizaje hasta los dos minutos, obteniendo resultados similares en el conjunto de pruebas. En la Fig. 6.10 se muestra de forma gráfica la diferencia de tiempo.

	Entrenamiento	Validación	Prueba
Porcentaje de acierto	93.6	89.3	89.4
F1	0.935	0.899	0.896
AUC	0.982	0.957	0.959
Eficiencia	0.915	0.868	0.867
Pureza	0.951	0.908	0.916

**Tabla 6.8:** Resultados para distintas métricas del modelo donde se concatenan las salidas de distintos modelos con distintas proyecciones.

Por otro lado, otra cosa que hemos probado con la misma filosofía que se tenía en el modelo que concatena redes, es hacer un *ensemble*. Para ello, vamos a usar los modelos entrenados para cada una de las proyecciones y obtener las probabilidades de que los eventos sean cascadas originadas por electrones. De esta forma, se obtiene una terna de probabilidades para cada instancia. Con estos valores construimos un nuevo conjunto de datos, donde cada columna contiene la probabilidad obtenida por los distintos modelos de que la cascada sea de origen electrónico. Además, con este nuevo conjunto de datos junto con las clases verdaderas, entrenamos unos nuevos clasificadores, por ejemplo un Random Forest. Los resultados obtenidos con este enfoque se muestran en la Tabla 6.9. Vemos



**Figura 6.10:** Comparación entre los tiempos de aprendizaje de los dos modelos que concatenan características extraídas por CNN. El modelo normal es el que construye una red con los tres modelos a la vez. El modelo precalculado consiste en un MLP donde se precisan las características de las tres proyecciones.

cómo los resultados que estamos obteniendo, para las distintas métricas de evaluación, son peores que los obtenidos al utilizar una única proyección. Los clasificadores han sido entrenados usando el conjunto completo de entrenamiento, aplicando una búsqueda de rejilla con validación cruzada.

En esta última tabla se observa bastante sobreajuste en el conjunto de pruebas. Al haber entrenado usando validación cruzada, no podemos ver si tenemos sobreajuste en los conjuntos de validación. Para asegurarnos, vamos a volver a entrenar dejando un conjunto de validación. De esta forma, obtenemos un 81.8 % de acierto en validación que sí se asemeja a lo que observamos en el conjunto de prueba.

	Random Forest		Regresión logística	
	Entrenamiento	Prueba	Entrenamiento	Prueba
Porcentaje de acierto	90.7	79.6	90.6	79.7
F1	0.908	0.800	0.908	0.804
AUC	0.964	0.886	0.967	0.901
Eficiencia	0.900	0.775	0.884	0.764
Pureza	0.914	0.809	0.921	0.818

**Tabla 6.9:** Resultados para distintas métricas de los *ensemble*. Estos se consrtuyen usando tres modelos distintos para obtener las probabilidades y se usa un nuevo clasificador para predecir usando estas probabilidades.

En definitiva, vemos cómo los resultados obtenidos con estos nuevos enfoques, que tratan de obtener mayor información de los eventos, no presentan una mejora con los obtenidos con los modelos simples que hacen uso de una única proyección.

# Capítulo 7

## Conclusiones y perspectiva de futuro

En este trabajo, hemos estudiado en detalle las cascadas electromagnéticas originadas en el interior de los detectores LArTPC para tratar determinar su origen. Determinar el origen permite contar el número de neutrinos electrónicos que llegan al detector, fundamental para la detección del neutrino estéril.

Para llevar a cabo este trabajo, hemos realizado toda la implementación desde cero, teniendo que ver cómo procesar los datos, qué técnicas de preprocesado emplear y qué modelos usar. En este trabajo hemos utilizado imágenes para representar los datos, pues nos hemos centrado en modelos de Deep Learning para hacer la clasificación. Por otro lado, haber diseñado cómo crear las imágenes nos ha permitido establecer variables que controlen la creación y estudiar su efecto en los resultados. Además, hemos abordado diversas formas de preprocesar las imágenes con el mismo fin.

En cuanto a implementación, hemos tenido que aprender a usar Pytorch y esta se ha efectuado haciendo uso de prácticas de MLOps, puesto que las consideramos fundamentales. De hecho, su uso nos ha permitido, por un lado, mantener un seguimiento de todas las pruebas obtenidas y, por otro lado, ir añadiendo nuevas funcionalidades de manera efectiva y simple. También, el haber utilizado estas herramientas simplificó notablemente utilizar un cluster, que fue necesario para realizar distintas pruebas, pues se ha podido establecer un servidor donde recoger los modelos obtenidos. No obstante, como consecuencia de la falta de experiencia, el diseño es mejorable, sobre todo en los primeros programas desarrollados. Además, hay algunas herramientas que no hemos empleado debido a su complejidad, en

trabajos futuros se pretende hacer uso de ellas para una mayor eficacia a la hora de elaborar nuevo código.

También, me gustaría remarcar que todo el desarrollo ha sido realizado desde cero y que la implementación de toda la solución es bastante compleja, en gran parte, debido a la integración de las distintas técnicas de MLOps o el uso de herramientas como MLFlow, que hacen más difícil el diseño, que no la implementación. No obstante, simplifican otras muchas cosas y son fundamentales para el trabajo realizado.

Por otro lado, una gran parte de este trabajo ha consistido en hacer un estudio exhaustivo del tratamiento de los datos propuesto y hemos visto que:

- Usar una transformación logarítmica sobre la carga de los *hits* no ha mostrado dar buenos resultados. La aplicación de la normalización global y una normalización por *batch*, tampoco ha dado lugar a mejora. De hecho, en general se observa un peor rendimiento y un aumento en el sobreajuste.
- De todas las representaciones planteadas, quedarnos únicamente con una proyección de la trayectoria ha sido la que mejor resultados nos ha dado. El resto de representaciones mostraban resultados interesantes, pero inferiores.
- Al variar la resolución, hemos observado que, en general, a mayor resolución mejores resultados. Es decir, que para la clasificación, una mayor definición del origen de la cascada parece ser mejor que una mayor cantidad de cascada, pero con peor definición.

Otra gran parte de este trabajo ha consistido en obtener modelos que obtenga los mejores resultados posibles, objetivo principal de este trabajo. El mejor modelo ha sido obtenido con la arquitectura EfficientNet B2 y ha alcanzado un 96.2 % de pureza sobre el conjunto de prueba. Hemos probado a usar distintos modelos, algunos diseñados por nosotros y otros del estado del arte, pero, en general, los distintos modelos han obtenido resultados similares tras realizar la búsqueda de hiperparámetros. Esto puede estar indicando que estamos exprimiendo al máximo la representación usada condicionados al uso de CNN. Por otro lado, hemos estudiado el modelo obtenido y hemos visto que podemos quitar muchos *hits* si estos no pertenecen al origen de la cascada. Es decir, las cascadas se caracterizan prácticamente por su origen, en concordancia con la física detrás del problema.

Una cuestión que podemos destacar de nuestro enfoque es que estamos descartando una gran parte de la trayectoria de la cascada al crear la imagen, tanto al tomar la ventana como al realizar la proyección. Por lo tanto, hemos buscado formas de tratar incluir más información en los modelos. Para ello, hemos agregado modelos entrenados con las distintas proyecciones. Para la agregación, hemos utilizado ensembles y un nuevo modelo que usa las características de los modelos para la clasificación. Sin embargo, los resultados observados no mejoran los iniciales, por lo que es interesante seguir buscando formas de mejorar estos resultados al agregar más información.

Con todo esto, consideramos que el trabajo es un éxito, pues hemos obtenido un modelo que hace una correcta clasificación de las cascadas electromagnéticas. Además, hemos realizado un estudio exhaustivo sobre distintos aspectos, permitiéndonos delimitar futuros trabajos que traten de mejorar los resultados obtenidos.

Finalmente, este trabajo ha sido nuestro primer paso en el uso del Deep Learning para la clasificación de cascadas electromagnéticas. Con él, hemos conseguido muy buenos resultados, pero dejamos muchas posibles líneas de trabajo futuro:

- Una de las representaciones planteadas, no está basada en imágenes y hace uso de transformers. Nosotros no la hemos probado, por lo que es interesante ver su rendimiento al clasificar imágenes.
- Seguir estudiando métodos que acoplen la información de distintas proyecciones, pues nosotros no hemos conseguido obtener ningún resultado prometedor.
- Los datos proporcionados se limitan únicamente a las cascadas electromagnéticas, pero los eventos del detector contienen también las otras partículas de la interacción. Por lo tanto, tenemos que ver como abordar el problema al introducir estas modificaciones sobre los datos.

# Anexo A

## Hiperparámetros de los estudios

En este anexo se recoge una descripción detallada de los hiperparámetros empleados en los distintos estudios realizados.

### A.1. Estudio de la capa de normalización por *batch* y transformación logarítmica

Para este estudio, hemos variado únicamente el *dropout*, la normalización, el uso de una capa de normalización por *batch* y el uso de la transformación logarítmica. Los valores de estos parámetros son los que se mostraban en la Tabla 6.1. El resto de hiperparámetros se ha mantenido fijo y se recogen en la Tabla A.1.

### A.2. Estudio de la representación

Para este estudio hemos variado la representación para distintos modelos, donde lo que hemos cambiado principalmente ha sido la proyección. No obstante, algunos de los hiperparámetros dependen del modelo por lo que para los distintos modelos se tienen distintos hiperparámetros. Por otro lado, en estos experimentos se ha tratado de mantener fijo el resto de hiperparámetros cuando se ha cambiado de representación, sin embargo, en

Nombre	Valor	Nombre	Valor	Nombre	Valor
seed	19	conv sizes	(3, 3)	optim name	adam
cube shape x	1000	conv strides	(1, 1)	optim lr	0.001
win shape	(62, 128)	pool sizes	(2, 2)	scheduler name	steplr
projection	y	pool strides	(2, 2)	sched. step size	3
projection pool	max	clf neurons	(32, 8, 2)	scheduler gamma	0.7
cube pool	mean	no linear fun	relu	—	—
model name	convnet	batch size	48	—	—
conv filters	(16, 24)	n epochs	10	—	—

**Tabla A.1:** Hiperparámetros fijos de los experimentos recogidos en la Tabla 6.1.

algunos casos se han tenido que modificar por problemas en el aprendizaje. En la Tabla A.2 se recogen los hiperparámetros que se han mantenido constantes en todos los experimentos, para todos los modelos y para todos las representaciones.

En la Tabla A.3 se muestran los parámetros que varían con el modelo y se han marcado con asteriscos aquellos hiperparámetros que han tenido que modificarse al cambiar de representación. En concreto, para el *lr* hemos tenido que reducir el ritmo de aprendizaje para las representaciones de color y 3D, ya que los modelos no aprendían. Por otro lado, en la representación de color, siempre hemos puesto el *dropout* en 0.2, y hemos aplicado la transformación de normalización. El motivo de esto es que al realizar una primeras pruebas se observaba sobreajuste. Además, en esta resolución se aplica en el procesamiento una transformación logarítmica y, en los resultados recogidos en la Tabla 6.1, se observaba como la normalización y el *dropout* ayudaban cuando se realizaba la transformación logarítmica.

Nombre	Valor	Nombre	Valor
seed	19	n epochs	10
cube shape x	1000	no linear fun	relu
projection pool	max	scheduler name	steplr
cube pool	mean	sched. step size	3
BN	False	log trans	False
scheduler gamma	0.7	—	—

**Tabla A.2:** Hiperparámetros fijos de los experimentos recogidos en la Tabla 6.2.

	Simple	Complejo	Preentrenados
win shape	62, 128	62, 128	224, 224
conv filters	(16, 24)	(32, 64, 64, 128)	—
conv sizes	(3, 3)	(3, 3, 3, 3)	—
conv strides	(1, 1)	(1, 1, 1, 1)	—
pool sizes	(2, 2)	(1, 2, 1, 2)	—
pool strides	(2, 2)	(1, 2, 1, 2)	—
clf neurons	(32, 8, 2)	(50, 100, 50, 2)	(32, 8, 2)
batch size	48	72	48
unfreeze layers	—	—	10
optim lr*	0.001	0.0001	0.001
dropout*	0	0	0.2
transform*	No	No	Si

**Tabla A.3:** Hiperparámetros fijos para las distintas representaciones pero que varían entre modelos. Corresponden a los experimentos recogidos en la Tabla 6.2 y los asteriscos indican que en alguna representación se ha tenido que modificar. Los valores vacíos significan que el modelo no tiene ese hiperparámetro.

### A.3. Estudio de la resolución

En este estudio, hemos variado la resolución para distintos modelos por lo que tenemos la misma situación que para el estudio de la representación. En la Tabla A.4 se muestran los hiperparámetros fijos en todas las pruebas realizadas y en la Tabla A.5 los parámetros que dependen del modelo.

Al realizar estas pruebas observamos que el modelo complejo presenta problemas a la hora de entrenar al cambiar las condiciones. Debido a esto tuvimos que modificar ligeramente los hiperparámetros como el tamaño del *batch* y el factor  $\gamma$ .

Nombre	Valor	Nombre	Valor
seed	19	n epochs	10
projection	y	no linear fun	relu
projection pool	max	scheduler name	steplr
cube pool	mean	sched. step size	3
BN	False	log trans	False

**Tabla A.4:** Hiperparámetros fijos de los experimentos recogidos en la Tabla 6.3.

	Simple	Complejo	Preentrenados
win shape	62, 128	62, 128	224, 224
conv filters	(16, 24)	(32, 64, 64, 128)	—
conv sizes	(3, 3)	(3, 3, 3, 3)	—
conv strides	(1, 1)	(1, 1, 1, 1)	—
pool sizes	(2, 2)	(1, 2, 1, 2)	—
pool strides	(2, 2)	(1, 2, 1, 2)	—
unfreeze layers	—	—	10
clf neurons	(32, 8, 2)	(50, 100, 50, 2)	(32, 8, 2)
batch size	48	64	48
optim lr	0.001	0.0001	0.001
transform	No	No	Si
scheduler gamma	0.7	1	0.7

**Tabla A.5:** Hiperparámetros fijos para las distintas resoluciones pero que varían entre modelos. Corresponden a los experimentos recogidos en la Tabla 6.3. Los valores vacíos significan que el modelo no tiene ese hiperparámetro.

## A.4. Ajuste del modelo LeNet simple

En este conjunto de prueba se ha usado el modelo LeNet simple y se han modificado el  $lr$ , el desplazamientos de los filtros, el tamaño de los filtros y el factor  $\gamma$ . El resto de parámetros se mantiene constante y se muestran en la Tabla A.6.

Nombre	Valor	Nombre	Valor	Nombre	Valor
seed	19	BN	False	optim name	adam
cube shape x	1500	log trans	False	optim lr	0.001
win shape	62, 128	pool sizes	(2, 2)	scheduler name	steplr
projection	y	pool strides	(2, 2)	sched. step size	3
projection pool	max	clf neurons	(32, 8, 2)	dropout	0
cube pool	mean	no linear fun	relu	—	—
model name	convnet	batch size	48	—	—
conv filters	(16, 24)	n epochs	10	—	—

**Tabla A.6:** Hiperparámetros fijos de los experimentos recogidos en la Tabla 6.4.

## A.5. Ajuste del modelo LeNet complejo

Este experimento consiste en el ajuste de algunos de los hiperparámetros del modelo LeNet complejo. Se han modificado el  $lr$ , el desplazamientos de los filtros, el tamaño de los filtros y el factor  $\gamma$ . Sin embargo, por problemas de aprendizaje, hemos tenido que usar valores distintos del tamaño del *batch* (*bs*) y de la frecuencia de cambio del  $lr$  (*step*) al cambiar el tamaño del filtro. Para el tamaño de filtro de  $5 \times 5$  hemos usado  $bs = 48$  y  $step = 3$  mientras que para  $3 \times 3$  hemos usado  $bs = 74$  y  $step = 5$ . Por otro lado, mencionar que, para los desplazamientos de tamaño dos, se ha tenido que modificar el modelo añadiendo *padding*. En la Tabla A.7 se recogen el resto de hiperparámetros que sí se han mantenido fijos en todo el conjunto de prueba.

Nombre	Valor	Nombre	Valor	Nombre	Valor
seed	19	BN	False	optim name	adam
cube shape x	1500	log trans	False	scheduler name	steplr
win shape	62, 128	sched. step size	3	transform	True
projection	y	dropout	0	pool sizes	(1, 2, 1, 2)
projection pool	max	n epochs	10	pool strides	(1, 2, 1, 2)
cube pool	mean	no linear fun	relu	—	—
conv filters	(32, 64, 64, 128)	model name	convnet	—	—

**Tabla A.7:** Hiperparámetros fijos de los experimentos recogidos en la Tabla 6.5.

## A.6. Ajuste modelos preentrenados: Optimización

Antes de hacer el ajuste de hiperparámetros de los modelos complejos, tratamos de determinar unos valores de  $lr$  y de  $\gamma$  manteniendo el resto de hiperparámetros constantes. Para estas pruebas hemos mantenido fijo tanto el modelo como el número de neuronas del clasificador. Todos estos hiperparámetros se recogen en la Tabla A.8.

Nombre	Valor	Nombre	Valor	Nombre	Valor
seed	19	BN	False	optim name	adam
cube shape x	3000	log trans	False	unfreeze layers	50
win shape	224, 224	sched. step size	3	scheduler name	steplr
projection	y	dropout	0.2	transform	True
projection pool	max	clf neurons	(256, 2)	n epochs	10
cube pool	mean	no linear fun	relu	—	—
batch size	48	model name	Effb2	—	—

**Tabla A.8:** Hiperparámetros fijos de los experimentos cuyos resultados se muestran en la Fig. 6.2.

Nombre	Valor	Nombre	Valor	Nombre	Valor
seed	19	BN	False	optim name	adam
cube shape x	3000	log trans	False	optim lr	0.01
win shape	224, 224	sched. step size	3	scheduler name	steplr
projection	y	dropout	0.2	transform	True
projection pool	max	scheduler gamma	0.5	—	—
cube pool	mean	no linear fun	relu	—	—
batch size	48	n epochs	10	—	—

**Tabla A.9:** Hiperparámetros fijos de los experimentos recogidos en la Tabla 6.6.

## A.7. Ajuste modelos preentrenados

Este último estudio consiste en ajustar el número de capas a descongelar y el número de neuronas usado en el clasificador para los modelos preentrenados. El resto de parámetros se han mantenido constantes y se recogen en la Tabla A.9.

# Anexo B

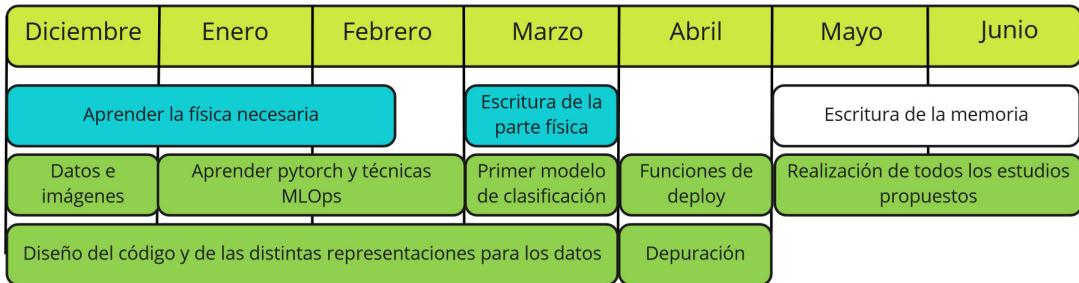
## Planificación y Presupuesto

### B.1. Planificación

En este anexo se muestra cómo se ha organizado este trabajo, desde su comienzo en diciembre, hasta que su finalización en junio. En la Fig. B.1 mostramos un diagrama de Gantt donde se muestran las principales tareas necesarias, así como una estimación del tiempo requerido para llevarlas a cabo. Indicar que en el tiempo dedicado es de unas 16 horas por semana, salvo el último mes, donde dedicamos unas 40 horas por semana al disponer de más tiempo.

Los primeros meses se dedican a aprender lo necesario para resolver el problema propuesto, es decir, las cuestiones físicas y de programación. Además, dedicamos tiempo a estudiar los datos proporcionados, así como de visualizar los eventos. En paralelo a todo esto, los primeros meses se tienen que dedicar a diseñar el código junto con las distintas formas de representar los datos hasta que se empiecen a usar los modelos de clasificación.

En marzo, ya se debería conocer perfectamente los datos y el problema, por lo que lo dedicamos a la escritura de la parte física y el entrenamiento de los primeros modelos para ver que todo funciona correctamente. El mes de abril, lo dejamos para establecer el entorno para las futuras pruebas, así como para depurar el código. Finalmente, en los dos últimos meses, realizamos los distintos experimentos y escribimos la memoria del trabajo.



**Figura B.1:** Diagrama de Gantt donde se muestra la planificación llevada a cabo para realizar este trabajo. En azul se indican las actividades relacionadas con la física y en verde las relacionadas con el código y la ciencia de datos.

## B.2. Presupuesto

A modo informativo, en este anexo vamos a hacer una breve estimación del precio que supondría realizar el desarrollo realizado en este trabajo. Para llevarlo a cabo es necesario de un equipo con una GPU y potencia suficiente para realizar las pruebas, así como tener a una persona realizando el estudio.

El tiempo dedicado para este trabajo ha sido de dos días a la semana, unas 8 horas cada día desde diciembre hasta junio, salvo el último mes en los que he trabajado 5 días a la semana. Esto resulta en unos 32 días de trabajo a 8 horas, por lo que podemos considerar que el trabajo se puede hacer en un mes y medio de trabajo con una jornada habitual. Indicar que esto es una aproximación y que probablemente no se estén considerando muchas horas de trabajo, por ejemplo reuniones. Por esto, vamos a considerar que el tiempo de desarrollo es de unos dos meses. En cuanto al equipo empleado, se va a hacer la estimación del precio usando servicios en la nube que nos permiten crear máquinas virtuales con unas especificaciones concretas.

En la Tabla B.1, se muestran los costes de personal y equipo necesarios para realizar todo desarrollo que hemos llevado a cabo en este trabajo, siendo el coste total de unos 5740.02€.

	Cantidad	Meses	€/mes	€(total)
Personal	1	2	2750	5500
Equipo	1	2	120.01	240.02

**Tabla B.1:** Coste estimado para la realización del desarrollo hecho en este trabajo. El coste del personal ha sido estimado usando el salario medio de un ingeniero informático [69] y el coste del equipo ha sido obtenido usando la calculadora de precios para la construcción de máquinas virtuales de Google Cloud (ver Fig. B.2).

Region: Madrid	
160 total hours per month	
Provisioning model: Regular	
Instance type: n1-standard-4	EUR 33.48
Operating System / Software: Free	
GPU dies: 1 NVIDIA TESLA K80	EUR 79.29
Local SSD: 1x375 GiB	EUR 7.24
<b>Estimated Component Cost: EUR 120.01 per 1 month</b>	
<b>Total Estimated Cost: EUR 120.01 per 1 month</b>	

**Figura B.2:** Estimación del coste mensual de una máquina virtual capaz de realizar el trabajo aquí presentado.

# Bibliografía

- [1] Xiongwei Wu, Doyen Sahoo y Steven CH Hoi. Recent advances in deep learning for object detection. En: *Neurocomputing* 396 (2020), págs. 39-64.
- [2] Rui Qian, Xin Lai y Xirong Li. 3D object detection for autonomous driving: a survey. En: *Pattern Recognition* (2022), pág. 108796.
- [3] Ashish Vaswani *et al.* Attention is all you need. En: *Advances in neural information processing systems* 30 (2017).
- [4] Tom Brown *et al.* Language models are few-shot learners. En: *Advances in neural information processing systems* 33 (2020), págs. 1877-1901.
- [5] Tianyang Lin *et al.* A Survey of Transformers. 2021. DOI: 10.48550/ARXIV.2106.04554. URL: <https://arxiv.org/abs/2106.04554>.
- [6] Alexander Radovic *et al.* Machine learning at the energy and intensity frontiers of particle physics. En: *Nature* 560.7716 (2018), págs. 41-48.
- [7] Pedro AN Machado, Ornella Palamara y David W Schmitz. The short-baseline neutrino program at Fermilab. En: *Annual Review of Nuclear and Particle Science* 69 (2019), págs. 363-387.
- [8] G Rajasekaran. The Story of the Neutrino. 2016. DOI: 10.48550/ARXIV.1606.08715. URL: <https://arxiv.org/abs/1606.08715>.
- [9] David Griffiths. Introduction to elementary particles. John Wiley & Sons, 2020.
- [10] Susanne Mertens. Direct Neutrino Mass Experiments. En: *Journal of Physics: Conference Series* 718 (mayo de 2016), pág. 022013.
- [11] Christopher W. Walter and. «The Super-Kamiokande Experiment». En: *Neutrino Oscillations*. WORLD SCIENTIFIC, mar. de 2008, págs. 19-43.

- [12] John David Jackson. Classical electrodynamics. 1999.
- [13] B. Abi *et al.* Deep Underground Neutrino Experiment (DUNE), Far Detector Technical Design Report, Volume I: Introduction to DUNE. 2020. doi: 10.48550/ARXIV.2002.02967. URL: <https://arxiv.org/abs/2002.02967>.
- [14] S. E. Woosley, A. Heger y T. A. Weaver. The evolution and explosion of massive stars. En: *Rev. Mod. Phys.* 74 (4 nov. de 2002), págs. 1015-1071.
- [15] Claus Grupen *et al.* Astroparticle physics. Vol. 50. Springer, 2005.
- [16] DUNE at LBNF. URL: <https://lbnf-dune.fnal.gov/how-it-works/introduction/> (visitado 09-06-2022).
- [17] Maria Concepcion Gonzalez-Garcia, Michele Maltoni y Thomas Schwetz. NuFIT: Three-Flavour Global Analyses of Neutrino Oscillation Experiments. En: *Universe* 7.12 (2021), pág. 459.
- [18] Fermilab Short-Baseline Near Detector. URL: <https://sbn-nd.fnal.gov/> (visitado 27-03-2022).
- [19] Wikimedia particle shower. URL: [https://commons.wikimedia.org/wiki/File:Schematic\\_of\\_a\\_particle\\_shower.svg](https://commons.wikimedia.org/wiki/File:Schematic_of_a_particle_shower.svg).
- [20] Krishanu Majumdar y Konstantinos Mavrokoridis. Review of liquid argon detector technologies in the neutrino sector. En: *Applied Sciences* 11.6 (2021), pág. 2455.
- [21] Yann LeCun, Yoshua Bengio y Geoffrey Hinton. Deep learning. En: *nature* 521.7553 (2015), págs. 436-444.
- [22] Mingxing Tan y Quoc Le. «Efficientnet: Rethinking model scaling for convolutional neural networks». En: *International conference on machine learning*. PMLR. 2019, págs. 6105-6114.
- [23] Ruihui Mu y Xiaoqin Zeng. A review of deep learning research. En: *KSII Transactions on Internet and Information Systems (TIIS)* 13.4 (2019), págs. 1738-1764.
- [24] Ian Goodfellow, Yoshua Bengio y Aaron Courville. Deep learning. MIT press, 2016.
- [25] Aditya Ramesh *et al.* Hierarchical Text-Conditional Image Generation with CLIP Latents. 2022. doi: 10.48550/ARXIV.2204.06125. URL: <https://arxiv.org/abs/2204.06125>.
- [26] Mark Chen *et al.* Evaluating Large Language Models Trained on Code. 2021. doi: 10.48550/ARXIV.2107.03374. URL: <https://arxiv.org/abs/2107.03374>.

- [27] Wikimedia MLP. URL: <https://commons.wikimedia.org/wiki/File:MultiLayerPerceptron.svg>.
- [28] Kurt Hornik, Maxwell Stinchcombe y Halbert White. Multilayer feedforward networks are universal approximators. En: *Neural networks* 2.5 (1989), págs. 359-366.
- [29] Balázs Csanád Csáji *et al.* Approximation with artificial neural networks. En: *Faculty of Sciences, Eötvös Loránd University, Hungary* 24.48 (2001), pág. 7.
- [30] Sebastian Ruder. An overview of gradient descent optimization algorithms. 2016. DOI: 10.48550/ARXIV.1609.04747. URL: <https://arxiv.org/abs/1609.04747>.
- [31] David E Rumelhart, Geoffrey E Hinton y Ronald J Williams. Learning representations by back-propagating errors. En: *nature* 323.6088 (1986), págs. 533-536.
- [32] Herbert Robbins y Sutton Monro. A stochastic approximation method. En: *The annals of mathematical statistics* (1951), págs. 400-407.
- [33] Diederik P. Kingma y Jimmy Ba. Adam: A Method for Stochastic Optimization. 2014. DOI: 10.48550/ARXIV.1412.6980. URL: <https://arxiv.org/abs/1412.6980>.
- [34] Rick Merritt. What is MLOps? Mar. de 2022. URL: <https://blogs.nvidia.com/blog/2020/09/03/what-is-mlops/>.
- [35] Noah Gift y Alfredo Deza. Practical MLOps. O'Reilly Media, Inc., 2021.
- [36] Machine Learning Operations (MLOps). Jul. de 2021. URL: <https://nealanalytics.com/expertise/mlops/>.
- [37] Sridhar Alla y Suman Kalyan Adari. «What is mlops?» En: *Beginning MLOps with MLFlow*. Springer, 2021, págs. 79-124.
- [38] C. Athanassopoulos *et al.* Evidence for  $\bar{\nu}_\mu \rightarrow \bar{\nu}_e$  Oscillations from the LSND Experiment at LAMPF. En: *Physical Review Letters* 77.15 (oct. de 1996), págs. 3082-3085. DOI: 10.1103/physrevlett.77.3082. URL: <https://doi.org/10.1103/2Fphysrevlett.77.3082>.
- [39] AA Aguilar-Arevalo *et al.* The miniboone detector. En: *Nuclear instruments and methods in physics research section a: accelerators, spectrometers, detectors and associated equipment* 599.1 (2009), págs. 28-46.
- [40] R Acciarri *et al.* Design and construction of the MicroBooNE detector. En: *Journal of Instrumentation* 12.02 (2017), P02017.

- [41] P Abratenko *et al.* Search for an Excess of Electron Neutrino Interactions in Micro-BooNE Using Multiple Final State Topologies. En: *arXiv preprint arXiv:2110.14054* (2021).
- [42] Dimitri Bourilkov. Machine and deep learning applications in particle physics. En: *International Journal of Modern Physics A* 34.35 (2019), pág. 1930019.
- [43] Dan Guest, Kyle Cranmer y Daniel Whiteson. Deep learning and its application to LHC physics. En: *Annual Review of Nuclear and Particle Science* 68 (2018), págs. 161-181.
- [44] Alexander Aab *et al.* Deep-learning based reconstruction of the shower maximum X max using the water-Cherenkov detectors of the Pierre Auger Observatory. En: *Journal of instrumentation* 16.07 (2021), P07019.
- [45] Alexander Aab *et al.* Reconstruction of events recorded with the surface detector of the Pierre Auger Observatory. En: *Journal of Instrumentation* 15.10 (2020), P10021.
- [46] Alexey Dosovitskiy *et al.* An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. 2020. DOI: 10.48550/ARXIV.2010.11929. URL: <https://arxiv.org/abs/2010.11929>.
- [47] Dominik Kreuzberger, Niklas Kühl y Sebastian Hirschl. Machine Learning Operations (MLOps): Overview, Definition, and Architecture. 2022. DOI: 10.48550/ARXIV.2205.02302. URL: <https://arxiv.org/abs/2205.02302>.
- [48] Leonardo Leite *et al.* A survey of DevOps concepts and challenges. En: *ACM Computing Surveys (CSUR)* 52.6 (2019), págs. 1-35.
- [49] MLFlow. URL: <https://www.mlflow.org/docs/latest/index.html> (visitado 09-06-2022).
- [50] Takuya Akiba *et al.* «Optuna: A next-generation hyperparameter optimization framework». En: *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*. 2019, págs. 2623-2631.
- [51] Snapper-ML. URL: <https://github.com/SnapperML/SnapperML> (visitado 09-06-2022).
- [52] Antonio Molner Domenech y Alberto Guillén. «ml-experiment: A Python framework for reproducible data science». En: *Journal of Physics: Conference Series*. Vol. 1603. 1. IOP Publishing. 2020, pág. 012025.

- [53] Amazon SageMaker. URL: <https://aws.amazon.com/es/sagemaker/> (visitado 09-06-2022).
- [54] Robert C Martin. Clean code: a handbook of agile software craftsmanship. Pearson Education, 2009.
- [55] P.A. Zyla *et al.* Review of Particle Physics. En: *PTEP* 2020.8 (2020), pág. 083C01. DOI: 10.1093/ptep/ptaa104.
- [56] Javier León *et al.* Photon/electron classification in liquid argon detectors by means of Soft Computing. En: *Engineering Applications of Artificial Intelligence* (2022). En revisión.
- [57] Nitish Srivastava *et al.* Dropout: a simple way to prevent neural networks from overfitting. En: *The journal of machine learning research* 15.1 (2014), págs. 1929-1958.
- [58] Connor Shorten y Taghi M Khoshgoftaar. A survey on image data augmentation for deep learning. En: *Journal of big data* 6.1 (2019), págs. 1-48.
- [59] Karl Weiss, Taghi M Khoshgoftaar y DingDing Wang. A survey of transfer learning. En: *Journal of Big data* 3.1 (2016), págs. 1-40.
- [60] HSL colour space. URL: <https://www.pythontutorials.co.uk/computer-science/colour/hsl-colour/>.
- [61] Yann LeCun *et al.* Backpropagation applied to handwritten zip code recognition. En: *Neural computation* 1.4 (1989), págs. 541-551.
- [62] Mingxing Tan y Quoc Le. «Efficientnet: Rethinking model scaling for convolutional neural networks». En: *International conference on machine learning*. PMLR. 2019, págs. 6105-6114.
- [63] Kaiming He *et al.* «Deep residual learning for image recognition». En: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, págs. 770-778.
- [64] Karen Simonyan y Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. 2014. DOI: 10.48550/ARXIV.1409.1556. URL: <https://arxiv.org/abs/1409.1556>.
- [65] Sergey Ioffe y Christian Szegedy. «Batch normalization: Accelerating deep network training by reducing internal covariate shift». En: *International conference on machine learning*. PMLR. 2015, págs. 448-456.

- [66] Okan Kopuklu *et al.* «Resource efficient 3d convolutional neural networks». En: *Proceedings of the IEEE/CVF International Conference on Computer Vision Workshops*. 2019, págs. 0–0.
- [67] Ben Graham. Sparse 3D convolutional neural networks. 2015. DOI: 10.48550/ARXIV.1505.02890. URL: <https://arxiv.org/abs/1505.02890>.
- [68] Laura Domíne, Kazuhiro Terao, DeepLearnPhysics Collaboration *et al.* Scalable deep convolutional neural networks for sparse, locally dense liquid argon time projection chamber data. En: *Physical Review D* 102.1 (2020), pág. 012005.
- [69] Salario medio ingeniero informático. URL: <https://es.talent.com/salary?job=ingeniero+de+software> (visitado 01-07-2022).