UNIVERSIDAD DE SANTIAGO DE CHILE

FACULTAD DE CIENCIA

Departamento de Física

Uso de técnicas de Machine Learning para detectar irregularidades en la rotación de púlsares de rayos gamma

Angel Alfonso Barra Muñoz

Profesor Guía:
Cristóbal Espinoza

Proyecto de Título para optar al Título Profesional de Ingeniero Físico

Santiago - Chile

2024

Resumen

En esta tesis se presenta la aplicación de redes neuronales convolucionales (CNN) para la detección automática de *glitches* en faseogramas de púlsares de rayos gamma. El objetivo principal fue optimizar la detección de estos eventos, que indican irregularidades en la rotación de los púlsares. Durante el desarrollo del trabajo, se entrenó un modelo utilizando un conjunto de datos simulados de 35,000 faseogramas, alcanzando una precisión del 98.2 % en la identificación de *glitches* con variaciones en la frecuencia del orden de 10^{-9} Hz, Posteriormente, al ampliar el conjunto de datos a 175,000 faseogramas, el modelo obtuvo una precisión del 92.1 % para *glitches* de menor magnitud (10^{-10} Hz), identificando correctamente el 88.4 % de estos en un conjunto de prueba de 25,000 faseogramas.

Agradecimientos

Esta tesis fue apoyada por el proyecto ANID FONDEQUIP EQM230160.

Tabla de Contenido

Índice de Tablas

Índice de Ilustraciones

Capítulo 1: Introducción

Motivación y planteamiento del problema

La inteligencia artificial (IA) se ha implementado en diversas áreas, como la medicina, la educación, la industria y la astronomía. Sus usos van desde la automatización de tareas repetitivas hasta el análisis de grandes volúmenes de datos, facilitando la toma de decisiones en tiempo real. En astronomía, la IA ha sido útil para el análisis de grandes cantidades de datos generados por observatorios, como imágenes, espectros y señales de radio. Estas herramientas son útiles en el descubrimiento de patrones, la clasificación de objetos celestes y la predicción de eventos cósmicos. Además ayudan a detectar anomalías, como exo planetas o fenómenos transitorios, que serían difíciles de identificar manualmente debido al volumen de información.

Entre sus principales beneficios se destacan el ahorro de tiempo y costos, la reducción de errores humanos y la capacidad para personalizar experiencias a nivel individual, como ocurre en plataformas de educación y comercio electrónico. Además, la IA impulsa innovaciones tecnológicas en áreas como la conducción autónoma, el reconocimiento de voz y la ciber seguridad, contribuyendo así a mejorar la calidad de vida y la competitividad de las empresa en un entorno digital en constante evolución.

Los púlsares son estrellas de neutrones en rápida rotación que emiten radiación electromagnética en intervalos regulares. Se forman a partir del colapso de estrellas masivas después de una supernova. Su fuerte campo magnético canaliza partículas cargadas, que emiten radiación en haces que, al ser interceptados desde la Tierra, aparecen como pulsos periódicos. Estas señales estables permiten a los astroónomos investigar fenómenos extremos, como la gravitación y la física nuclear.

Estos exhiben dos tipos principales de irregularidades en su comportamiento rotacional: el ruido rojo (*timing noise*) y los *glitches*. Estos últimos son notables por presentarse como aumentos súbitos en la frecuencia de rotación del púlsar, con magnitudes que pueden variar desde 10^{-5} hasta 10^{2} microhertzios.

La detección de *glitches* generalmente se realiza mediante el análisis de la emisión en ondas de radio de los púlsares. Sin embargo, algunos emiten tanto en ondas de radio como en rayos gamma, y un número reducido lo hace exclusiva-

mente en estos últimos. Detectar *glitches* en rayos gamma es un desafío, ya que requiere la revisión visual de faseogramas, que son histogramas bidimensionales de fase y tiempo. Estos gráficos permiten observar el comportamiento rotacional del púlsar en ventanas temporales específicas, consideradas aquí en periodos de dos años. Las pequeñas desviaciones en los faseogramas pueden indicar la presencia de un *glitch*, pero su sutileza dificulta su detección. Por ello, una herramienta basada en aprendizaje automático se presenta como una solución adecuada para optimizar y automatizar este proceso.

Para generar un faseograma, se requiere información derivada del modelado de la rotación del púlsar y de la forma de su pulso. Si durante la ventana observacional ocurre un *glitch*, este se manifestará como irregularidades en el faseograma, que pueden ir desde cambios abruptos hasta desplazamientos sutiles.

Esta tesis propone una metodología para la detección automatizada de *glitches* en púlsares de rayos gamma utilizando técnicas de aprendizaje automático. Se simularán 600,000 faseogramas etiquetados, cada uno clasificado según la presencia o ausencia de *glitches*. Utilizando redes neuronales convolucionales (CNN), el modelo estimará la probabilidad de que este fenómeno haya afectado la rotación del púlsar.

Objetivos

Objetivo general

Mediante aprendizaje automático, confeccionar un modelo capaz de identificar patrones en faseogramas que indiquen la presencia de un *glitch*, estimando la probabilidad de que este fenómeno haya afectado la rotación. Esto mediante la construcción de una CNN y ajustes conocido como *fine tuning* o ajuste fino.

Objetivos específicos

- Simular faseogramas de púlsares de rayos gamma, clasificándolos según la presencia o ausencia de *glitches*.
- Desarrollar y optimizar una red neuronal convolucional (CNN) para detectar *glitches* en un intervalo de observación de dos años.
- Aplicar un proceso de ajuste fino al modelo para mejorar la detección de *glitches* de menor magnitud.
- Validar la efectividad del modelo utilizando datos reales de púlsares en rayos gamma.

Metodología de trabajo

Este trabajo propone la aplicación de técnicas de aprendizaje automático, con un enfoque en redes neuronales convolucionales (CNN), para analizar faseogramas a partir de la fase y el tiempo del pulso característico de los púlsares.

- Desarrollar un algotimo en Python para simular la rotación de púlsares de rayos gamma, registrando el tiempo y la fase del pulso para generar faseogramas.
- Entrenar un modelo de CNN con los datos simulados, evaluando y optimizando su rendimiento en la detección de *glitches*.
- Refinar el modelo entrenado con *glitches* de variaciones del orden de 10^{-9} hertzios. aplicando técnicas de ajuste fino para mejorar su capacidad de detectar *glitches* cuya variación es del orden de 10^{-10} hertzios.
- Aplicar el modelo optimizado a faseogramas reales de púlsares en rayos gamma, tanto en escenarios con *glitches* como sin ellos.

Resultados

En esta tesis se presenta la aplicación de redes neuronales convolucionales (CNN) para la detección automática de *glitches* en faseogramas de púlsares de rayos gamma. El objetivo principal fue optimizar la detección de estos eventos, que indican irregularidades en la rotación de los púlsares. Durante el desarrollo del trabajo, se entrenó un modelo utilizando un conjunto de datos simulados de 35,000 faseogramas, alcanzando una precisión del 98.2 % en la identificación de *glitches* con variaciones en la frecuencia del orden de 10^{-9} Hz, Posteriormente, al ampliar el conjunto de datos a 175,000 faseogramas, el modelo obtuvo una precisión del 92.1 % para *glitches* de menor magnitud (10^{-10} Hz), identificando correctamente el 88.4 % de estos en un conjunto de prueba de 25,000 faseogramas.

Capítulo 2: Sobre Aprendizaje Automático

2.1. Machine Learning y Redes Neuronales Convolucionales (CNN)

Machine Learning es una rama de la Inteligencia Artificial (IA) que utiliza algoritmos para permitir que la IA imite el aprendizaje humano, mejorando su precisión a medida que recibe más datos y se entrena en múltiples iteraciones. Esto significa que, a medida que el modelo recibe más datos y se entrena durante varias iteraciones, ajusta sus parámetros internos para reducir errores y hacer predicciones más precisas. [1]

Dentro del Machine Learning, existen las Redes Neuronales Convolucionales (CNN) las cuales son un tipo de modelo que se utiliza especialmente para el análisis de datos que tienen una estructura de cuadrícula, como las imágenes. A diferencia de las redes neuronales tradicionales, las CNN son capaces de aprovechar la estructura espacial de los datos visuales, lo que les permite identificar y aprender patrones específicos dentro de una imagen, como bordes, texturas y formas complejas. Estas redes aplican filtros a las imágenes, lo que les permite extraer características relevantes y aprender representaciones más abstractas a medida que los datos atraviesan varias capas de la red. [2]

El procesamiento de aprendizaje de una CNN se realiza en varias etapas llamadas épocas. Durante una época, todo el conjunto de datos de entrenamiento pasa por la red, y los parámetros del modelo se ajustan para minimizar el error en las predicciones. Este ajuste se hace iterativamente, mejorando la capacidad del modelo para reconocer patrones visuales y hacer predicciones más precisas. Sin embargo, entrenar por demasiadas épocas puede llevar al sobre ajuste, donde la red aprende demasiado bien los datos de entrenamiento y pierde capacidad para generalizar a nuevos datos.

Las CNN han demostrado ser altamente efectivas en tareas de procesamiento de imágenes, como la clasificación de imágenes y la detección de objetos. Estas redes se han utilizado exitosamente en las competiciones como ImageNet, donde han superado a otros métodos tradicionales en la clasificación de imágenes a gran escala. Su capacidad para extraer automáticamente características relevantes y combinarlas en múltiples niveles ha hecho que las CNN sean una de las herramientas más potentes en el campo del reconocimiento de imágenes. [3]

Pesos y Sesgos

Los pesos o *weights* son parámetros de la red neuronal que se ajustan durante el proceso de entrenamiento. Representan la fuerza de la conexión entre las neuronas y determinan la importancia de las características de entrada. Cada conexión entre neuronas tiene un peso asociado que se ajusta para minimizar el error del modelo.

Por otra parte, los sesgos o *biases* son parámetros adicionales en una red neuronal que permiten desplazar la función de activación, lo que facilita el ajuste del modelo para mejor representación de los datos de entrada. A diferencia de los pesos, los sesgos no están conectados directamente a ninguna entrada específica, sino que son valores constantes añadidos a las salidas de las neuronas. [4]

Funciones de Activación

Las funciones de activación son componentes esenciales en las redes neuronales convolucionales (CNN), ya que introducen no linealidades en el modelo. Esto permite que la red neuronal pueda aprender y modelar relaciones complejas entre las entradas y las salidas, lo que es fundamental para resolver problemas que no pueden abordarse de manera efectiva con funciones lineales.

Existen diferentes funciones de activación, como *sigmoid*, *tanh*, *rectified linear unit (ReLU)* y *softmax*. La elección de la función de activación depende del tipo de problema que se esté resolviendo con la CNN o de la capa específica del modelo en la que se esté utilizando.

Por ejemplo, la función *ReLU* se usa generalmente en las capas internas de la red. Esta función permite que los valores positivos pasen sin cambios y convierte los valores negativos en cero, lo que facilita que el modelo aprenda de manera más eficiente y reduce problemas como el desvanecimiento del gradiente.

La función *sigmoid*, por otro lado, es útil en la capa final de una red cuando se necesita una salida que represente una probabilidad entre 0 y 1, lo que la hace adecuada para problemas de clasificación binaria.

La función *tanh* también se utiliza ocasionalmente en las capas internas, y su valor de salida oscila entre -1 y 1, lo que puede ser beneficioso en situaciones donde se necesite que los datos tengan un comportamiento más centrado.

Por último, la función *softmax* es común en la capa final de redes diseñadas para tareas de clasificación múltiple. Esta función convierte los valores de salida en probabilidades que suman 1, permitiendo interpretar los resultados como las probabilidades de pertenencia de la entrada a cada una de las clases posibles.

[5]

Capa de entrada:

La capa de entrada es la primera capa que recibe los datos. Aquí no se produce ningún cálculo complejo, aquí se introducen los datos en la red. En el caso de imágenes, la capa de entrada suele tener dimensiones que corresponden a las dimensiones de las imágenes, por ejemplo, 224x224x3, para imágenes RGB.

Previo a la capa de entrada se preprocesa el set de datos con el fin de normalizar y re dimensionar los datos, con el fin de que los datos estén en el rango adecuado y sean consistentes en tamaño. Ejemplos comunes de las dimensiones de las imágenes son 224x224, 128x128 y similares. [6]

Capas Convolucionales:

Estas capas aplican filtros (*kernels*) aprendibles a las entradas para producir mapas de activación. Cada filtro se desliza a través de la imagen de entrada y realiza un producto escalar entre los pesos del filtro y las regiones conectadas de la imagen de entrada, generado así un mapa de características que resalta características específicas de la imagen. Luego, se aplican funciones de activación en las capas convolucionales para introducir no linealidades, lo que permite a la red aprender representaciones más complejas obteniendo así un mapa de activación.

£Qué es un *kernel?* Un *kernel* es un filtro de dimensiones mucho menores a las de entrada. Por ejemplo, sí la imagen de entrada es de 224x224 píxeles, un *kernel* podría ser de 3x3 o 5x5 píxeles.

Este *kernel* se "*desliza*"sobre la imagen realizando operaciones de convolución, multiplicando sus valores por los de la región correspondiente de la imagen y sumando los resultados para generar un valor en el mapa de características o *feature map*.

La utilidad de los *kernel* consiste en que estos permiten a la red detectar patrones locales en la imagen. A medida que avanzan por las capas, los *kernel* combinan estas características locales para detectar patrones más complejos. [6]

Capas de Pooling:

Las capas de *pooling*, como el *max-pooling*, son utilizadas en redes neuronales para reducir la dimensionalidad de los mapas de activación, lo que disminuye la cantidad de parámetros y la carga computacional en la red. Este proceso consiste

en tomar el valor máximo de pequeñas regiones de la entrada, lo que también introduce invarianza ante pequeñas traslaciones o variaciones en los datos.

Existen diferentes tipos de *pooling* con distintas funciones. El *max-pooling* selecciona el valor máximo en cada ventana de la matriz de entrada, lo que es eficaz para capturar las características más prominentes de los datos, además de ser robusto frente a pequeñas variaciones o traslaciones. Por otro lado, el *average pooling* calcula el promedio de los valores en cada ventana de la matriz, lo que le permite retener más información de contexto en comparación con el *max-pooling*, aunque tiende a suavizar características importantes. El *global pooling* tiene dos variantes: *Global Max-Pooling* y *Global Average Pooling*. A diferencia de los métodos anteriores, estos realizan la operación sobre toda la matriz de características, no solo sobre pequeñas regiones, reduciendo el mapa a un solo valor por canal.

Los *pooling* no globales, como el *max-pooling* y el *average pooling*, se aplican principalmente en las capas intermedias de la red para reducir la dimensionalidad de los datos. En contraste, los *global pooling* se aplican antes de las capas completamente conectadas, preparando los datos para la clasificación final.

Aunque tanto los *kernels* como las operaciones de *pooling* recorren el *feature map*, la principal diferencia radica en su función. Los *kernels* se encargan de generar nuevos mapas de características al resaltar patrones específicos, como bordes o texturas. Por su parte, el *pooling* simplifica los mapas ya existentes, reduciendo su tamaño y manteniendo las características más importantes. En resumen, mientras los *kernels* extraen nuevas características, el *pooling* reduce y simplifica los datos procesados. [6]

Capas Totalmente Conectadas

También conocidas como capas densas o *fully connected layers*, son un componente fundamental en las redes neuronales. Estas capas suelen encontrarse al final de una red neuronal convolucional (CNN) y poseen varias características importantes.

Una de sus principales propiedades es la **conexión completa**: cada neurona en una capa densa está conectada a todas las neuronas de la capa anterior. Esta arquitectura permite que la red combine todas las características aprendidas en las capas anteriores para tomar decisiones finales.

Además, una capa totalmente conectada realiza una **transformación lineal** de sus entradas, lo que se expresa matemáticamente de la siguiente manera:

$$y = g(\sum_{i} w_i x_i + b) \tag{2.1}$$

Por último, estas capas permiten la **integración de características**. Después de pasar por las capas convolucionales y de *pooling*, la imagen se encuentra representada en un formato altamente abstracto. La capa totalmente conectada utiliza esta representación para clasificar la imagen o realizar otras tareas relacionadas con la toma de decisiones finales. [6]

Parámetros de entrenamiento.

Al entrenar un modelo de Machine Learning, es esencial dividir los datos en diferentes subconjuntos, comúnmente en entrenamiento, validación y prueba. El conjunto de entrenamiento permite al modelo identificar patrones e información relevante que serán útiles para realizar predicciones. Durante el proceso de entrenamiento, se emplea el conjunto de validación, que contiene datos no vistos previamente por el modelo. Esta fase es crucial ya que permite evaluar lo aprendido y ajustar los hiper parámetros, mejorando así el rendimiento del modelo sin comprometer su capacidad de generalización. Una vez completado el entrenamiento, el modelo se evalúa utilizando el conjunto de prueba, que también contiene datos no antes vistos, proporcionando una estimación precisa de su capacidad de predicción y asegurando su eficacia en condiciones reales.

La distribución de los datos entre estos subconjuntos suele ser desigual, con una proporción mayor destinada al entrenamiento, como un 75 %, mientras que el 12.5 % restante se asigna tanto a la validación como a la prueba. Este balance permite que el modelo cuente con suficientes datos para aprender, mientras se reserva una parte significativa para su evaluación.

Otro aspecto clave en el proceso de entrenamiento es el tamaño del lote, conocido como *batch size*. Este hiper parámetro define cuántas muestras se procesan antes de actualizar los parámetros del modelo, afectando directamente tanto la velocidad como la estabilidad del entrenamiento. Comúnmente, los valores seleccionados para el *batch size* son potencias de 2, como 16, 32 o 64, ya que ofrecen un buen equilibrio entre rendimiento y eficiencia computacional.

La tasa de aprendizaje, o *learning rate*, es otro hiper parámetro fundamental que determina la magnitud de los ajustes en los pesos del modelo durante cada iteración. Una tasa alta puede acelerar el proceso de entrenamiento, pero corre el riesgo de generar un modelo inestable; en contraste, una tasa baja asegura una convergencia más estable, aunque a un ritmo más lento. [7]

Además, la normalización de datos se presenta como una técnica esencial de pre procesamiento, ajustando los valores de las características de entrada para que se ubiquen en un rango común, típicamente entre 0 y 1, o con una media de 0 y una desviación estándar de 1. Este procedimiento facilita el entrenamien-

to, asegurando que todas las características influyan de manera equitativa en el modelo.

Para evitar el sobre ajuste, se emplea la regularización L2, también conocida como *weight decay*. Esta técnica penaliza los pesos grandes del modelo al añadir un término a la función de pérdida proporcional al cuadrado de la magnitud de los pesos. Este enfoque incentiva al modelo a mantener pesos más pequeños y distribuidos, mejorando así su capacidad de generalización y reduciendo el riesgo de sobre ajuste.

Finalmente, la técnica de *dropout* se utiliza en redes neuronales como método de regularización. Durante el entrenamiento, algunas neuronas se desactivan aleatoriamente, según una probabilidad definida por la tasa de *dropout*. Esto evita que el modelo se vuelva excesivamente dependiente de ciertas neuronas, forzándolo a aprender representaciones más robustas y generalizables, lo que mejora su desempeño en situaciones reales. [8]

Métricas de evaluación de modelos.

La **validación cruzada** o ***cross-validation*** es una técnica ampliamente utilizada para evaluar la capacidad de generalización de un modelo de aprendizaje automático. Esta técnica consiste en dividir el conjunto de datos en múltiples subconjuntos y entrenar el modelo varias veces, utilizando un subconjunto diferente como conjunto de prueba en cada iteración [9]. Este proceso permite obtener una evaluación más robusta del rendimiento del modelo y evita problemas asociados con una simple división de los datos en un solo conjunto de entrenamiento y prueba, reduciendo así el riesgo de sobre ajuste.

Una de las métricas más comunes para evaluar el rendimiento de un modelo de clasificación es la **exactitud** o *accuracy*, que mide la proporción de predicciones correctas realizadas por el modelo sobre el total de predicciones. Esta métrica es especialmente efectiva cuando las clases están balanceadas y se calcula como la razón entre la suma de verdaderos positivos (TP) y verdaderos negativos (TN) sobre el total de predicciones, incluyendo los falsos positivos (FP) y los falsos negativos (FN) [10]:

$$\frac{TP + TN}{TP + TN + FP + FN} \tag{2.2}$$

En este contexto, TP representa los verdaderos positivos, TN los verdaderos negativos, FP los falsos positivos, y FN los falsos negativos.

La **pérdida** o *loss* es una medida clave que evalúa la diferencia entre las predicciones del modelo y los resultados reales, proporcionando una base para ajustar los pesos del modelo durante el entrenamiento [11]. Ejemplos comunes incluyen el

error cuadrático medio (*Mean Squared Error*, MSE) para problemas de regresión y la entropía cruzada para problemas de clasificación.

La **precisión** o *precision* mide la proporción de verdaderos positivos sobre el total de predicciones positivas (TP + FP). Esta métrica es particularmente importante cuando el costo de los falsos positivos es alto, ya que refleja la exactitud de las predicciones positivas del modelo [12]. Se calcula con la fórmula:

$$\frac{TP}{TP + FP} \tag{2.3}$$

El **recall** o sensibilidad, por su parte, mide la proporción de verdaderos positivos sobre el total de casos reales positivos (TP + FN), y es crucial en situaciones donde es necesario identificar la mayoría de los casos positivos, como en la detección de enfermedades [12]. Su cálculo es el siguiente:

$$\frac{TP}{TP + FN} \tag{2.4}$$

El **F1 Score** combina precisión y *recall* en una única métrica calculando la media armónica de ambas, siendo especialmente útil en escenarios con datasets desbalanceados [13]. La fórmula es:

$$\frac{2 * Precision * Recall}{Precision + Recall} \tag{2.5}$$

El **Log-Loss**, también conocido como pérdida logística o entropía cruzada, es una métrica que mide el rendimiento de modelos de clasificación con salidas predichas en forma de probabilidades entre 0 y 1. Penaliza más severamente las predicciones incorrectas con alta confianza, haciendo que un valor menor de Log-Loss indique un mejor rendimiento del modelo. [14]

La **curva ROC (Receiver Operating Characteristic)** y su **área bajo la curva (AUC)** son herramientas gráficas utilizadas para evaluar la capacidad de un modelo de discriminar entre clases. La curva ROC muestra la relación entre la tasa de verdaderos positivos y la tasa de falsos positivos a través de diferentes umbrales de clasificación, mientras que el AUC proporciona una medida cuantitativa de esta capacidad discriminativa. [15]

Por último, la **matriz de confusión** es una herramienta que descompone las predicciones de un modelo en verdaderos positivos, falsos positivos, verdaderos negativos, y falsos negativos, facilitando así una evaluación detallada del rendimiento del modelo para cada clase [10]. Esta matriz permite no solo medir la exactitud global sino también identificar y mejorar áreas específicas del modelo.

2.1.1. Transfer learning y Fine Tauning

Al utilizar modelos pre-entrenados, es importante entender dos conceptos importantes: el *transfer learning* y el *fine tuning*.

transfer learning

El *transfer learning* es una técnica útil cuando se dispone de datos insuficientes para un nuevo dominio que se desea manejar con una red neuronal. Permite aprovechar un gran conjunto de datos preexistente y transferirlo a un problema específico. Por ejemplo, aunque solo se tengan 1,000 imágenes, es posible utilizar una red neuronal convolucional (CNN) existente entrenada con grandes conjuntos de datos.

El proceso de *transfer learning* comienza eliminando la capa de salida original de la red, que se utiliza para hacer predicciones, y reemplazándola con una nueva capa de salida específica para la nueva tarea. Esta nueva capa determina cómo el entrenamiento penaliza las desviaciones entre los datos etiquetados y las predicciones.

Luego, se entrena el modelo actualizado utilizando el conjunto de datos más pequeño, ya sea en toda la red neuronal, en las últimas capas o solo en la nueva capa de salida. Aplicando estas técnicas de *transfer learning*, el nuevo resultado de la CNN será la identificación en la nueva tarea. [16]

fine tuning

El *fine tuning* o ajuste fino es un proceso dentro de la transferencia de aprendizaje en el que se adapta un modelo de inteligencia artificial pre entrenado para tareas específicas mediante un entrenamiento adicional con un conjunto de datos más pequeño y específico. Este proceso aprovecha el conocimiento general del modelo pre entrenado, lo que reduce la cantidad de datos y potencial computacional necesarios en comparación con el entrenamiento desde cero. El ajuste fino es particularmente útil en áreas como el proceso de lenguaje natural y visión por computadora.

Existen tres maneras de aplicar esta técnica, las cuales consisten en **ajuste completo** del modelo, donde se re entrena el modelo con un nuevo conjutno de datos. Mientras que, por otra parte, está el **ajuste parcial** donde solo se re entrenan las últimas capas del modelo, manteniendo las capas iniciales congeladas sin cambiar sus pesos. Por último, existe el **ajuste eficiente de parámetros**, donde se re entrenan solo ciertos parámetros seleccionados, lo que puede llegar a ser más

eficiente en términos de recursos. [17]

Con estos métodos es posible personalizar modelos pre entrenados para tareas específicas, mejorando su rendimiento sin necesidad de grandes cantidades de datos o tiempo de computación.

Las CNN se pueden trabajar en múltiples lenguajes de programación, entre estos tenemos Java, MATLAB, C++, Julia, R, Python, entre otros. SI nos referimos a Python también tenemos multiples librerías las cuales permiten el trabajo con CNN, estas son TensorFlow, Keras, Theano, MXNet, Caffe, CNTK, Chainer y PyTorch. Esta última corresponde a la librería de interés y a utilizar en esta tesis.

£Qué es PyTorch? Es una biblioteca de código abierto para Machine Learning, desarrollada principalmente por Facebook's AI Research lab (FAIR). PyTorch está diseñado para construir y entrenar modelos de redes neuronales de manera flexible e intuitiva, con una estructura que facilita la implementación de investigación y producción. Destaca por su enfoque dinámico, que permite construir gráficos de computación sobre la marcha, lo que facilita la depuración y el ajuste de modelos complejos, siendo especialmente popular en la investigación académica y aplicaciones de Deep Learning. [18]

Capítulo 3: Sobre Púlsares

3.1. Púlsares

Los púlsares son estrellas de neutrones altamente magnetizadas, cuyo nombre deriva del acrónimo en inglés **pulsa**ting **r**adio source. Estas estrellas emiten haces de radiación electromagnética desde sus polos magnéticos. Dado que el eje de rotación puede orientarse en cualquier dirección, la emisión de los púlsares parece púlsar, similar a un faro.

Los rayos emitidos por los polos magnéticos están altamente colimados y abarcan prácticamente todo el espectro electromagnético. Es posible detectar estos pulsos de radiación y existen técnicas avanzadas para analizarlos y determinar la frecuencia rotacional de los púlsares con gran precisión. [19]

Los púlsares emiten tanto en ondas de radio como en rayos gamma, y el estudio de estas nos permite comprender su magnetosfera y dinámica rotacional. Los rayos gamma son relativamente menos frecuentes que las ondas de radio, aunque hay casos en los que se registran ambas emisiones de un mismo púlsar.

En el caso de las ondas de radio, se integra la señal sobre varias rotaciones (durante algunos minutos) para obtener un pulso estable en un proceso conocido como *folding*.

El *folding* implica sumar múltiples períodos de las señales emitidas por un púlsar, alineándolos por su fase para crear un promedio del pulso. Este método mejora la relación señal-ruido, facilitando un análisis más detallado de las características del pulso, como la frecuencia ν y su derivada $\dot{\nu}$. El pulso resultante se utiliza para desarrollar un modelo de rotación que puede predecir la llegada de futuros pulsos con una precisión del orden de decenas de microsegundos.

En cuanto a los rayos gamma, la cantidad de datos recibida en el mismo período es menor. Esta cantidad depende del flujo de fotones, medido en 10^{-8} **ph cm**$^{-2}$ **s**$^{-1}$, y sus valores oscilan entre $0{,}1$ y 50 en su mayoría, con excepciones notables como el púlsar Vela, que tiene un flujo de fotones de $1088\ 10^{-8}$ ph cm^{-2} s^{-1} [20]. Considerando herramientas como el Fermi LAT, el cual es un telescopio de rayos gamma en órbita, cuya área efectiva de detección es de 7000 cm^2 y suponemos la cantidad de fotones recibidos en un día en segundos, el número puede variar entre los $0{,}6$ y 300 en la mayoría de los casos. Para obtener pulsos estables, es necesario acumular fotones durante meses, utilizando el mo-

delo de rotación obtenido de las observaciones de ondas de radio para asignar una fase rotacional a cada fotón.

La fase rotacional se calcula de la siguiente manera:

$$\phi(t) = \phi_0 + \nu(t - t_0) + \frac{1}{2}\dot{\nu}(t - t_0)^2 \tag{3.1}$$

Donde t_0 puede ser cualquier momento en el intervalo de tiempo y t es el instante donde queremos medir la fase $\phi(t)$.

Por otra parte, el resultado de este proceso puede visualizarse en un faseograma, que es un histograma bidimensional donde un eje representa el tiempo (generalmente medido en Modern Juilian Day o MJD) y el otro la fase rotacional del fotón entre 0 y 1. Un ejemplo de faseograma se muestra en la figura 3.1. [21]

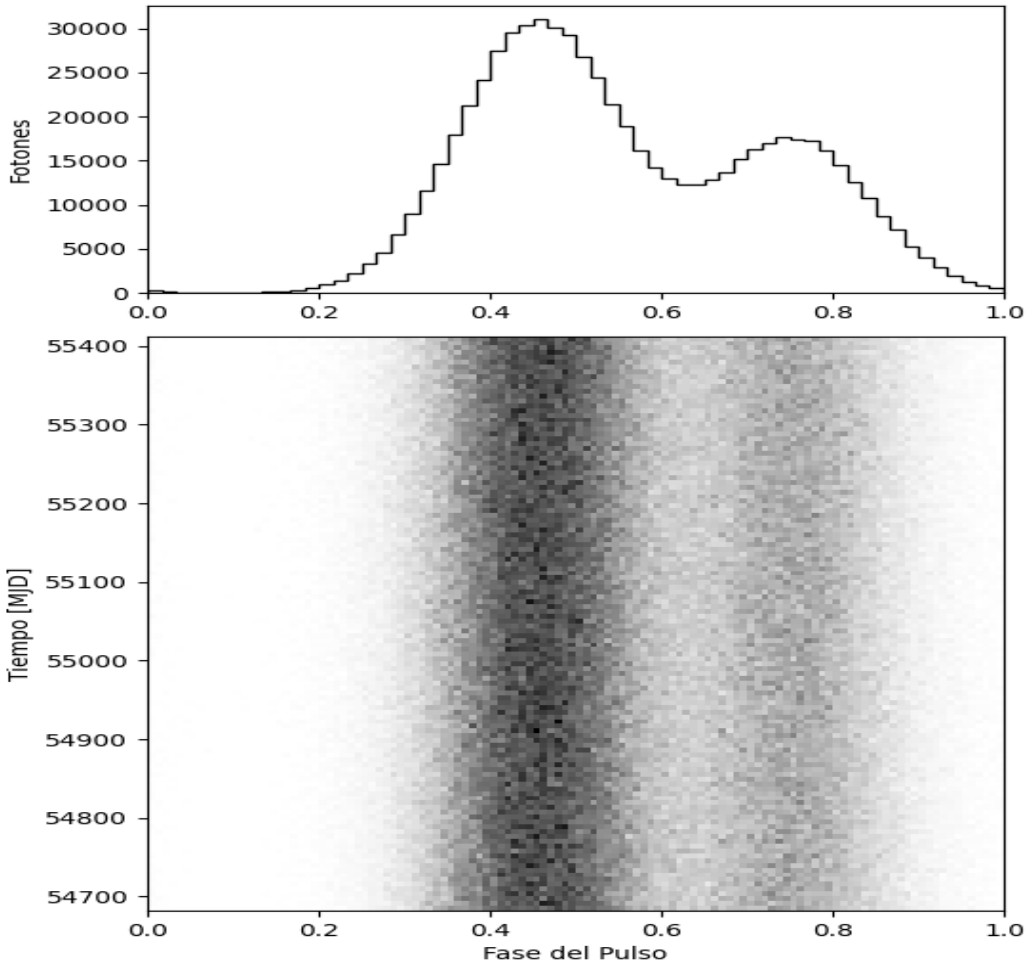

Figura 3.1: *En el gráfico superior observamos el pulso estable o característico de un púlsar y en la inferior el faseograma del mismo en un lapso de 2 años.*

En la figura 3.1 observamos cómo el pulso estable y el faseograma están directamente relacionados. Variaciones en el faseograma pueden señalar la presencia de irregularidades, como el *timing noise* y los *glitches*. Estos últimos son especialmente relevantes para esta tesis, ya que representan cambios abruptos y significativos en la rotación del púlsar, los cuales son el principal objeto de nuestro estudio, los cuales se describirán y representarán a continuación.

3.1.1. Irregularidades: *Timing Noise*

El *timing noise* se refiere a las fluctuaciones continuas y de bajo nivel en los residuos temporales de los púlsares. Los residuos temporales corresponden a la diferencia de los datos reales extraídos de la observación y las predicciones del modelo. Están caracterizados por ser suaves y lentas. Aunque este fenómeno es más prominente en púlsares jóvenes, está presente en la mayoría de púlsares. La magnitud de este fenómeno se ha demostrado estar correlacionada con los parámetros que describen la rotación de un púlsar. Este fenómeno puede visualizarse gráficamente en un faseograma, como se muestra en la figura a continuación:

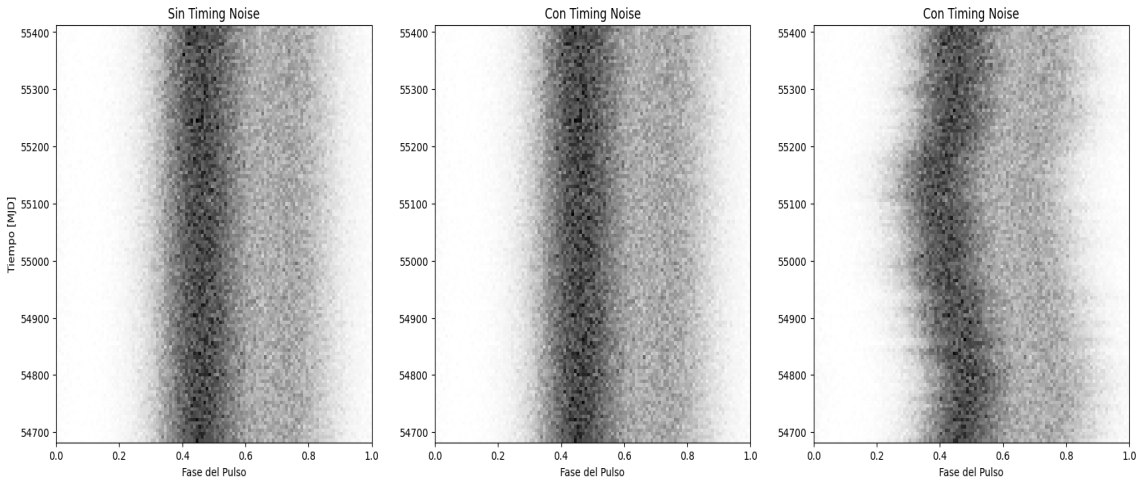

Figura 3.2: *En la imagen de la izquierda podemos ver un púlsar sin Timing Noise, mientras que la imagen del centro y a la derecha poseen timing noise.*

Como podemos observar en la figura 3.2, la presencia del *timing noise* en la rotación del púlsar, en comparación con su ausencia, revela variaciones sutiles en los faseogramas, incluso en casos es imperceptible. Estas variaciones pueden manifestarse como pequeños desplazamientos en la rotación del púlsar alrededor del *peak* del pulso estable.

Las causas exactas del *timing noise* aún son objeto de estudio, aunque su existencia está bien documentada en una amplia variedad de púlsares [21]. A pesar de no conocerse con precisión su origen, el *timing noise* se puede caracterizar como un tipo de ruido rojo", o *red noise* debido a la representación espectral que se asocia a este fenómeno. El espectro de potencia del ruido se describe mediante la siguiente expresión:

$$S(f) = (\frac{1}{f})^{\beta} \tag{3.2}$$

En esta ecuación, $S(f)$ representa la densidad espectral de potencia en función de la frecuencia f, la cual corresponde a la frecuencia en el dominio del espectro, no a la frecuencia de rotación del púlsar. Por otro lado, β es un parámetro que caracteriza el tipo de ruido. En el caso del ruido rojo o browniano, β toma el valor de 2, lo que indica que la potencia del ruido disminuye a medida que aumenta la frecuencia. [22]

Es importante aclarar que la frecuencia f en este contexto no se refiere al número de muestras (*samples*) del fenómeno observado, sino a la frecuencia asociada al análisis espectral del ruido en el dominio de Fourier. Es decir, f representa los componentes de frecuencia que describen cómo varía el ruido en función del tiempo, mientras que $S(f)$ cuantifica la potencia del ruido en dichas frecuencias.

3.1.2. Irregularidades: *Glitches*

Los *glitches* son irregularidades que corresponden a eventos de aceleración repentina en la rotación de los púlsares, esta se caracteriza por un aumento $\Delta\nu$ en la frecuencia ν, sumado a un cambio en la tasa de desaceleración $\Delta\dot{\nu}$. Estos eventos albergan una relación aún no entendida del todo entre la corteza del púlsar y el superfluido que se encuentra en su interior.

La magnitud de estas variaciones oscila entre 10^{-4} y los $50\ \mu Hz$ y la detectabilidad de estos *glitches* disminuye con la magnitud de la variación. Esto se puede apreciar en la figuras 3.3 y 3.4 donde podemos observar un faseograma sin la presencia de *glitches* y otros cuatro con estos presentes, respectivamente. Podemos observar en las figuras que, a menor magnitud la del *glitch* más difícil será de detectar, como por ejemplo, en el orden de magnitud -10 obtenemos una desviación muy menor, mientras que en el orden de magnitud -7 la forma del pulso es irreconocible post *glitch*.

El método de detección de *glitches* en emisiones de rayos gamma consiste en la observación visual realizada por investigadores, quienes examinan los faseogramas en busca de estas anomalías.

Una vez detectado un glitch, es posible compensar sus efectos en el modelo de rotación del púlsar. Para corregir el cambio abrupto en el faseograma y en el modelo de rotación causado por el glitch, se introducen ajustes específicos en la fórmula utilizada para calcular la fase rotacional. Estos ajustes se aplican después del momento t_g, que representa el instante en que ocurrió el glitch, mientras que t corresponde al insante en el que se quiere calcular $\phi(t)$. La ecuación modificada para la fase $\phi(t)$ se expresa como la siguiente:

$$\phi(t) = \phi_0 + \nu(t - t_0) + \frac{1}{2}\dot{\nu}(t - t_0)^2 + \Delta\nu(t - t_g) + \frac{1}{2}\Delta\dot{\nu}(t - t_g)^2 \tag{3.3}$$

De igual forma en esta ecuación se presentan $\Delta\nu$ y $\Delta\dot{\nu}$, valores que se mencionaron anteriormente, los cuales representan el cambio súbito en la frecuencia de rotación y su tasa de cambio, respectivamente. Este ajuste permite que el modelo continúe prediciendo la fase de manera precisa a pesar de la perturbación generada por los *glitches*. [21][23]

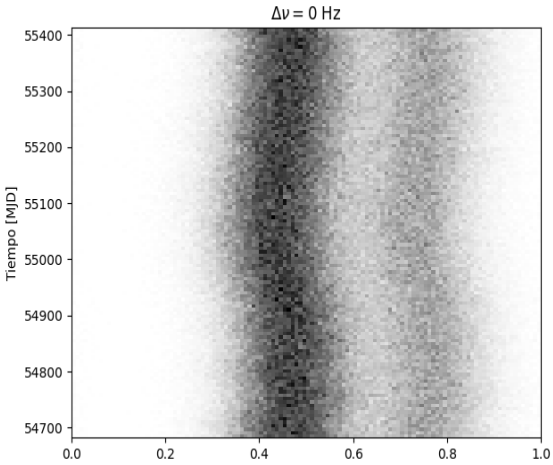

Figura 3.3: *Faseograma sin glitches.*

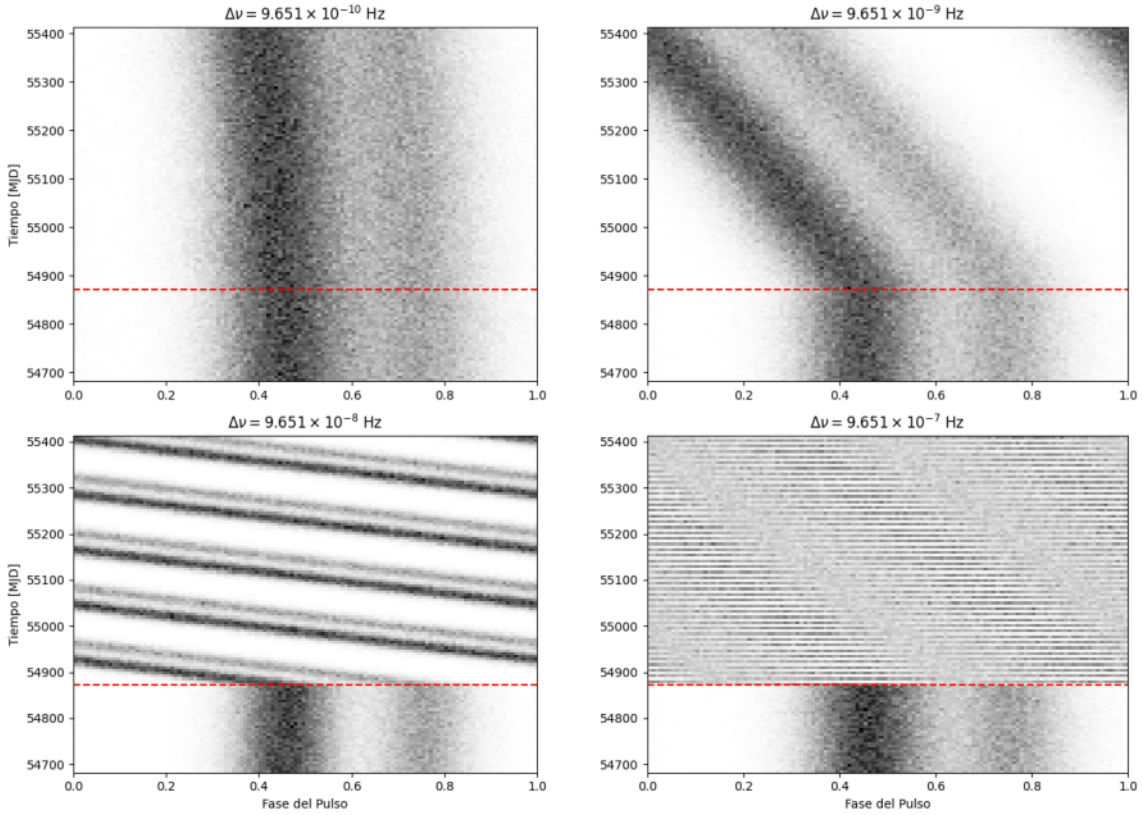

Figura 3.4: *Faseogramas del mismo púlsar con glitches de diferentes órdenes de magnitud en el día* $t_g = 54872{,}4 MJD$

3.2. Detección de *glitches* y Machine Learning actualmente.

3.2.1. Métodos de detección sin Machine Learning.

Según el artículo review *"púlsar glitches: observations and physical interpretation"* existen dos métodos automáticos de detección:

El primer método consiste en buscar la presencia de *glitches* después de cada ToA disponible, analizando un intervalo de 10-20 días alrededor de la posible irregularidad. En este enfoque, se definen los *glitches* como un cambio positivo en la frecuencia de rotación del púlsar ($\Delta\nu > 0$). Este cambio positivo en la frecuencia puede ocurrir simultáneamente con una alteración en la tasa de disminución de la frecuencia de rotación, conocida como tasa de *spindown* ($\Delta\dot{\nu} \neq 0$).

Este método emplea técnicas de ajustes de datos para identificar los posibles *glitches* en grandes volúmenes de datos de observación.

Por otra parte, tenemos el método de Markov, el cual nos permite detectar *glitches* en tiempo real. Este método define la evolución esperada de la frecuencia del púlsar tanto en presencia como en ausencia de un *glitch*. Utilizando el factor *Bayes*, se compara la probabilidad de los datos observados bajo ambos modelos. A medida que se procesan los Tiempos de Llegada (ToAs) en tiempo real, el método asigna cada dato al modelo con mayor probabilidad, determinando así si ha ocurrido o no un *glitch*. Esta comparación basada en probabilidfades permite una detección inmediata y precisa de irregularidades en la rotación del púlsar. [23]

3.2.2. Método de detección con Machine Learning.

Existe otro método de detección, este relacionado al Machine Learning. Este se puede leer a profundidad en *"Search for Glitches of Gamma-Ray púlsars with Deep Learning"*, disponible en la biblioteca en línea *arXiv*, pero que no ha sido publicado en una revista especializada.

Este artículo propone un método de CNN. Los datos extraídos del Fermi-LAT, recolectados entre el 4 de agosto de 2008 y el 3 de marzo de 2015 y se prepararon utilizando las Fermi Science Tools.

El método consiste en calcular estadísticas de H ponderadas para los fotones detectados, indicando la probabilidad de que los fotones provengan de un púlsar en específico. Estos se organizan en ventanas de tiempo y se aplica una corrección *gtbary* de Fermi tools, la cual ajusta los tiempos de llegada de los fotones

a un sistema de referencia baricéntrico, corrigiendo los efectos del movimiento de la Tierra. Las estadísticas H se maximizan sobre la tasa de disminución de frecuencia para poder identificar cambios abruptos, los cuales se asocian a los *glitches*.

La CNN se entrenó con datos generados mediante Monte Carlo, simulando pulsos con y sin *glitches*. En este artículo se logra una alta precisión para detectar *glitches* superiores a 10^{-7} Hz, con una tasa de acierto del 98 % pero disminuyendo abruptamente a un 10 % para *glitches* de magnitudes de 10^{-8} Hz. [24]

Capítulo 4: Simulación de datos y entrenamiento de modelos.

4.1. Simulando un púlsar

Actualmente, se conocen alrededor de 300 púlsares de púlsares que emiten en rayos gamma, lo cual representa menos del 10 % de los aproximadamente 3,400 púlsares conocidos en total.

Por lo que uno de los principales desafíos al preparar un modelo de Machine Learning para identificar si un púlsar ha experimentado *glitches* es la escasez de datos. Para solucionar esto, se desarrolló un *script* en Python para simular la emisión de rayos gamma y generar faseogramas realistas de estos púlsares.

Para preparar el entorno de trabajo se instalaron librerías, las cuales fueron **math**, **numpy**, **matplotlib**, **time scipy**, **random** y **colorednoise**.

4.1.1. Definiendo los parámetros de la rotación de un púlsar.

Replicando el flujo de fotones.

Para lograr replicar faseogramas lo más reales posibles necesitábamos contar con parámetros de púlsares reales o replicar su variabilidad. El primero de estos era el Flujo de Fotones o Photon Flux, el cual para replicar utilizamos datos reales del flujo de 68 púlsares [20], cuya distribución vemos en la figura 4.1. Aquí podemos ver que la mayoría de los púlsares poseen un flujo de fotones menor a 100 ph $cm^{-2}s^{-1}$ 10^{-8} y la gran parte de estos se concentra entre 0 y 50 ph $cm^{-2}s^{-1}$ 10^{-8}. Utilizamos una distribución gaussiana para replicar el flujo de fotones, limitándonos solo a valores positivos. Con esta distribución se generó la función **generate_photon_flux()**, la cual retorna un valor de la distribución ya mencionada, el cual se define como **flux**, este valor no posee su orden de magnitud, el cual es 10^{-8}.

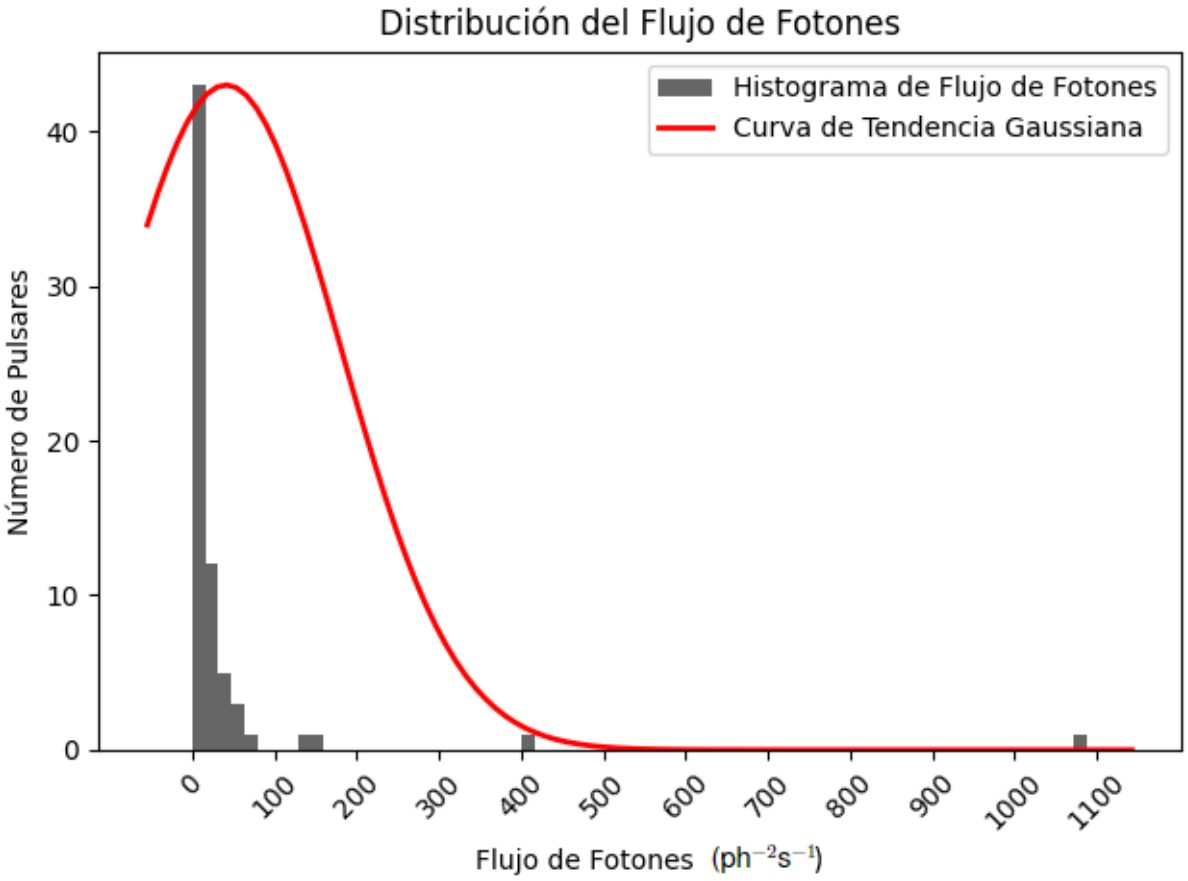

Figura 4.1: *Distribución del flujo de fotones de 68 púlsares.*

Como las unidades del flujo de fotones está en **ph cm**$^{-2}$**s**$^{-1}$ para conseguir la cantidad de fotones recibidos en un día tenemos que multiplicar el valor de flux:

$$f\_por\_dia = flux \cdot 86400s \cdot 7000cm^2 \cdot 10^{-8} \tag{4.1}$$

Donde los 7000 cm^2 representan el área efectiva de detección del Fermi LAT, esto suponiendo datos reales, los 86400 son los segundos necesarios para calcular la cantidad de fotones recibidos en un día, mientras que el último termino le agrega el orden de magnitud al flux. Con esto nosotros podemos calcular un valor aproximado de fotones recibidos en día para nuestra simulación.

Replicando la frecuencia y su derivada.

Otro parámetro necesario son la frecuencia de rotación ν y su derivada $\dot{\nu}$. A diferencia del flujo de fotones, utilizamos datos de 17 púlsares [21], de donde descargue los ephem), de los cuales se obtuvo la frecuencia y su derivada guardándolas en 2 arrays distintos, **F_0_value** y **F_1_value**, respectivamente. Se ordenaron de tal forma que el primer elemento de las derivadas, era la derivada del primer elemento de las frecuencias, por lo que se seleccionó aleatoriamente un valor de F_0_value y con ese índice obtuvimos su derivada correspondiente en F_1_value.

A partir de esto generamos la función **create_par()** que entrega F_0, F_1 de forma aleatoria y también nos proporciona el periodo de rotación P_0, el cual se calcula de la siguiente forma:

$$P_0 = \frac{1}{F_0} \tag{4.2}$$

Obteniendo así la frecuencia, su derivada y el periodo de rotación.

Replicando los *glitches*.

Como se mencionó en el marco teórico, la magnitud de los *glitches* puede variar desde los 50 hasta los 10^{-4} μHz. Como para esta tesis queremos simular y detectar *glitches* pequeños, trabajaremos con variaciones en la frecuencia $\Delta\nu$ del orden de 10^{-9} y 10^{-10} Hz, donde podemos apreciar cambios notables en el primero y cambios difíciles de ver en el segundo. Por otra parte su derivada $\Delta\dot{\nu}$ estará fija en variaciones del orden de 10^{-18} Hz s^{-1}. Para obtener ambos valores se utilizó **np.random.uniform()** y los valores para las variaciones y su derivada guardarlos como **GLF0** y **GLF1**.

Lo anterior mencionado se utiliza en la función **generate_glitch_param()**, esta se utiliza para poder calcular la variación de la frecuencia y su derivada. Para obtener una variación $\Delta\nu$ con magnitud 10^{-9} Hz los valores máximos y mínimos para **np.random.uniform()** deben ser 10^{-8} y 10^{-9} Hz respectivamente. Mientras que para que la variación sea de 10^{-10} Hz estos valores deben ser 10^{-9} y 10^{-10} Hz. Así como con $\Delta\dot{\nu}$, ya que para obtener un valor aleatorio con una magnitud 10^{-18} Hz s^{-1} debemos oscilar entre 10^{-17} y 10^{-18} Hz s^{-1} .

La salida de esta función será la variación de la frecuencia $\Delta\nu$ como **GLF0** y su derivada $\Delta\dot{\nu}$ como **GLF1.**

Replicando el pulso estable o curva característica

El pulso estable obtenido de un púlsar es característico de este y se pueden modelar con gaussianas (1 o más) o KDE [21]. Esta característica del púlsar está directamente relacionada al faseograma que buscamos generar, por lo que obtener una gran aleatoriedad en el pulso estable es crucial.

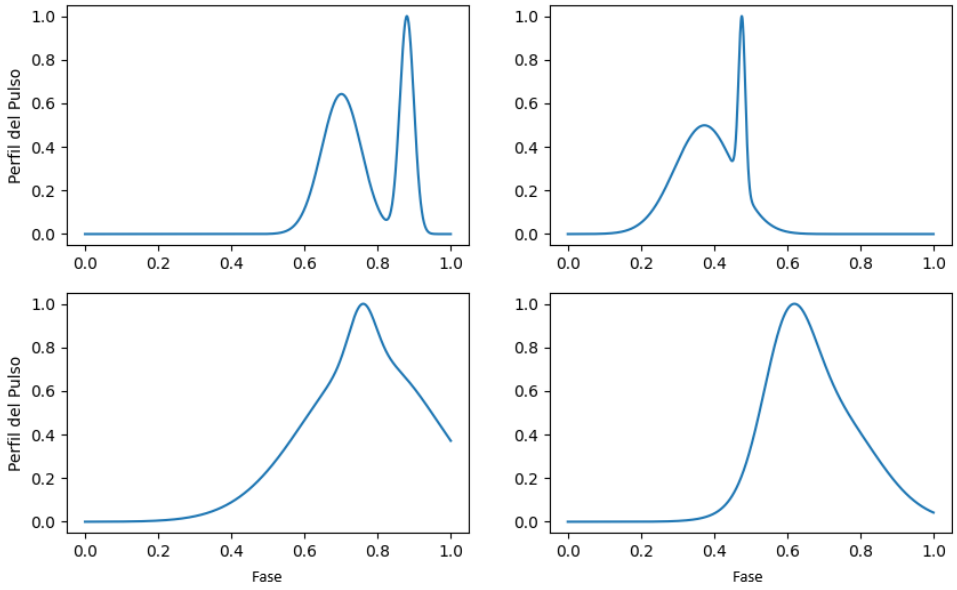

Figura 4.2: *Distintos perfiles de pulso generados aleatoriamente por la función* ***simulate_púlsar_pulse()***

Para esta tarea construimos una función llamada **simulate_púlsar_pulse()**. Esta genera de manera aleatoria entre 1 a 3 gaussianas iguales o distintas, como podemos ver en la figura 4.2. Estas gaussianas son posicionadas entre 0 y 1 con el fin de poder tener coherencia con las fases del pulso del faseograma. De esta forma obtenemos las curvas de la figura 4.2.

Con la salida, que principalmente suele escribirse de la siguiente manera: **x, y = simulate_púlsar_pulse()**, valores que corresponden a la fase y la amplitud, respectivamente.

A partir de este pulso estable y los valores **x** e **y**, podemos transformarlo en una distribución de probabilidad con la función **generar_fase(x,y)**. Esta función retorna una probabilidad aleatoria del pulso estable como **fase** y esta es la que, a la hora de simular la emisión de rayos gamma de un púlsar, se asigna al fotón,

por lo que a cada uno de estos se le asocia una fase de rotación distinta acorde a la distribución de probabilidad generada con **generar_fase(x,y)**.

4.1.2. Replicando los tiempos de llegada y fase de la emisión de rayos gamma.

Con la fase de rotación ya conocida, como vimos en la sección anterior, necesitamos también el tiempo de llegada de ese fotón, este tiempo lo calculamos de la siguiente manera:

$$t_{llegada} = t + dt \tag{4.3}$$

En la expresión anterior tenemos $t_{llegada}$ es el tiempo asociado a la fase de llegada del fotón, mientras que t es el instante inicial en el caso del primer fotón y el tiempo de llegada del fotón anterior posteriormente, definido por simplicidad acorde a las mediciones de Ray (citar), el cual corresponde a 54682 MJD, cuya fecha corresponde al 4 de agosto del año 2008. Después tenemos el valor dt, el cual se calcula de la siguiente manera:

$$dt = \frac{1}{f\_por\_dia} \tag{4.4}$$

Lo cual nos indica cada cuanto tiempo debería llegar un fotón durante un día. Por ejemplo si tenemos un flujo de fotones de **20** $\cdot$**10**$^{-8}$ **ph cm**$^{-2}$**s**$^{-1}$, en un día, de acuerdo a la ecuación 4.3, la cantidad de fotones en un día serian aproximadamente 121, que se traduce en la llegada de uno cada 714,05 segundos, lo que correspondería al dt.

A este valor de $t_{llegada}$ hay que aplicarle una corrección, por lo que el tiempo para cada fotón se vería de la siguiente manera:

$$t_{fotón} = t_{llegada} + t_{corr} \tag{4.5}$$

Donde t_{corr} es lo siguiente:

$$t_{corr} = \frac{\phi_0 - \phi}{frec(t, t_0) \cdot 86400} + \frac{\hat{\phi}}{frec(t, t_0) \cdot 86400} + backgroundnoise + rednoise \tag{4.6}$$

Donde haremos un desglose término a término. ϕ_0 representa la fase inicial de rotación, este valor puede estar entre 0 y 1. La elección para las simulaciones de esta tesis fue $\phi_0 = 0$.

ϕ proviene de la función llamada **fase(t)**, la cual se obtiene a partir de la expresión 3.1. La salida de esta función es la fase de rotación y se calcula de la siguiente

forma:

$$\phi = \phi_0 + F0 \cdot (t - t_0) \cdot 86400 + \frac{1}{2} \cdot F1 \cdot (t - t_0)^2 \cdot 86400^2 \tag{4.7}$$

Donde F0 y F1 los obtuvimos de la función **create_par()** y el producto por 86400 es necesario para que las dimensiones sean en segundos. ϕ_0 es 0, t es el $t_{llegada}$ calculado previamente y t_0 es un tiempo cualquiera dentro del intervalo, en nuestro caso ese valor es $t_0 = 55212$ MJD para todas las simulaciones.

En el caso de que queramos colocar un *glitch*, la expresión cambia a la siguiente y se utiliza la función **fase_glitch()**:

$$\begin{aligned} \phi = \phi_0 + F0 \cdot (t - t_0) \cdot 86400 + \frac{1}{2} \cdot F1 \cdot (t - t_0)^2 \cdot 86400^2 \\ + GLF0 \cdot (t - t_g) \cdot 86400 + \frac{1}{2} \cdot GLF1 \cdot (t - t_g)^2 \cdot 86400^2 \end{aligned} \tag{4.8}$$

Al igual que la expresión para el caso sin *glitches*, esta expresión está derivada de otra expresión, en este caso, de la ecuación 3.2. Aquí los parámetros GLF0 y GLF1 los obtenemos de la función **generate_glitch_param()** y el valor t_g es el tiempo donde ocurre este evento, en este caso se calcula de forma aleatoria con la función **np.random.uniform()** entre las fechas 54682 y 55230, periodo que abarca 18 meses, de la ventana total de dos años.

Debido a que la frecuencia de rotación del púlsar va variando con el tiempo, es necesario realizar un ajuste, que se define en la función $frec(t, t_0)$. Y el cálculo es el siguiente:

$$F = F0 + F1 \cdot (t - t_0) \cdot 86400 \tag{4.9}$$

Donde t y t_0 son los valores ya definidos. Con esto podemos ir ajustando la fase de rotación a cada fotón recibido simulado.

$\hat{\phi}$ corresponde al valor arrojado por **generar_fase(x,y)**. Su rol consiste en darle la forma del pulso estable a nuestro púlsar simulado.

t_g, como se definió anteriormente, este valor corresponde al tiempo en el qué ocurre el *glitch*, y lo definimos de forma aleatorio dentro de un intervalo en el tiempo de observación. Este intervalo corresponde hasta $\frac{3}{4}$ del tiempo total. La razón de esto se explica desde la figura 4.3 a la figura 4.8.

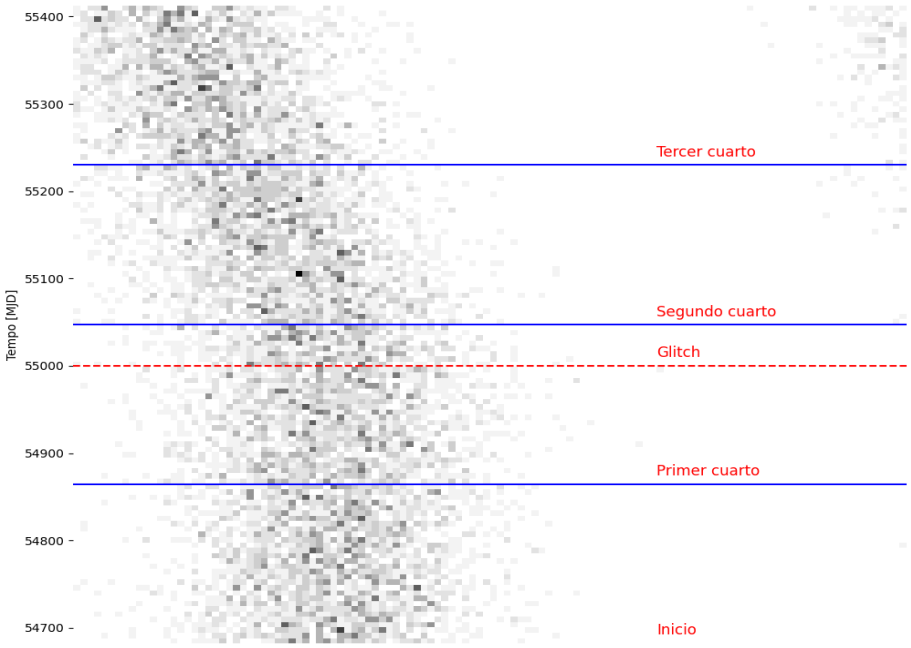

Figura 4.3: *Glitch ocurre en el* 55047 *MJD.*

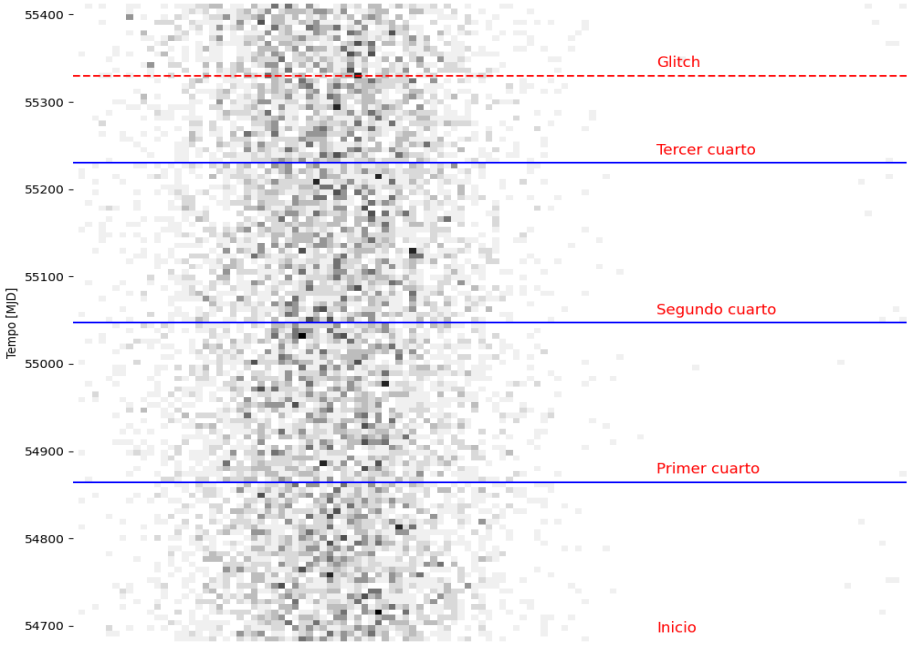

Figura 4.4: *Glitch ocurre en el* 55230 *MJD.*

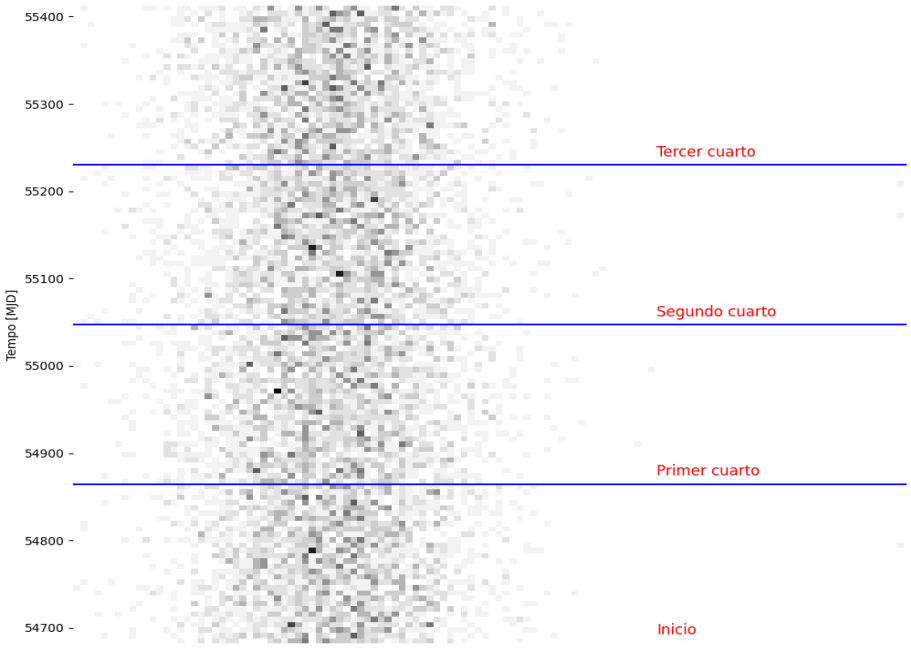

Figura 4.5: *No ocurre ningún glitch.*

Como se puede ver en la figura 4.3, el *glitch* es visible y ocurre dentro del intervalo de $\frac{3}{4}$ del tiempo que definimos. Por otra parte, en la figura 4.4 podemos ver que el *glitch* ocurre sobre este límite y si notamos mientras más arriba sucede el *glitch* más recto se ve, por lo tanto, más similar a un faseograma sin *glitches*, siendo más parecido a los *glitches* de la figura 4.13. Los faseogramas de la figura anterior fueron simulados con variaciones notorias, dígase *glitches* con variaciones del orden de 10^{-9} Hz. Por lo que si nos referimos a los irregularidades cuyo variación es del orden de 10^{-10} Hz, mientras más arriba, menos notoria sería la desviación. Esto puede presentar un problema a la hora de entrenar los modelos ya que llevaría a tener una noción confusa sobre como se ve un *glitch* a la hora del entrenamiento.

La solución propuesta, si bien facilitaría la capacidad del modelo de clasificar *glitches*, presentaría también un problema a la hora de detectar *glitches* que estén en este último cuarto de la ventana de tiempo, entonces £Cómo lo solucionamos?.

La solución reside en el último paso de nuestro trabajo, ya que, si bien el modelo estará entrenado para analizar faseogramas en intervalos de 2 años, la cantidad mínima que se podrá ingresar al código será de 3 años, esto para poder obtener dos faseogramas de dos años de datos cada uno que se solapan durante un año. Esto se ve con más claridad en la figura 4.6

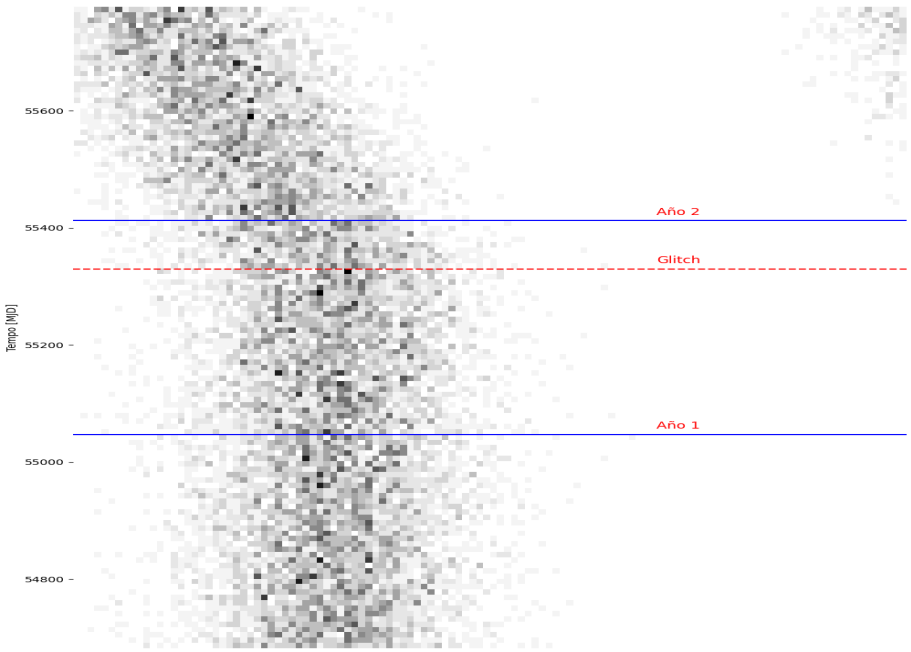

Figura 4.6: *púlsar simulado durante 3 años, desde* 54682 *MJD, hasta* 55412 *MJD, con un glitch ubicado en* 55320 *MJD.*

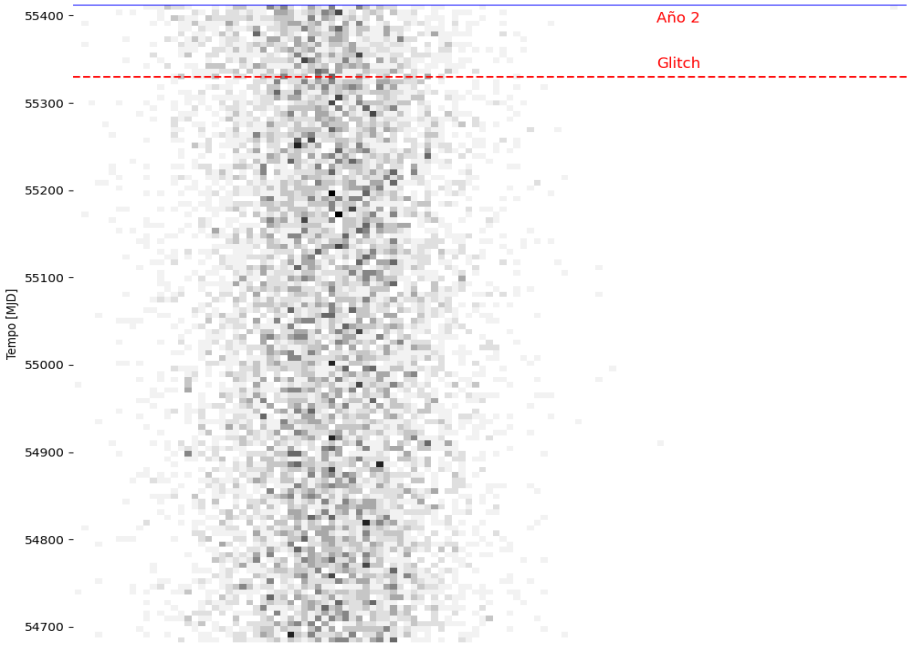

Figura 4.7: *Dos primeros años de simulación, donde podemos ver que el glitch ocurre al final de este intervalo.*

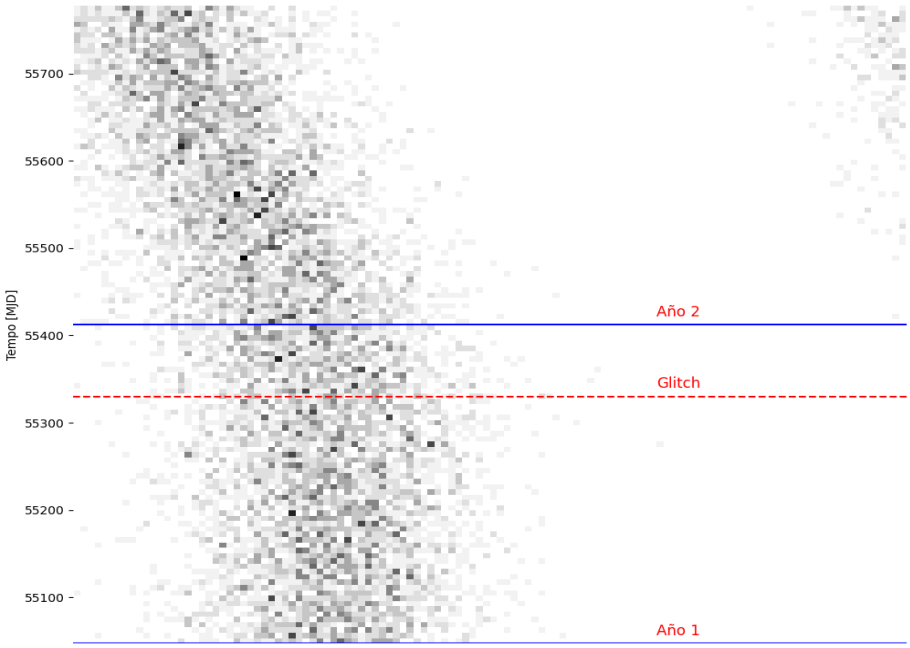

Figura 4.8: *Dos últimos años de simulación, donde podemos ver el glitch de forma clara.*

Como se ve en la figura 4.6, hasta el segundo año pareciese que no existe ningún *glitch* en nuestro faseograma, por lo que es más probable que el modelo lo asocie a un pulsar que no tuvo este tipo de irregularidad. En la figura 4.7 podemos ver como se vería el faseograma de los primeros dos años, donde la existencia del *glitch* no se nota a simple vista. Como ya se mencionó, la existencia de un faseograma así perjudicaría la capacidad de clasificación de nuestro modelo, ya que sería un *glitch* con las características de la ausencia de este.

Mientras que, si observamos los últimos dos años de la figura 4.6 podemos ver como el *glitch* comienza a pronunciarse previo al segundo año, por lo que al tener un faseograma que enseñe los últimos 2 años de este periodo, nos dará una visión más clara de que si existió o no esta irregularidad. En la figura 4.8 podemos ver el faseograma de los dos últimos años y notamos lo claro que es la ocurrencia del *glitch*, lo que hará más simple la clasificación y detección del modelo.

Ruido de fondo o background noise:

Este término se genera a partir de una distribución gaussiana centrada en 0, este también tiene una desviación estándar del 10^{-7} y se ve de la siguiente forma:

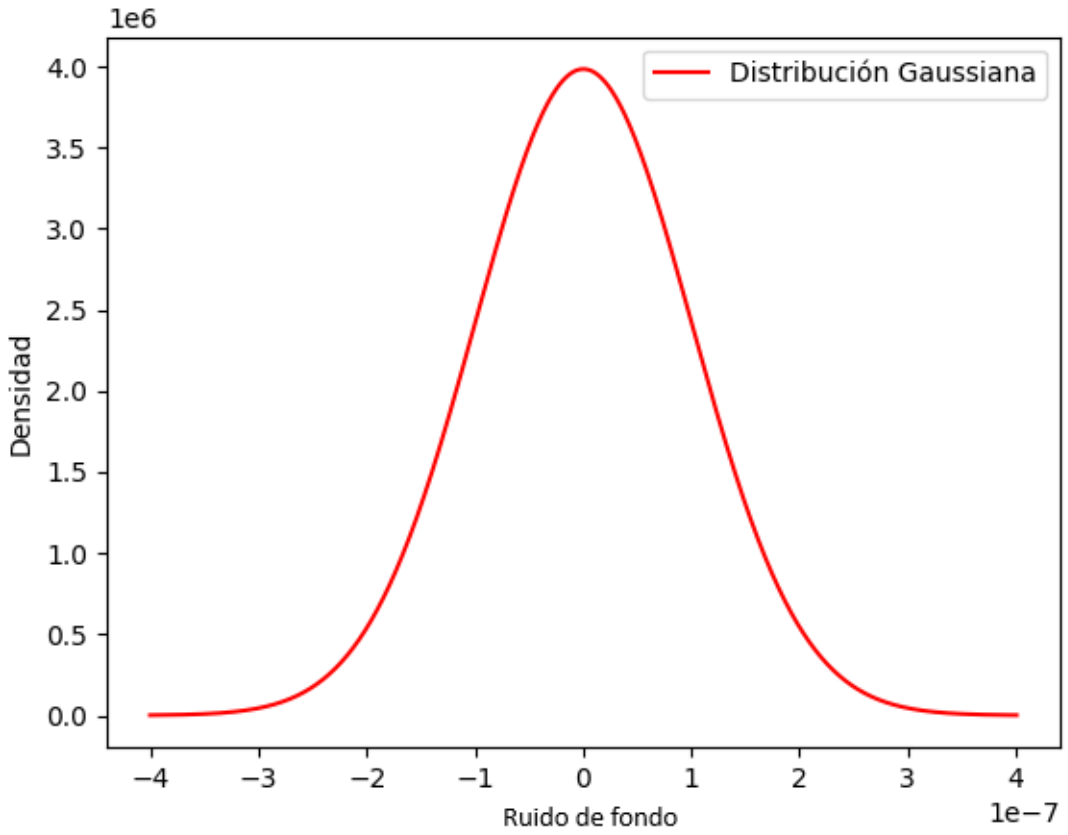

Figura 4.9: *Distribución gaussiana de la cual se obtieen el ruido de fondo, está centrada en cero y su desviación estándar de* 10^{-7}

Este ruido de fondo se guarda en una lista llamada **ruido_fondo** la cual posee una cantidad de elementos iguales a los fotones que se simularán, cuyo número es:

$$int(f\_por\_dia \cdot 365{,}25 \cdot 2) \tag{4.10}$$

ya que son la cantidad de fotones aproximada que llegarían durante dos años de observación del púlsar simulado, por lo que, a la hora de simular cada fotón, se extrae el elemento **ruido_fondo[i]** correspondiente al fotón **i**. Este valor se define en días.

Ruido rojo o *timing noise/red noise:*

Para poder agregar *timing noise* a nuestra simulación necesitamos utilizar la ecuación 3.2, esta ecuación la pudimos aplicar en python gracias a la librería colorednoise (citar github), específicamente la función **powerlaw_law_gaussian(β, samples**). al ser *timing noise* el valor de β es 2 y el número de samples es igual a la cantidad de fotones simulados para dos años. Esto se calcula de igual forma que la ecuación 4.10.

Los valores obtenidos del *timing noise* se multiplican por un valor aleatorio definido para cada púlsar a partir de una distribución gaussiana del tipo 4.9. A partir de esos valores se registran los valores en la lista **ruido_rojo**. Cuando simulamos un fotón de la emisión de un púlsar obtenemos el elemento **i** de esta lista, llamándolo **ruido_rojo[i]**.

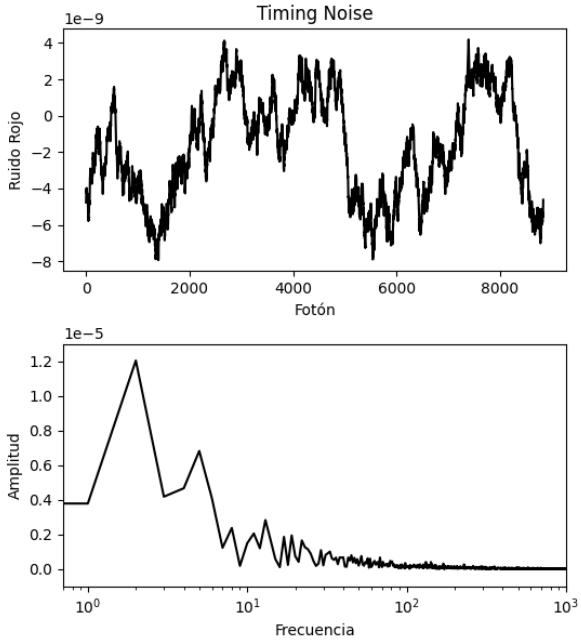

Figura 4.10: *El gráfico superior representa el timing noise según la cantidad de fotones simulados, mientras que el gráfico inferior vemos la transformada de fourier de esta serie de datos.*

4.1.3. Ejecutando la simulación

Para llevar a cabo la simulación, se decidió generar un total de 600 mil datos de púlsares. Realizar este proceso en un único *script* habría sido extremadamente lento, requiriendo aproximadamente dos meses de tiempo de ejecución. Con el objetivo de optimizar este proceso, se diseñó una estrategia que consistía en la creación de 60 *scripts*, cada uno encargado de simular 10 mil púlsares. De estos 60, 30 se dedicaron a generar púlsares con *glitches*, donde 20 *scripts* introducían variaciones en la frecuencia del orden de 10^{-9} Hz, y los 10 restantes variaciones de 10^{-10} Hz. Los otros 30 *scripts* simulaban púlsares sin *glitches*. Esta metodología permitió reducir el tiempo de simulación a 12 horas, ya que todos los *scripts* fueron ejecutados de manera simultánea mediante un *script bash*.

En cuanto el proceso realizado por cada uno de estos códigos, se puede describir de la siguiente manera. En primer lugar, se prepararon los parámetros necesarios para la simulación. Se definieron arrays con valores de flujo de fotones, los cuales se utilizaron para simular estos mismos, basándonos en medias y desviaciones estándar predefinidas. Estos valores fueron ajustados mediante una distribución KDE para representar mejor la distribución simulada. Asimismo, los parámetros del púlsar, como la frecuencia y su derivada, se seleccionaron aleatoriamente a partir de arrays predefinidos, los cuales determinaban el comportamiento del púlsar durante la simulación.

Posteriormente, se generaron los perfiles de pulso para cada púlsar, utilizando una combinación de funciones gaussianas. Dichos perfiles representan cómo varía la intensidad del pulso en función de la fase rotacional. Una vez generado el perfil, se procedió a simular el proceso de emisión de fotones. El flujo de fotones se tradujo en una cantidad de fotones recibidos por día, lo que permitió el intervalo de tiempo entre la llegada de cada fotón. Para hacer la simulación más realista, se incorporaron correcciones que incluían ruido de fondo y ruido rojo. En el caso de los púlsares con *glitches*, se añadieron efectos adicionales, modelados como variaciones abruptas en la frecuencia de rotación del púlsar. La fase de cada púlsar se calculó utilizando su frecuencia y derivada de frecuencia, aplicando correcciones adicionales en los púlsares con *glitches* para modelar los cambios abruptos en la fase debido a esta irregularidad

Finalmente, se generaron faseogramas para cada púlsar simulado. Estos faseogramas mostraban la fase de cada fotón simulado y fueron fundamentales para el posterior entrenamiento de los modelos de detección de *glitches*. Los faseogramas generados fueron almacenados para su análisis posterior.

En cuanto a la diferencia entre la simulación de púlsares con y sin *glitches*, es importante señalar que en la versión del código que no los incluye, se simula in flujo continuo de fotones sin interrupciones o irregularidades en la fase del púlsar.

En cambio, en la versión que los incluye, se simulan irregularidades en la rotación del púlsar, implementadas como modificaciones en el cálculo de la fase. Estas modificaciones implican un cambio abrupto en la frecuencia y su derivada, que ocurre en un momento aleatorio dentro de los primeros 18 meses de simulación.

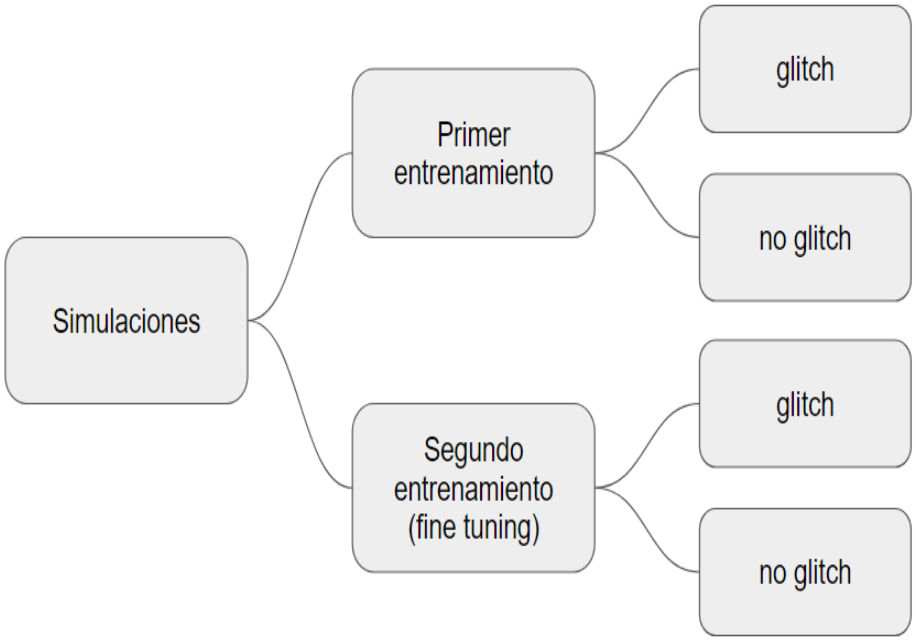

Figura 4.11: *Almacenamiento de simulaciones*

Resultados de las simulaciones.

Las imágenes resultantes deben verse con el siguiente formato:

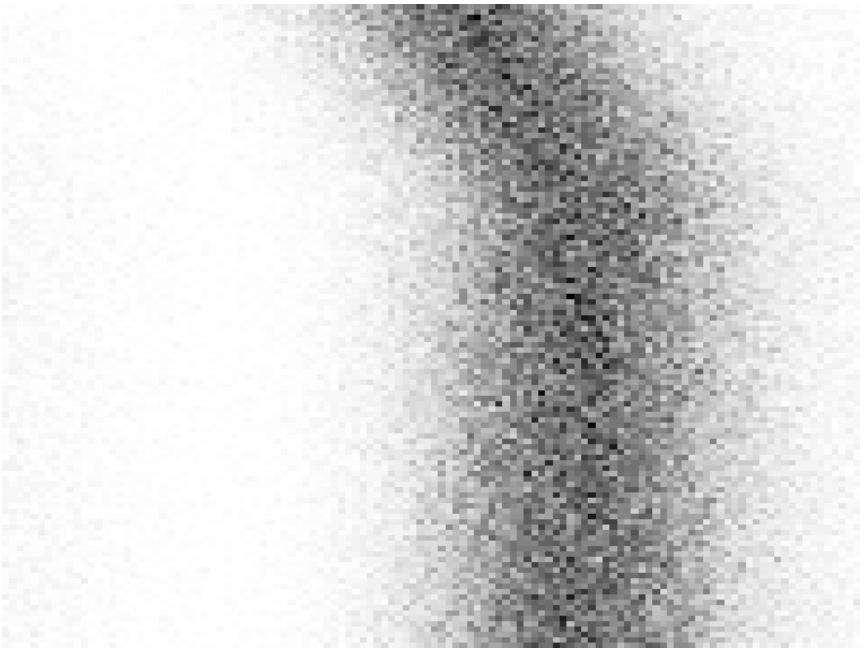

Figura 4.12: *Faseograma simulado, en el cual definimos una variación en la frecuencia ν del orden de magnitud 10^{-9} Hz.*

En la imagen anterior podemos observar que no tenemos ejes, bordes ni espacios en blanco más allá de los correspondientes al propio faseograma. Esto se debe a que a la hora de entrenar nuestros modelos de aprendizaje necesitamos la menor cantidad de información posible dejando solo la relevante para la detección o no de *glitches.*

En las siguientes figuras veremos los resultados de los 4 tipos de faseogramas que generamos.

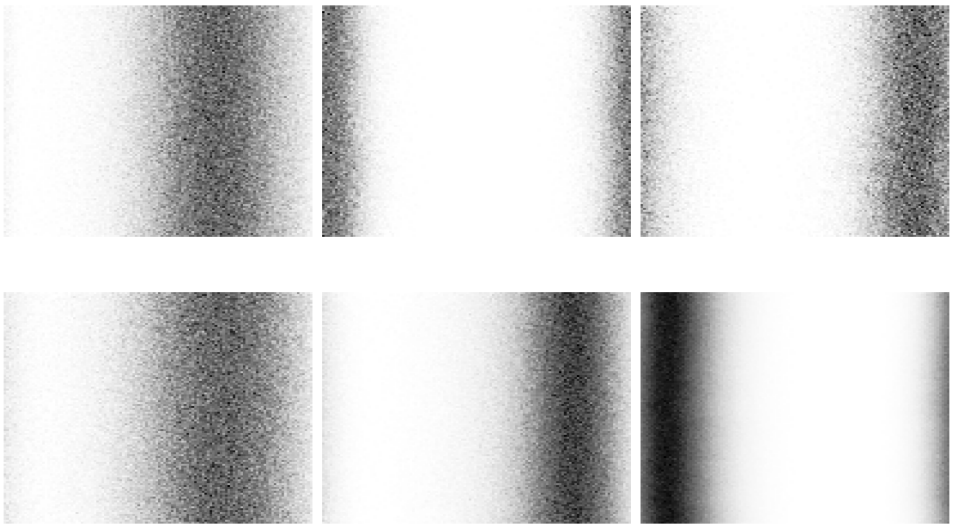

Figura 4.13: *Faseogramas simulados de púlsares sin glitches.*

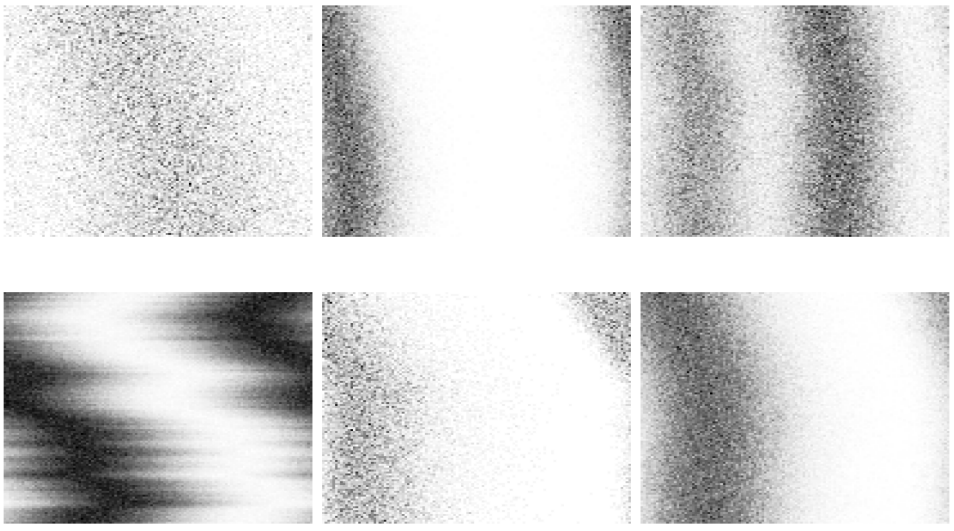

Figura 4.14: *Faseogramas simulados de púlsares con glitches cuyo aumento en la frecuencia ν es del orden 10^{-9} Hz en el primer conjunto de datos.*

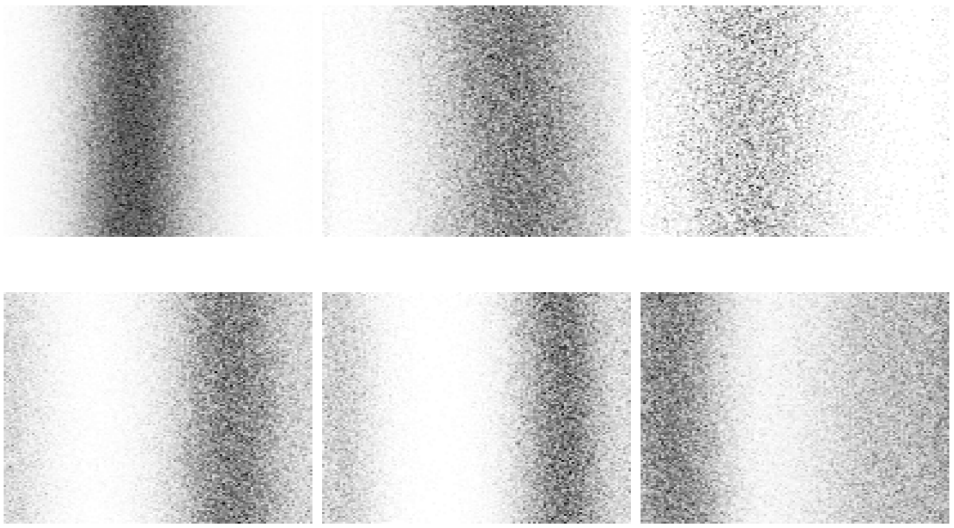

Figura 4.15: *Faseogramas simulados de púlsares con glitches cuyo aumento en la frecuencia ν es del orden 10^{-10} Hz en el segundo conjunto de datos.*

En las figuras anteriores podemos ver pequeñas muestras de los datos simulados, en las figuras 4.13 tenemos los casos sin *glitches*, los cuales no presentan desviaciones más allá del ya visto ruido rojo. Por otra parte las figuras 4.14 y 4.15 presentan desviaciones, ya que estas si poseen *glitches* de distintas magnitudes que nosotros definimos. Como podemos notar en la figura 4.15 las desviaciones son pequeñas y cuesta notarlas, por lo que no es difícil que pasen por casos sin *glitches*. Estas desviaciones pequeñas son las que queremos detectar aplicando técnicas de Machine Learning.

Organización de los directorios.

A continuación podemos observar cómo se redistribuyeron los datos de la figura 4.11 en la que podemos observar a continuación:

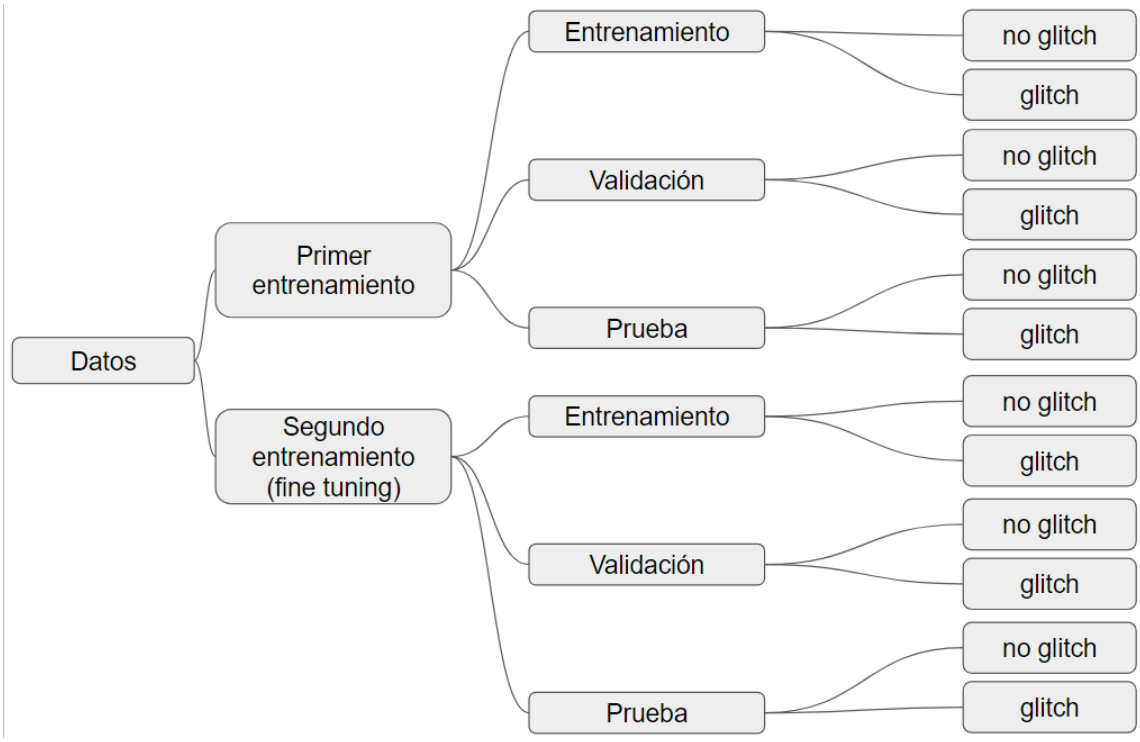

Figura 4.16: *Organización de los datos.*

La distribución de los datos entre entrenamiento, validación y prueba es la siguiente para los datos del primer modelo y el ajuste fino: 75 %, 12.5 % y 12.5 % respectivamente. Donde cada set está dividido equitativamente entre *glitches* y sin *glitches*.

Además de esos directorios se generaron dos más, el primero contenía el 10 % de los datos del entrenamiento para el primer modelo y el segundo contenía el 10 % de los datos para el ajuste fino, siguiendo la misma estructura observada en la figura 4.16.

Nuestros datos fueron subidos en archivos *.zip* al servidor donde trabajaremos. Estos en 4 carpetas, los datos del primer entrenamiento, el 10 % de estos, los datos para el ajuste fino y el 10 % de estos, cada uno con su respectivo conjunto de entrenamiento, validación y prueba.

4.2. Construcción del modelo y búsqueda de hiper parámetros.

4.2.1. Carga de los datos, ordenarlos y procesarlos para el entrenamiento.

Los datos se subieron a nuestro entorno de trabajo en cuatro carpetas; datos, datos_10, datos_fine_tuning y datos_fine_tuning_10. Donde cada uno de estos directorios poseía las carpetas entrenamiento, validación y prueba, donde cada una tenía las carpetas *glitch* y *no_glitch*, tal como se observa en la figura 4.16. Las carpetas datos_10 y datos_fine_tuning_10 poseen solo el 10 % del total de los datos, ya que estos datos se utilizarán para las pruebas y ajustar el modelo.

Las imágenes generadas por la función 'plt.hist2d()' se guardan inicialmente en formato '.png', con dimensiones 4x600x400. Para pasar estas imágenes a formato RGB, las convertimos a '.jpg', lo que reduce las dimensiones a 3x600x400, eliminando el canal alfa.

Luego, definimos un objeto 'transform' que se encarga de realizar una serie de transformaciones en los faseogramas. Primero, las imágenes se convierten de RGB a escala de grises, reduciendo los canales de color de 3 a 1. A continuación, las imágenes se re dimensionan a 1x128x128 píxeles y se transforman en tensores PyTorch con la forma 1x128x128, donde 1 representa el canal de la imagen en escala de grises.

Finalmente, los datos se normalizan para que los valores de los píxeles estén centrados alrededor de 0, con una desviación estándar de 1, para facilitar así el entrenamiento del modelo con las entradas estandarizadas.

A continuación 'transform':

```
transform = transforms.compose([
    transforms.Greyscale(),
    transforms.Resize((128, 128)),
    transforms.ToTesnor(),
    transforms.Normalize((0.5), (0.5,))
])
```

Estas transformaciones se aplicarán a todos nuestros datos, que serán definidos como 'train_dataset', 'val_dataset' y 'test_dataset'.

Para evitar que durante el entrenamiento el modelo aprenda el orden de los datos, al 'train_dataset' se definirá el parámetro 'shuffle' en 'True'.

4.2.2. Construyendo el modelo

Como nuestro trabajo consiste en el ajuste fino de un modelo, nuestro primer objetivo es generar un modelo capaz de detectar *glitches* cuya magnitud sea del orden de 10^{-9} Hz. Si observamos las imágenes de las figura 4.14, podemos notar que estas son fáciles de diferenciar. El modelo propuesto para este trabajo cuenta con un número reducido de capas, lo que lo hace adecuado para esta tarea específica. A continuación, se presenta su arquitectura:

```
class glitch_CNN(nn.Module):
    def __init__(self, dropout_rate):
        super(glitch_CNN, self).__init_()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, stride=1, padding=1)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2,padding=0)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1)
        self.fc1 = nn.Linear(64*32*32, 128)
        self.dropout = nn.Dropout(p=dropout_rate)
        self.fc2 = nn.Linear(128, 1)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1,64*32*32)
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)

        return x
```

El modelo cuenta con dos capas convolucionales. La primera, denominada 'self.conv1', tiene un único canal de entrada, dado que las imágenes han sido preprocesadas para estar en escala de grises. Esta capa genera 32 canales de salida, lo que corresponde a 32 mapas de características. Estos mapas se crean utilizando 32 kernels de 3 x 3 píxeles, con un stride de 1 y un padding de 1, lo que mantiene constante las dimensiones espaciales de la imagen de entrada.

La segunda capa convolucional, 'self.conv2', recibe 32 canales de entrada, correspondientes a los 32 mapas de características generados por la misma capa, y produce 64 canales de salida, generando así 64 nuevos mapas de características. El stride y el padding en esta capa se mantienen en 1.

Para reducir la resolución espacial de los mapas de características y, por ende, disminuir la cantidad de parámetros en las siguientes capas, se aplica una operación de Max Pooling a través de la función 'nn.MaxPool2d'. Esta capa utiliza kernels de 2 x 2 píxeles con un stride de 2 y sin padding ('padding=0'). Como resultado, las dimensiones espaciales de los mapas de características se reducen a la mitad en cada dimensión.

El modelo incluye dos capas totalmente conectadas para realizar las predicciones. La primera toma un vector de tamaño 64 x 32 x 32 como entrada y los transforma en un vector de 128 elementos. La segunda capa toma estos 128 elementos y los condensa en un único valor de salida.

Entre las capas totalmente conectadas, se incorpora una capa de dropout ('nn.Dropout') para prevenir el sobre ajuste.

Propagación hacia adelante con 'forward(self, x)'

Con la arquitectura definida en 'glitch_CNN', podemos observar como los datos fluyen a través de sus capas durante la propagación hacia adelantes.

La propagación hacia adelante del modelo sigue una secuencia lógica. Los datos de entrada, denominados 'x', tienen la forma (1, 128, 128), donde 1 representa el canal de la imagen en escala de grises y 128 x 128 son las dimensiones espaciales de la imagen. Primero, estos datos pasan a través de la primera capa convolucional, 'conv1', y se les aplica una función de activación ReLU, produciendo una salida de dimensiones (32, 128, 128). A continuación, se aplica la operación de Max Pooling, lo que reduce las dimensiones a (32, 64, 64).

La salida reducida pasa luego por la segunda capa convolucional, 'conv2', donde nuevamente se aplica la función de activación ReLU, resultando en una salida de tamañop (64, 64, 64). Después de aplicar Max Pooling, las dimensiones se

reducen a (64, 32, 32). A continuación, la salida de la segunda capa de pooling se aplanan mediante la función 'view', convirtiendo los datos tridimensionales en un formato unidimensional, lo que prepara los datos para su procesamiento por las capas totalmente conectadas.

En la primera capa totalmente conectada, se aplica nuevamente la activación ReLU, produciendo un vector de 128 características. A continuación, se pasa por la cada de dropout, que ayuda a reducir el sobre ajuste del modelo. Finalmente, los datos pasan por la segunda y última capa totalmente conectada, la cual genera un único valor de salida, que será utilizado para la clasificación binaria del evento como *glitch* o no.

Entrenando el modelo.

Para entrenar nuestra red *glitch_CNN()*, utilizamos la función `train_model()`, que recibe varios parámetros clave. El primer parámetro, 'model', corresponde al modelo que deseamos entrenar, en este caso, nuestra red *glitch_CNN()*. El segundo parámetro, 'train', se refiere al conjunto de datos utilizados para el entrenamiento, mientras que 'val' es el conjunto de datos utilizado para la validación del modelo.

La función de pérdida, especificada como 'loss', mide el error entre las predicciones del modelo y los valores reales. Dado que estamos enfrentando un problema de clasificación binaria, utilizamos la función *Binary Crossentropy* ('BCEWithLogitsLoss()').

El 'optimzier' es el algoritmo que se encarga de actualizar los pesos del modelo para minimizar la función de pérdida. En nuestro caso, utilizamos el optimizador *Adam*, definido como:

```
optimizer = optim.Adam(model.parameters(), learning_rate, weight_decay)
```

Aquí, *learninn rate* define la tasa de aprendizaje, mientras que *weight decay* se refiere a la regularización L2 aplicada para evitar el sobre ajuste.

También utilizamos un 'scheduler', que ajusta dinámicamente la tasa de aprendizaje a lo largo del entrenamiento, según una condición específica, como el comportamiento de la pérdida de validación. El parámetro 'epochs' define el número total de iteraciones completas que se realizarán sobre el conjunto de entrenamiento.

Finalmente, el parámetro 'patience' controla el mecanismo de *early stopping*, que detiene automáticamente el entrenamiento si la pérdida de validación no mejora después de un número determinado de épocas consecutivas, evitando así un entrenamiento innecesario.

Durante el entrenamiento, los datos se cargan y, en cada época, se calculan y almacenan los valores de la pérdida y la precisión. Estos valores nos permiten monitorear el rendimiento del modelo a lo largo del tiempo. Para la fase de validación, el modelo se utiliza en modo de evaluación 'model.eval()', lo que permite calcular los mismos valores de pérdida y precisión sin actualizar los pesos del modelo, garantizando que las predicciones se realicen de manera controlada y sin modificar el estado del modelo.

4.2.3. Búsqueda de hiper parámetros.

Con el modelo ya definido y utilizando el 10 % de los datos correspondientes al primer entrenamiento, es crucial encontrar la combinación de hiper parámetros que nos proporcione alta precisión y estabilidad. Para lograr esto, desarrollamos una serie de *scripts* capaces de obtener los mejores valores para cada hiper parámetro, considerando lo siguientes: *batch size*, el *learning rate*, el factor de regularización l2 (*weight decay*) y *dropout*.

La búsqueda de los valores óptimos para estos hiperparámetros se llevó a cabo de manera sistemática. Los valores considerados para el tamaño de lote fueron 16, 32, 64 y 128, mientras que para la tasa de aprendizaje se probaron 10^{-5}, 10^{-4} y 10^{-3}. Para el factor de regularización *weight decay*, se exploraron valores que iban desde 10^{-5} hasta 10^{-2} en potencias de 10. En cuanto al *dropout*, se evaluaron valores que iban desde 0.1 a 0.5, en pasos de 0.1.

El proceso de búsqueda de los mejores hiperparámetros fue progresivo. A medida que se encontraba un valor óptimo para uno de ellos, se procedía a ajustar el siguiente, utilizando los mejores valores obtenidos previamente. En el caso del *weigth decay* y del *dropout*, se implementaron *scripts* específicos para su búsqueda, mientras que el tamaño de lote y la tasa de aprendizaje se buscaron simultáneamente en un único *script*, utilizando diferentes combinaciones de estos valores.

Cada uno seguía una estructura clara. El *script* llamado *search_bs_lr.py* se dedicaba a la búsqueda de la mejor combinación entre el tamaño de lote y la tasa de aprendizaje. Por otro lado, *search_w_d.py* se enfocaba exclusivamente en la búsqueda del valor óptimo de *weight decay*. Finalmente, *search_d.py* se utilizaba para encontrar el mejor valor *dropout.*

Para cada *script*, se siguió un proceso estandarizado de entrenamiento de modelos. Este proceso incluyó la definición de la red, seguida de su entrenamiento para optimizar diferentes hiper parámetros, como la tasa de aprendizaje, el tamaño de lote, y el valor de *dropout rate*. En cada caso, se entrenó el modelo un máximo de 30 épocas con una paciencia de 5, basada en la pérdida (*loss*).

Durante el entrenamiento, se registraron los parámetros de rendimiento, como la exactitud (*accuracy*) y la pérdida (*loss*), tanto para las fases de entrenamiento como de validación. Estos datos se utilizaron para generar gráficos comparativos y se almacenaron en archivos *.log*, permitiendo revisar el progreso y comparar los resultados entre diferentes combinaciones de hiper parámetros.

En total, se entrenaron 20 combinaciones de tasa de aprendizaje y tamaño de lote, y posteriormente, con el valor de *weight decay* definido, se entrenó el modelo 10 veces más para optimizar el *dropout rate*, siguiendo el mismo procedimiento.

4.2.4. Validación cruzada.

Para validar nuestro modelo a través de este método, combinamos los datos del 10 % del entrenamiento y el 10 % de la validación en un total de 35,000 datos. Empleamos un esquema de validación cruzada con 7 *folds* para nuestro conjunto de entrenamiento. En cada *fold*, se utilizaron aproximadamente el 85 % de los datos para el entrenamiento y el otro 15 % la validación del modelo. Para cada uno de los *folds* se guardaron las métricas de desempeño del modelo, para su posterior análisis,

4.2.5. Entrenando el modelo.

Con el modelo optimizado tras la búsqueda de hiper parámetros y habiendo verificado su generalización mediante validación cruzada, el siguiente paso consistió en entrenarlo de forma definitiva con el 100 % de los datos mediante la función 'train_model()'.

Este entrenamiento se definió para ejecutarse durante un máximo de 100 épocas, con una paciencia de 5 épocas. También se definió un *learning rate scheduler* con una paciencia de 3 épocas, con el objetivo de poder afinar lo más posible el modelo antes de que su entrenamiento se detenga definitivamente.

Finalmente el modelo fue guardado como 'best_model.pth', el cual será cargado posteriormente para aplicar ajuste fino repitiendo el proceso de optimización y validación presentado en esta sección.

4.2.6. Ajuste fino o *fine tuning*.

Cargando el modelo pre entrenado.

Para realizar este proceso cargamos el modelo 'glitch_CNN()' como *"best_model,pth"*, el cual ya hemos pre entrenado con los datos del primer entrenamiento. Este modelo ya tiene una estructura capaz de detectar *glitches* en faseogramas, pero requiere ser refinado para adaptarse a *glitches* más pequeños. A continuación se muestra el código para cargar el modelo pre entrenado:

```
model = glitch_CNN(dropout_rate=dropout).to(device)
model.load_state_dict(torch.load('best_model.pth'))
```

Congelación de capas convolucionales.

Para aprovechar las características previamente aprendidas por el modelo pre entrenado, se optó por congelar las primeras capas convolucionales ('conv1'). Estas capas fueron entrenadas con un gran volumen de datos en la tarea original y ya habían capturado patrones relevantes en los fasoegramas. Congelar estas capas evita que sus pesos se actualicen durante el ajuste fino, enfocando el entrenamiento en las capas superiores, para que estas se adapten a los nuevos datos.

```
for param in model.conv1.parameters():
    param.requires_grad = False
```

Por otro lado, las capas posteriores ('conv2') y las capas completamente conectadas ('fc1' y 'fc2') fueron dejadas sin congelar, permitiendo que sus pesos se ajustaran durante el entrenamiento en la nueva tarea.

La capa 'conv2' se descongeló de la siguiente forma:

```
for param in model.conv1.parameters():
   param.requires_grad = True
```

Modificación de la capa de salida

Dado que el nuevo conjunto de datos y la tarea de detección implican una distribución ligeramente diferente, se modificó la última capa completamente conectada ('fc2') del modelo para adaptarla mejor a la nueva tarea. En lugar de producir una única salida binaria directamente, se agregó una capa intermedia para mejorar la capacidad del modelo para captar la complejidad de los nuevos datos.

```
model.fc2 = nn.Sequential(
    nn.Linear(128.64),
    nn.ReLU(),
    nn.Linear(64,1)
).to(device)
```

Entrenamiento y validación con Ajuste Fino

El modelo fue entrenado utilizando el nuevo conjunto de datos. Se utilizó un esquema de entrenamiento similar al descrito en la construcción principal del modelo, con el mismo proceso de cálculo de la función de pérdida y actualización de los pesos, pero ahora adaptado para la nueva tarea.

Para entrenar el modelo se utilizó la misma función ya definida 'train_model()'. En este caso al ingresar el modelo, este es la versión posterior a las modificaciones mencionadas anteriormente. La función de pérdida continuará siendo BCEWithLogitsLoss() y el optimizador será 'Adam'.

4.2.7. Procesamiento de Datos y Detección de *glitches* en Faseogramas.

Para aplicar de manera efectiva nuestro modelo en la detección de irregularidades en faseogramas, es esencial preprocesar los datos y alimentar el modelo con información adecuada. Este proceso se implementó a través de dos rutinas *Python*, una llamada 'detector.py' encargada de realizar el procesamiento de los datos, las predicciones y su almacenamiento, y 'utils.py', desde donde se importaban las funciones necesarias para el procesamiento y la estructura de la CNN.

Una regla importante en nuestro enfoque es que, aunque el modelo de aprendizaje está entrenado para detectar irregularidades en ventanas de 2 años, como se explicó anteriormente en la figura 4.3 y posteriores, es necesario contar con un mínimo de tres años de datos para optimizar la detección de *glitches*.

El flujo del código comienza con la preparación de los datos. Los datos descar-

gados directamente del satélite Fermi no incluyen la fase rotacional del púlsar ϕ. Para obtener esta información, es necesario procesas los datos mediante herramientas especializadas como 'tempo2' o 'Fermi tools', que permiten calculas la fase rotacional en función del tiempo. Una vez procesados, los datos se guardan en un archivo en formato '.fits', el cual contiene los tiempos de llegada de los pulsos al satélite, junto con las fases rotacionales calculadas. En el código, utilizamos la biblioteca 'astropy' para abrir y leer archivos '.fits'. De estos archivos, se extraen las columnas relevantes, que incluyen tanto la fase rotacional como el tiempo de llegada de los pulsos.

Posteriormente, con la información obtenida de los archivos '.fits', el código genera los faseogramas. Para facilitar este proceso, los datos se dividen en intervalos de tiempo. Cada uno de estos intervalos cubre un total de 730.5 días, con un solapamiento del 50 %. Este solapamiento asegura que se mantenga la continuidad temporal entre los gráficos generados. La fórmula utilizada para calcular el número total de faseogramas en función de la duración de los datos (T) en días, el intervalo de tiempo (I) y el solapamiento (S) es la siguiente:

$$N = \frac{T - I}{I - S} + 1 \tag{4.11}$$

En el código, esta operación se realiza a través de la función 'dividir_vector_por_intervalos', la cual divide los datos de tiempo y fase en intervalos con solapamiento. Esta función es importaba desde el archivo 'utils.py'. Posteriormente, se genera un histograma 2D para cada intervalo de tiempo, el cual se guarda en la carpeta 'graficos_fase_vs_tiempo'.

Una vez guardados los gráficos en formato '.png', estos se convierten a formato '.jpg' para su posterior procesamiento. En la fase de análisis y detección de *glitches*, el modelo previamente entrenado, denominado 'glitch_detectos_CNN.pth', es utilizado para analizar cada uno de los faseogramas y detectar posibles *glitches* en los datos. La estructura del modelo se encuentra definida en el archivo 'utils.py', y se importa a 'detector.py'. A través del archivo '.pth', se cargan los pesos del modelo entrenado, lo que permite realizar predicciones sobre cada faseograma generado.

Finalmente, los resultados obtenidos de la detección se almacenan en un archivo de texto cuyo nombre es especificado por el usuario. Además, cada faseograma, junto con su predicción, se guarda en formato '.jpg', permitiendo al investigador realizar una verificación visual de los resultados obtenidos.

4.3. Equipos utilizados.

Para este trabajo, se tuvo acceso a los servidores de la Universidad de Santiago de Chile (USACH). Se accedió a estos servidores utilizando el software *OpenVPN* junto con las credenciales proporcionadas por la universidad.

Una vez conectado a la red *VPN* de la universidad, se empleó el software *MobaXterm* para establecer conexión con los servidores a través de sus direcciones IP utilizando el protocolo SSH (Secure Shell). El comando utilizado para acceder a los servidores fue $ssh\ user@IP$, donde el nombre de usuario (*user*) y la dirección IP fueron proporcionados por la universidad.

Se tuvo acceso a dos servidores, cada uno con una dirección IP diferente, uno destinado para simulaciones y otro para el entrenamiento de modelos de machine learning. A continuación, se detallan las especificaciones de cada equipo.

Servidor para simulaciones

El servidor, denominado *fiscpu1*, estaba equipado con dos procesadores AMD EPYC 7662. Cada procesador cuenta con 64 núcleos y 128 hilos (2 hilos por núcleo) con una frecuencia base de 2.0 GHz. Este servidor soporta hasta 4TB de memoria DDR4, lo que lo hace adecuado para cargas de trabajo de centros de datos y computación de alto rendimiento.

Este servidor nos permitió simular nuestros 600,000 datos en 12 horas, al distribuir la carga de la simulación en 60 códigos distintos y ejecutando todos estos a la vez. Esto redujo el tiempo de dos meses a las 12 horas mencionadas.

Servidor para entrenamiento de modelos

El servidor utilizado para el entrenamiento de modelos, denominado *donnager*, estaba equipado con tres unidades de procesamiento gráfico (GPU) RTX A4500, basadas en la arquitectura Ampere. Estas GPUs cuentan con 7168 núcleos CUDA, 56 núcleos RT para trazado de rayos, 224 Tensor Cores, y 20 GB de memoria GDDR6, lo que las hace ideales para tareas intensivas en gráficos, inteligencia artificial y ciencia de datos. Además, el servidor contaba con dos procesadores AMD EPYC 7453, los cuales proporcionaron un soporte adicional para las cargas de trabajo de entrenamiento.

Capítulo 5: Resultados y análisis

5.1. Búsqueda de hiper parámetros

En la búsqueda de hiper parámetros y validación del modelo, en ambos casos se utilizó la función 'train_model()' con una paciencia de 5 épocas. con un máximo de 30 de entrenamiento. No se aplicó un *learning rate scheduler* en estos pasos.

5.1.1. Búsqueda para los *glitches* grandes.

En la primera búsqueda, el código arrojó los siguientes resultados para los mejores *batch size* y *learning rate:*

Tabla 5.1: *Exactitud de entrenamiento y validación para distintos valores de batch size y learning rate.*

Learning Rate	10^{-5}		10^{-4}		10^{-3}	
Batch Size	**Train Acc.**	**Val. Acc.**	**Train Acc.**	**Val. Acc.**	**Train Acc.**	**Val. Acc.**
16	0.9852	0.9796	0.9932	0.9800	0.9984	0.9766
32	0.9798	0.9680	0.9966	0.9820	0.9964	0.9818
64	0.9794	0.9798	0.9814	0.9750	0.9974	0.9774
128	0.9740	0.9706	0.9838	0.9756	0.9976	0.9778

Tabla 5.2: *Pérdida de entrenamiento y validación para distintos valores de batch size y learning rate.*

Learning Rate	10^{-5}		10^{-4}		10^{-3}	
Batch Size	**Train Loss**	**Val. Loss**	**Train Loss**	**Val. Loss**	**Train Loss**	**Val. Loss**
16	0.0453	0.0628	0.0214	0.0830	0.0062	0.1452
32	0.0661	0.0904	0.0120	0.0737	0.0102	0.0913
64	0.0683	0.0845	0.0571	0.1022	0.0091	0.0972
128	0.0835	0.1024	0.0514	0.0782	0.0089	0.1065

Como se observa en las tablas anteriores, en términos de exactitud, la combinación con mejor rendimiento corresponde a un tamaño de lote de 32 y una tasa de

aprendizaje de 10^{-4}, logrando un 98.20 % de exactitud en el conjunto de validación y una pérdida de 0.0913. Por otro lado, si nos enfocamos en la pérdida de validación, el valor más bajo se obtiene con una tasa de aprendizaje de 10^{-5} y un tamaño de lote de 16, con una pérdida de 0.0628 y una exactitud de 97.96 %.

Observando la tabla, se puede notar que la exactitud de validación no difiere en más de aproximadamente un 2 % con respecto a la exactitud de entrenamiento en todos los casos presentados. En cuanto a la pérdida, se observa que, a medida que aumentamos la tasa de aprendizaje, esta también incrementa. Un comportamiento similar se detecta al aumentar el tamaño de lote. Si analizamos los valores de pérdida de entrenamiento y validación, las combinaciones donde estos valores se mantienen más cercanos y bajos corresponden a un tamaño de lote de 16 con una tasa de aprendizaje de 10^{-5}. A pesar de que hay configuraciones con menores pérdidas en entrenamiento, estas difieren significativamente de las pérdidas en validación, lo que sugiere la presencia de sobre ajuste.

Por lo tanto, para priorizar la buena generalización del modelo, tomaremos la combinación de un tamaño de lote de 16 con una tasa de aprendizaje de 10^{-5}. A continuación vamos a analizar el impacto de diferentes valores de regularización L2 (o *weight decay*) en el rendimiento del modelo.

Tabla 5.3: *Comparación de la exactitud y pérdida para los distintos valores de weight decay.*

Weight Decay	Train Acc.	Val. Acc.	Train Loss	Val. Loss
1e-5	0.9852	0.9804	0.0433	0.0591
1e-4	0.9827	0.9762	0.0540	0.0733
1e-3	0.9823	0.9752	0.0555	0.0729
1e-2	0.9704	0.9756	0.0936	0.0941
1e-1	0.9432	0.9358	0.1856	0.1831

Después de realizar la búsqueda para determinar el mejor valor de *weight decay*, obtuvimos la tabla anterior, que resume los resultados. Podemos observar que la pérdida de validación incrementa consistentemente a medida que aumentamos el *weight decay*, alcanzando un máximo de 0.1831 cuando este valor es de 0.1. Este aumento en la pérdida indica que el modelo se vuelve menos preciso en sus predicciones conforme el valor de *weight decay* aumenta.

Paralelamente, la exactitud de validación disminuye en aproximadamente un 4 % al utilizar el mayor valor de *weight decay*, lo cual sugiere que la capacidad del mo-

delo para generalizar se ve comprometida con valores más altos de *weight decay*. Este deterioro en el rendimiento, tanto en términos de pérdida como de exactitud de validación, sugiere que los valores altos de *weight decay* introducen un exceso regularización, limitando la capacidad del modelo para capturar la complejidad subyacente en los datos.

Si analizamos los valores de pérdida y exactitud, ambos alcanzan su mejor desempeño con un valor de 10^{-5}. En particular, la diferencia de pérdidas entre el entrenamiento y la validación en la época 30 es de 0.0158, mientras que antes de la aplicación de la regularización esta diferencia era de 0.0175 en la misma época. Esta comparación sugiere que el menor valor de *weight decay* no solo mantiene una baja pérdida, sino que también reduce la discrepancia entre las pérdidas de entrenamiento y validación, lo que es indicativo de un buen equilibrio entre ajuste y generalización.

Por lo tanto, basándonos en estos resultados, el valor óptimo de *weight decay* seleccionado es 10^{-5}, ya que proporciona el mejor equilibrio entre baja pérdida de validación y alta exactitud de validación, optimizando así el rendimiento general del modelo. A continuación, se explorará la introducción de una capa de *dropout* y la búsqueda de su valor óptimo, lo que podría ayudar a mejorar la estabilidad y robustez del modelo en futuras iteraciones del entrenamiento.

Tabla 5.4: *Comparación de la exactitud y pérdida para los distintos valores de dropout.*

Dropout	Train Acc.	Val. Acc.	Train Loss	Val. Loss
0.1	0.9828	0.9808	0.0533	0.0657
0.2	0.9756	0.9744	0.0767	0.0835
0.3	0.9776	0.9758	0.0738	0.0785
0.4	0.9758	0.9784	0.0799	0.0799
0.5	0.9723	0.9750	0.0915	0.0872

En la tabla anterior, observamos que tanto la exactitud como la pérdida de entrenamiento tienden a deteriorarse a medida que incrementamos el valor de dropout. Este comportamiento es esperado, ya que valores mayores introducen regularización adicional al 'apagar' aleatoriamente una proporción de las neuronas durante el entrenamiento. Aunque esto ayuda a prevenir el sobre ajuste, también puede dificultar que el modelo se ajuste completamente a los datos de entrenamiento.

En el conjunto de validación, la disminución en el rendimiento es más gradual,

con una diferencia de solo 0.5 % en la exactitud entre un *dropout* de 0.1 y 0.5. Mientras tanto, en el conjunto de entrenamiento, la diferencia en la exactitud es más pronunciada, alcanzando aproximadamente un 1 % entre los mismos valores de dropout. Este comportamiento sugiere que, aunque el *dropout* aumenta la dificultar de ajuste durante el entrenamiento, su efecto sobre la capacidad del modelo de generalizar en datos no vistos es menos severo.

Los mejores valores, tanto en exactitud como de pérdida, se obtienen con un *dropout* de 0.1, donde la exactitud presenta una diferencia mínima de 0.2 % entre el entrenamiento y la validación, y la pérdida muestra una diferencia de solo 0.0124. Este valor de *dropout* logra la más alta exactitud de validación con un 98.08 % y la pérdida más baja con 0.06577. Estas métricas indican un buen equilibrio entre el ajuste a los datos de entrenamiento y la capacidad de generalización del modelo.

Comparando con los valores obtenidos con la regularización L2, el modelo con *dropout* 0.1 no solo mantiene una exactitud sobre el 98 %, sino que también reduce la diferencia entre la exactitud de entrenamiento y la validación en un 0.28 %. Además, la diferencia en la pérdida entre entrenamiento y validación disminuyó de 0.0158 a 0.0124. Esto sugiere que la adición de *dropout* con un valor de 0.1 proporciona una regularización más efectiva, permitiendo obtener un modelo con mejor generalización y menor riesgo de sobre ajuste.

Como vimos, el mejor valor de *dropout* corresponde a 0.1, ya que proporciona los mejores resultados en términos de pérdida y exactitud.

Con este ajuste, el siguiente paso es aplicar la validación cruzada, con el objetivo de confirmar que el modelo se comporta de manera consistente, independientemente del orden de los datos. A continuación, se presentarán los resultados de la exactitud y la pérdida para cada uno de los 7 *folds*, utilizando 35,000 datos.

Tabla 5.5: *Comparación de la exactitud y pérdida para los distintos pliegues de la validación cruzada.*

Fold	Train Acc.	Val. Acc.	Train Loss	Val. Loss
1	0.9830	0.9802	0.0516	0.0662
2	0.9797	0.9806	0.0644	0.0726
3	0.9795	0.9760	0.0640	0.0832
4	0.9768	0.9760	0.0744	0.0792
5	0.9820	0.9788	0.0571	0.0665
6	0.9815	0.9820	0.0556	0.0655
7	0.9834	0.9842	0.0524	0.0655

De acuerdo con la tabla anterior, el rendimiento del modelo en términos de exactitud fue bastante consistente a lo largo de diferentes *folds*, manteniéndose alrededor del 98 %. La diferencia entre la exactitud en el conjunto de entrenamiento y el conjunto de validación fue menor al 0.5 %, lo que indica que el modelo aprende de manera similar a lo largo de las distintas distribuciones de datos.

Sin embargo, la pérdida muestra una mayor variabilidad en sus resultados. Aunque los valores de la pérdida de validación se mantienen relativamente bajos, existe una mayor dispersión, con un valor máximo de 0.0832. En la mayoría de los casos, la diferencia entre la pérdida de entrenamiento y la de validación es inferior a 0.0198. Esta variabilidad podría estar relacionada con el hecho de que los faseogramas contienen un alto porcentaje de ruido rojo, lo que posiblemente dificultó el aprendizaje del modelo durante el entrenamiento. Como resultado, el modelo podría estar generalizando peor en algunos pliegues del conjunto de validación, donde podría haber faseogramas con la ausencia del ruido rojo. Alternativamente, es posible que algunos pliegues de validación contuvieran una cantidad significativa de estas imágenes, lo que impactó en la capacidad del modelo para generalizar adecuadamente.

En la tabla a continuación se presentan las métricas clave para evaluar el rendimiento del modelo en cada uno de los *folds* durante la validación cruzada, incluyendo *precision*, *recall*, F1 Score, ROC AUC y *Log Loss*. Estas métricas fueron calculadas utilizando un conjunto de datos de prueba independiente para cada *fold*, lo que proporciona una evaluación robusta del modelo en datos no vistos durante el entrenamiento.

Tabla 5.6: *Comparación de las métricas de rendimiento para los distintos pliegues de la validación cruzada*

Fold	Precisión	Recall	F1 Score	ROC AUC	Log Loss
1	0.9843	0.9780	0.9811	0.9964	0.0660
2	0.9823	0.9780	0.9802	0.9955	0.0748
3	0.9862	0.9696	0.9778	0.9954	0.0786
4	0.9845	0.9684	0.9764	0.9955	0.0783
5	0.9826	0.9728	0.9777	0.9946	0.0793
6	0.9818	0.9716	0.9767	0.9952	0.0761
7	0.9800	0.9804	0.9802	0.9953	0.0748
Promedio	0.9831	0.9741	0.9786	0.9954	0.0754
Error	0.0019	0.0043	0.0017	0.0005	0.0042

Los valores de precision y recall obtenidos en los diferentes *folds* son consistentemente altos, con promedios de 0.9831 y 0.9741 respectivamente. Estos resultados indican que el modelo propuesto maneja eficazmente la identificación de las clases, logrando un buen alcance entre la minimización de falsos positivos y la captura de verdaderos positivos. La alta precisión sugiere que el modelo rara vez m clasifica incorrectamente instancias negativas como positivas, mientras que el elevado recall muestra que el modelo es capaz de identificar la mayoría de las instancias positivas correctamente, minimizando los falsos negativos.

El F1 Score tiene un promedio de 0.9786 y presenta una variabilidad baja de 0.0017, lo cual es un indicador positivo de que el modelo mantiene un equilibrio estable entre ambas métricas a lo largo de los distintos *folds*. Esto indica que el modelo no favorece una métrica sobre la otra, sino que está bien equilibrado en su capacidad para detectar correctamente las clases positivas sin sobrestimar la clase negativa.

El valor del ROC AUC, con un promedio de 0.9954 y una desviación estándar extremadamente baja, de un 0.5 %, confirma que el modelo tiene un excelente rendimiento en términos de discriminación entre clases.

Por último, el log loss, con un promedio de 0.0754, refleja que el modelo tiene un buen desempeño al medir la probabilidad asignada a cada clase. Aunque los valores de este, presentan una mayor variabilidad (aproximadamente un 5.6 %), todos los valores están en un rango aceptable, con un máximo de 0.0786 en el tercer pliegue. Esto indica que el modelo está bien calibrado en términos de asignación de probabilidades, pero también sugiere que ciertos pliegues pueden contener datos más difíciles que generan esta variabilidad,

Finalmente con el modelo optimizado incluimos un scheduler en la función 'train_model()' con el objetivo de reducir la tasa de aprendizaje hasta un valor de 10^{-7} (Comenzó como 10^{-5}). Obteniendo así, la siguiente gráfica:

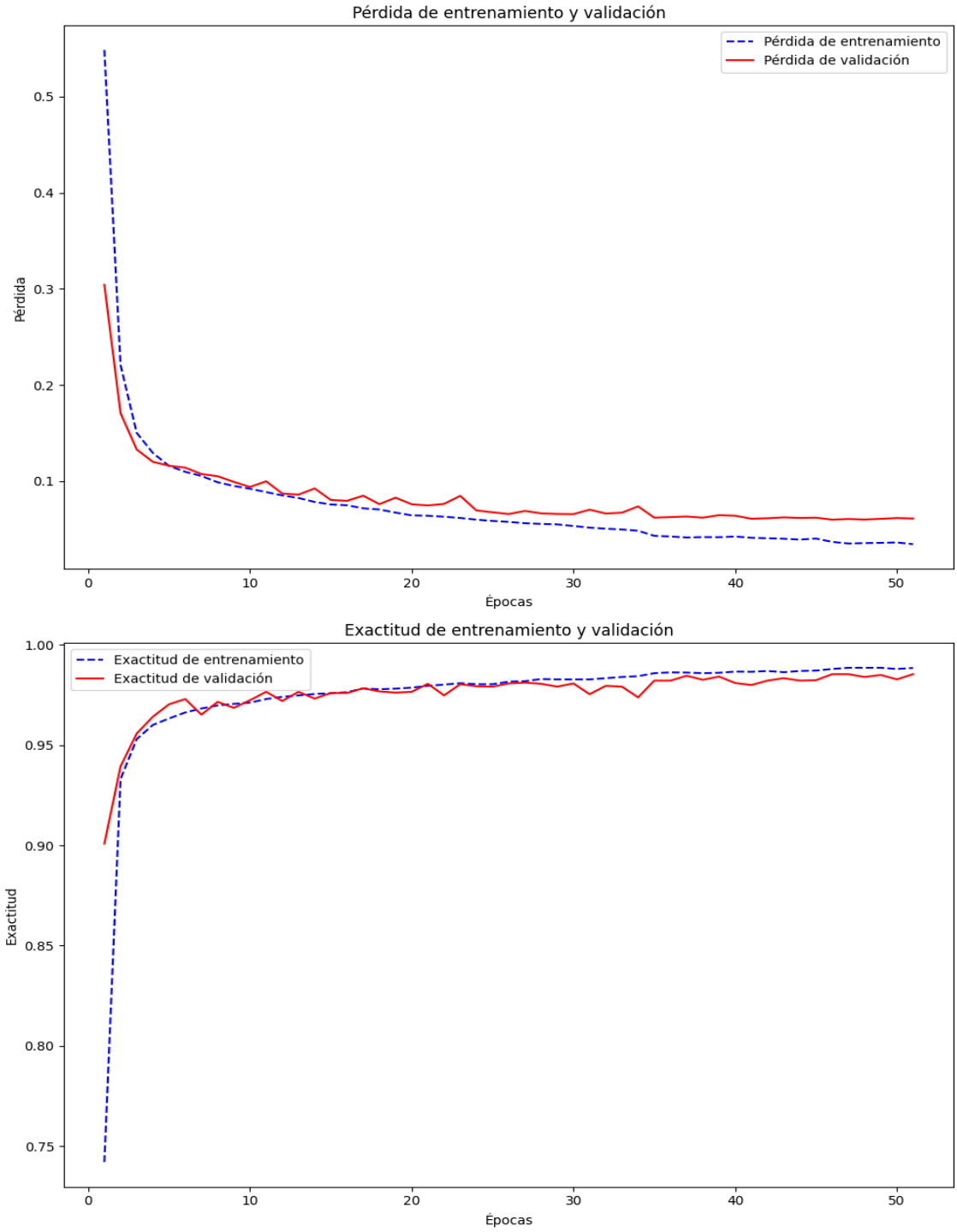

Figura 5.1: *Gráficas de pérdida y exactitud con la mejor combinación de batch size y learning rate. La exactitud de entrenamiento fue de un 98.85 %, mientras que la pérdida fue de 0.0344 durante la última época de entrenamiento.*

El entrenamiento completo fue inicialmente programado para ejecutarse durante 100 épocas, con un mecanismo de *early stopping* configurado con una paciencia de 5 épocas sin mejoras en el rendimiento de la pérdida de validación y un *learning rate scheduler* que llegaba hasta un valor de 10^{-7} y una paciencia de 3 épocas, este entrenamiento fue realizado con 35,000 datos, entre entrenamiento y validación y se corroboraron sus resultados con 5,000 datos no vistos previamente por el modelo. Como se puede observar en la figura, esto resultó en que el modelo se entrenara durante 51 épocas antes de detenerse.

La curva de pérdida de entrenamiento desciende rápidamente durante las 10 primeras épocas, lo que indica que el modelo está aprendiendo de manera efectiva, Posteriormente, ambas curvas, tanto de entrenamiento como de validación, se estabilizan y alcanzan valores bajos, lo que sugiere que el modelo ha convergido correctamente. Aunque la pérdida de entrenamiento es ligeramente inferior a la de validación durante la mayor parte del proceso, esta diferencia es mínima y no muestra signos de sobre ajuste. Después de aproximadamente de la época 35, la pérdida de validación se estabiliza hasta que el entrenamiento se detiene en la época 51.

Si notamos la curva de pérdida en el entrenamiento podemos ver como alrededor de las épocas 35, 45 y 51 hay pequeños escalones, estos se deben al *learning rate scheduler* definido anteriormente, el cual se encargó de ajustar la tasa de aprendizaje a medida que el modelo no presentaba mejoras en su rendimiento, donde al final, en la época 51, logramos ver que si bien, reduce la tasa de aprendizaje, el modelo no presenta una mejora en el rendimiento, viendo así, que este ya alcanzó el valor de convergencia alrededor de 0.0555.

Por otro lado, la curva de exactitud de entrenamiento y validación también aumenta rápidamente durante las primeras 10 épocas, en consonancia con la disminución observada en la pérdida. Ambas curvas muestran rendimiento elevados, superando el 98 % de exactitud de la vigésima época, y se mantienen estables durante el resto del entrenamiento, hasta alcanzar un 98.56 % en el conjunto de validación al final del entrenamiento.

De manera similar a lo observado con la pérdida, la exactitud de validación sigue de cerca la curva de entrenamiento, lo que es un buen indicio de que el modelo generaliza bien. Aunque la exactitud de validación presenta ligeras fluctuaciones, estas son mínimas y se mantienen alrededor del 98 %. Por ejemplo, en la época 34 se observa una pequeña disminución en la exactitud, alcanzando un valor del 97.38 %, pero posteriormente el modelo vuelve a estabilizarse alrededor del 98 % durante más de 10 épocas.

A continuación, en la figura 5.2 se presenta la matriz de confusión, que proporciona una visión detallada del desempeño del modelo al mostrar las predicciones correctas e incorrectas para cada clase.

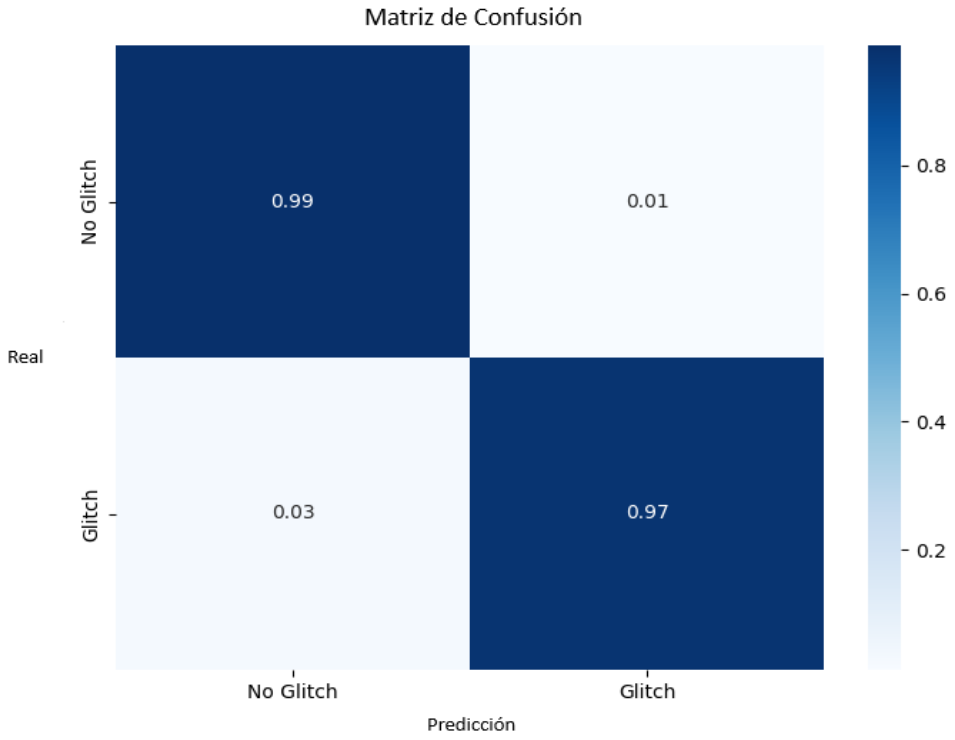

Figura 5.2: *Matriz de confusión obtenida con el set de datos de prueba con glitches con variaciones del orden de magnitud de 10^{-9} Hz*

La matriz de confusión muestra que el modelo posee un rendimiento muy alto en la clasificación de las dos clases. Esta matriz fue generada utilizando un conjunto de datos de prueba nuevos compuesto por 5,000 ejemplos, que no habían sido vistos previamente por el modelo.

En el cuadrante superior izquierdo, se observa que el 98 % de los ejemplos que no contenían *glitches* fueron correctamente clasificados como tales, mientras que solo el 2 % fue clasificado incorrectamente con la presencia de *glitche*s. Por otra parte, en el cuadrante inferior derecho, vemos que el 98 % de los datos que poseían *glitches* fueron etiquetados de la forma correcta.

Las métricas obtenidas sobre el conjunto de prueba refuerzan estos resultados.

La precisión del modelo es de 0.9820, lo que indica que una gran parte de las predicciones positivas son correctas. El recall es de 0.9824, lo que sugiere que el modelo identifica casi todas las instancias positivas disponibles. El F1 Score, que combina precisión y recall, se sitúa en 0.9822, reflejando un balance adecuado entre ambas métricas. La curva ROC AUC muestra un valor de 0.9965, lo que demuestra una excelente capacidad de discriminación entre las clases. Finalmente, el Log-Loss se reporta en 0.0617, lo que indica un buen rendimiento en términos de la probabilidad predicha frente a las clases reales.

Estos resultados indican que el modelo tiene una alta capacidad de generalización, manteniendo un excelente desempeño incluso con datos nuevos. La tasa de error es baja en ambas clases, con una proporción simétrica de falsos positivos y falsos negativos, lo que sugiere que el modelo no está sesgado hacia una clase en particular y puede identificar tanto *glitches* como la ausencia de estos de manera equilibrada.

Con este modelo ya optimizado y de alto rendimiento, se guardó bajo el nombre "best_model.pth", el cual se utilizará en el siguiente paso, el ajuste fino.

5.1.2. Ajuste fino.

Para realizar el ajuste fino utilizaremos el mismo método, buscaremos primero la mejor combinación de tasa de aprendizaje con el tamaño del lote, luego buscaremos el mejor para la regularización L2, y por último haremos la búsqueda de el mejor *dropout*. Tras esto validaremos el modelo mediante validación cruzada y por último lo entrenaremos con la totalidad de nuestros datos de *glitches* pequeños y así obtener nuestro modelo final.

A continuación los datos para obtener la mejor combinación de *batch size* y *learning rate*:

Tabla 5.7: *Exactitud de entrenamiento y validación para distintos valores de batch size y learning rate.*

Learning Rate	10^{-5}		10^{-4}		10^{-3}	
Batch Size	**Train Acc.**	**Val. Acc.**	**Train Acc.**	**Val. Acc.**	**Train Acc.**	**Val. Acc.**
16	0.8877	0.8728	0.8751	0.8456	0.9224	0.7992
32	0.8845	0.8760	0.8613	0.8424	0.9499	0.7976
64	0.8813	0.8812	0.8708	0.8552	0.9045	0.8072
128	0.8749	0.8728	0.8828	0.8496	0.9055	0.8384

Tabla 5.8: *Pérdida de entrenamiento y validación para distintos valores de batch size y learning rate.*

Learning Rate	10^{-5}		10^{-4}		10^{-3}	
Batch Size	**Train Loss**	**Val. Loss**	**Train Loss**	**Val. Loss**	**Train Loss**	**Val. Loss**
16	0.2893	0.3382	0.3136	0.3993	0.1956	0.5272
32	0.2990	0.3388	0.3447	0.3901	0.1323	0.6685
64	0.3165	0.3436	0.3225	0.3721	0.2384	0.4674
128	0.3361	0.3559	0.2993	0.3951	0.2405	0.4082

En la tabla se observa que al incrementar tanto el tamaño del lote como la tasas de aprendizaje, las métricas de exactitud y pérdida tienden a deteriorarse, especialmente en el caso de la pérdida, que empeora significativamente. El peor resultado se alcanza con una tasa de aprendizaje de 10^{-3} y un tamaño de lote de 32, donde la pérdida de validación alcanza un valor elevado de 0.6685. Esto indica que el modelo tiene dificultades para converger con tasas de aprendizaje más agresivas y tamaños de lote mayores.

Por otro lado, los valores más bajos de pérdida se obtienen con una tasa de aprendizaje más pequeña, específicamente con 10^{-5}, donde el valor de la pérdida de validación es de 0.3382 con un tamaño de lote de 16 y una exactitud de validación del 87.28 %. Este resultado sugiere que una tasa de aprendizaje más baja

ayuda a estabilizar el entrenamiento y a mejorar la capacidad de generalización del modelo, aunque con una exactitud ligeramente inferior.

Asimismo, la mayor exactitud en el conjunto de validación se obtiene con un tamaño de lote de 64 y una tasa de aprendizaje de 10^{-5}, alcanzando un 88.12 % de exactitud con una pérdida de validación de 0.3436- Esto implica que si se prioriza la exactitud sobre la pérdida, esta combinación podría ser una opción válida.

Como nos centraremos en mejorar el valor de la pérdida, utilizaremos la combinación de un tamaño de lote de 16 y una tasa de aprendizaje de 10^{-5}, que logra un buen equilibrio entre pérdida y exactitud. A continuación la tabla con los resultados de la búsqueda del mejor valor de *weight decay* para el modelo que estamos entrenando.

Tabla 5.9: *Comparación de la exactitud y pérdida para los distintos valores de weight decay.*

Weight Decay	Train Acc.	Val. Acc.	Train Loss	Val. Loss
1e-5	0.8861	0.8756	0.2920	0.3315
1e-4	0.8849	0.8592	0.2920	0.3587
1e-3	0.8796	0.8740	0.3055	0.3484
1e-2	0.8589	0.8616	0.3629	0.3710

Como podemos observar en la tabla, el mejor rendimiento en términos de exactitud se obtiene con un *weight decay* de 10^5, alcanzando un 87.56 % de exactitud en el conjunto de validación. Este valor también produce la pérdida de validación más baja, con 0.3315, lo que refuerza que esta configuración es la más adecuada en términos de generalización y ajuste.

En comparación con los valores reportados en la tabla1, se ha logrado un aumento de 0.28 %, en la exactitud de validación de 0.0067. Además, la diferencia entre la pérdida de entrenamiento y la de validación se ha reducido de 0.0489 a 0.0395, lo que indica una menor discrepancia entre ambos conjuntos y sugiere que el modelo está logrando un mejor equilibrio entre el ajuste a los datos de entrenamiento y su capacidad de generalización. Este comportamiento es útil para evitar el sobre ajuste.

Por lo tanto, con base en estos resultados, seleccionamos un *weight decay* de 10^{-5} como el valor óptimo. El próximo paso será aplicar distintos valores de *dropout* con el objetivo de reducir aún más las oscilaciones y mejorar la capacidad

de generalización del modelo. A continuación , se presentarán los resultados obtenidos para diferentes valores de este.

Tabla 5.10: *Comparison of Validation Accuracy and Loss for Different dropout Values*

Dropout	Train Acc.	Val. Acc.	Train Loss	Val. Loss
0.1	0.8809	0.8724	0.3075	0.3394
0.2	0.8722	0.8760	0.3322	0.3376
0.3	0.8651	0.8732	0.3448	0.3406
0.4	0.8604	0.8620	0.3672	0.3598
0.5	0.8437	0.8400	0.4047	0.4146

Según la tabla anterior, un *dropout* de 0.2 ofrece las mejores métricas tanto en exactitud como en pérdida, alcanzando un 87.60 % de exactitud en el conjunto de validación y una pérdida de 0.3376. Las diferencias entre el conjunto el conjunto de entrenamiento y validación son mínimas, con una diferencia de solo 0.38 % en la exactitud y 0.0054 en la pérdida. Esto representa una mejora considerable en comparación con las diferencias observadas anteriormente, donde la brecha entre entrenamiento y validación era del 1.05 % en la exactitud y 0.0395 en la pérdida.

Además, al observar como los valores de las métricas empeoran a medida que incrementamos el valor de *dropout*, podemos incluir que un valor de 0.2 no solo optimiza el rendimiento del modelo, sino que también minimiza el riesgo de sobre ajuste. Específicamente, valores de *dropout* superiores como 0.4 y 0.5 muestran una disminución en la exactitud y un aumento significativo en la pérdida, lo que sugiere que se introduce una regularización excesiva, limitando la capacidad del modelo para ajustarse adecuadamente.

Por lo tanto, la mejor opción para optimizar nuestro modelo, considerando el balance entre exactitud y pérdida, así como la reducción en la discrepancia entre entrenamiento y validación, es seleccionar un *dropout* de 0.2. Con el modelo ya optimizado, procedemos a presentar las tablas con los resultados de la validación cruzada.

Tabla 5.11: *Comparación de la exactitud y pérdida para los distintos pliegues de la validación cruzada.*

Fold	Train Acc.	Val. Acc.	Train Loss	Val. Loss
1	0.8729	0.8572	0.3368	0.3560
2	0.8727	0.8756	0.3278	0.3316
3	0.8658	0.8744	0.3464	0.3558
4	0.8723	0.8720	0.3333	0.3519
5	0.8681	0.8768	0.3325	0.3433
6	0.8723	0.8712	0.3253	0.3361
7	0.8657	0.8708	0.3498	0.3399

A partir de la tabla anterior podemos observar los resultados de la validación cruzada de nuestro modelo de ajuste fino, a lo largo de 30 épocas.

Respecto a la exactitud podemos observar que esta varia entre un 86.75 % y un 87.29 %, mientras que en el conjunto de validación oscila entre 85.72 % y 87.68 %. Esta consistencia en las métricas entre los diferentes *folds* indica que el modelo ha logrado generalizar bien a lo largo de las distintas distribuciones de datos.

Las pequeñas diferencias entre la exactitud de entrenamiento y validación, las cuales no superan el 1 % en su mayoría, indican que el modelo no está sobre ajustando y es capaz de detectar estos *glitches*.

Por otra parte, la pérdida en el conjunto de entrenamiento varía entre 0.3253 y 0.3498, mientras que la pérdida de validación entre 0.3316 y 0.3560. Aunque la pérdida de validación es ligeramente superior en todos los pliegues, las diferencias son menores a 0.015, lo que indica que el modelo si está pudiendo traspasar el aprendizaje de forma efectiva desde el conjunto de entrenamiento al de validación.

De todas formas la variación entre la exactitud en los conjuntos de validación oscilan alrededor de un 2 % entre el valor máximo y el mínimo, mientras que las pérdidas son menores a 0.025. Esta discrepancia puede tener relación con dos cosas. La primera consiste en la dificultad ya intrínseca en identificar desviaciones tan pequeñas en los faseogramas lo que haría más volátil al modelo a la detección correcta de los *glitches*, por otra parte puede tener relación a la distribución de los datos de los distintos pliegues, obteniendo un gran número de imágenes con ruido rojo alto en uno de los dos conjuntos (entrenamiento y validación) por lo que dificultaría la validación del modelo.

En la siguiente tabla veremos las métricas de rendimiento.

Tabla 5.12: *Comparación de las métricas de rendimiento para los distintos pliegues de la validación cruzada*

Fold	Precisión	Recall	F1 Score	ROC AUC	Log Loss
1	0.9123	0.8160	0.8615	0.9263	0.3446
2	0.8840	0.8536	0.8685	0.9308	0.3309
3	0.8802	0.8640	0.8720	0.9274	0.3476
4	0.8776	0.8600	0.8687	0.9268	0.3438
5	0.8809	0.8640	0.8724	0.9307	0.3376
6	0.8741	0.8776	0.8758	0.9340	0.3345
7	0.8818	0.8592	0.8703	0.9306	0.3413
Promedio	0.8844	0.8563	0.8699	0.9295	0.3400
Error	0.0118	0.0178	0.0041	0.0026	0.0055

La precisión varía entre un mínimo de 0.8776 y un máximo de 0.9123 a lo largo de los diferentes pliegues, con un promedio de 0.8844 y una desviación estándar relativamente baja de 1.18 %. Aunque esta métrica ha disminuido aproximadamente un 10 % en comparación con el primer entrenamiento, sigue indicando que el modelo es efectivo en minimizar los falsos positivos después del ajuste fino. Esto sugiere que el modelo clasifico correctamente las instancias en las que lo ocurrieron *glitches* en la mayoría de los casos, lo que es un buen indicador de su rendimiento en esta tarea.

El recall varía entre un máximo de 0.8776 y un mínimo de 0.8160, con un promedio de 0.8563 y una desviación estándar de 1.78 %. Este valor promedio es de 2.81 % menor que el de la precisión, lo que indica que, aunque el modelo es ligeramente menos efectivo en detectar *glitches*, aún mantiene un buen rendimiento, capturando la mayoría de los casos positivos a lo largo de los pliegues.

El F1 Score tiene un promedio de 0.8699 y una desviación estándar de 0.41 %. Esto muestra que el modelo logra mantener un buen balanca entre la capacidad de identificar correctamente los *glitches* y minimizar los falsos positivos.

El valor ROC AUC, que mide la capacidad del modelo para discriminar entre *glitches* y la ausencia de ellos, tiene un promedio de 0.9296 y una variabilidad de solo 0.26 %. Esto indica que el modelo discrimina bien entre ambas clases.

Por último, el Log Loss presenta un promedio de 0.3400 y una desviación están-

dar de 0.0055. Aunque este valor es más alto en comparación con el entrenamiento previo, su variabilidad es bastante reducida, lo que indica que el modelo ha logrado una convergencia consistente a lo largo de los distintos pliegues. Un valor de Log Loss más alto puede estar asociado con la mayor dificultad que presenta el nuevo conjunto de datos, debido a la menor magnitud de los *glitches*.

Si bien la mayoría de las métricas han disminuido aproximadamente un 10 % en comparación con el primer entrenamiento, este descenso no es necesariamente negativo o injustificado. Las pequeñas variaciones observadas entre los pliegues reflejan cómo el modelo ha logrado mantener un rendimiento estable, a pesar de la mayor complejidad introducida por los *glitches* de menor magnitud (del orden de 10^{-10}). Este comportamiento tiene sentido, ya que el modelo fue exitoso al diferencias *glitches* con variaciones del orden de 10^{-9}, pero su rendimiento ha disminuido notoriamente al tratar de detectar *glitches* más sutiles, como los entrenados en el ajuste fino. Como se observa en la figura 4.15, estos *glitches* de menor magnitud son extremadamente difíciles de distinguir, incluso para el ojo humano, lo que explica la reducción en el rendimiento.

Por esto, aunque las métricas han descendido, la consistencia del modelo en los diferentes pliegues y la dificultad inherente en la detección de *glitches* más pequeños justifican estos resultados.

Finalmente con el modelo optimizado incluimos un scheduler en la función 'train_model()' con el objetivo de reducir la tasa de aprendizaje hasta un valor de 10^{-7} (Comenzó como 10^{-5}). Obteniendo así, losgráficos de la figura 5.3.

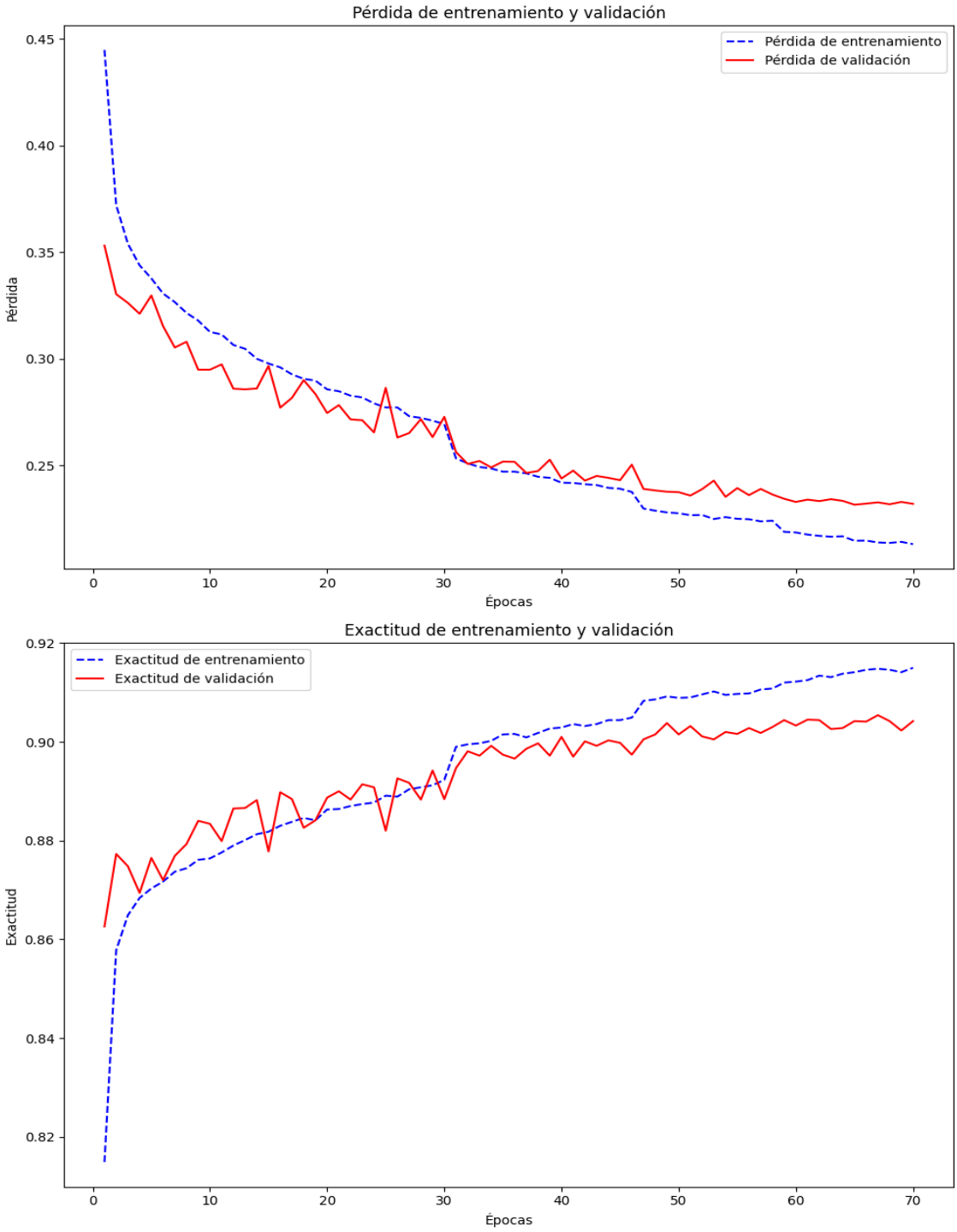

Figura 5.3: *Gráficas de pérdida y exactitud con la mejor combinación de batch size y learning rate. La exactitud de entrenamiento fue de un 91.50 %, mientras que la pérdida fue de 0.2320 durante la última época de entrenamiento.*

El entrenamiento fue inicialmente programado para ejecutarse durante 100 épocas, con un mecanismo de *early stopping* de 5 épocas sin mejoras en la pérdida de validación. Como se puede observar en la figura anterior, esto resultó en que el modelo se estrenara durante 70 épocas antes de detenerse.

La curva de pérdida de entrenamientos desciende rápidamente durante las primeras 10 a 15 épocas, lo que indica que el modelo está aprendiendo de manera efectiva. Posteriormente, ambas curvas, tanto de entrenamiento como de validación, continúan descendiendo y se estabilizan en torno a valores bajos. La pérdida de entrenamiento alcanza aproximadamente un valor de 0.21, mientras que la pérdida de validación se estabiliza alrededor de 0.23. La diferencia entre ambas es mínima, lo que sugiere que el modelo está generalizando correctamente y no hay indicios de sobre ajuste. Este comportamiento es un buen indicio de que el modelo no está memorizando los datos de entrenamiento, sino que está aprendiendo patrones generales útiles para predecir datos nuevos.

Por otro lado, la curva de exactitud de entrenamiento y validación también aumenta rápidamente durante las primeras 10 épocas, en consonancia con la disminución observada en la pérdida. Ambas curvas muestran valores elevados y estables, superando el 90 % de exactitud después de la vigésima época. La exactitud de entrenamiento se mantiene alrededor del 91.5 %, mientras que la exactitud de validación alrededor del 90.4 %. La pequeña diferencia de 1 % entre ambas curvas es otro indicio de que el modelo está generalizando bien, sin mostrar signos de sobre ajuste.

Tanto las curvas de pérdida como las de exactitud indican que el modelo converge correctamente. La pequeña diferencia entre el rendimiento de entrenamiento y validación sugiere que el modelo está bien ajustado y no hay evidencia significativa de sobre ajuste. La implementación de *early stopping* fue efectiva, deteniendo el entrenamiento en un punto óptimo en la época 70, cuando las curvas ya estaban estabilizadas. El rendimiento del modelo es sólido, con buena capacidad de generalización y un buen ajuste en los datos de validación.

En esta figura también somos capaces de ver los cambios generados por el *scheduler* en la tasa de aprendizaje de forma más pronunciada, tanto en la pérdida como en la exactitud. Estos ajustes se presentan en las épocas 30, 48 y 59 y nos permite ver como este cambio permitió que durante las últimas 10 épocas de entrenamiento, la pérdida se mostró constante alrededor del valor 0.2320.

Finalmente el modelo se evaluó en el conjunto de prueba, que contenía 25,000 datos. En la figura 5.4 podemos observar la matriz de confusión.

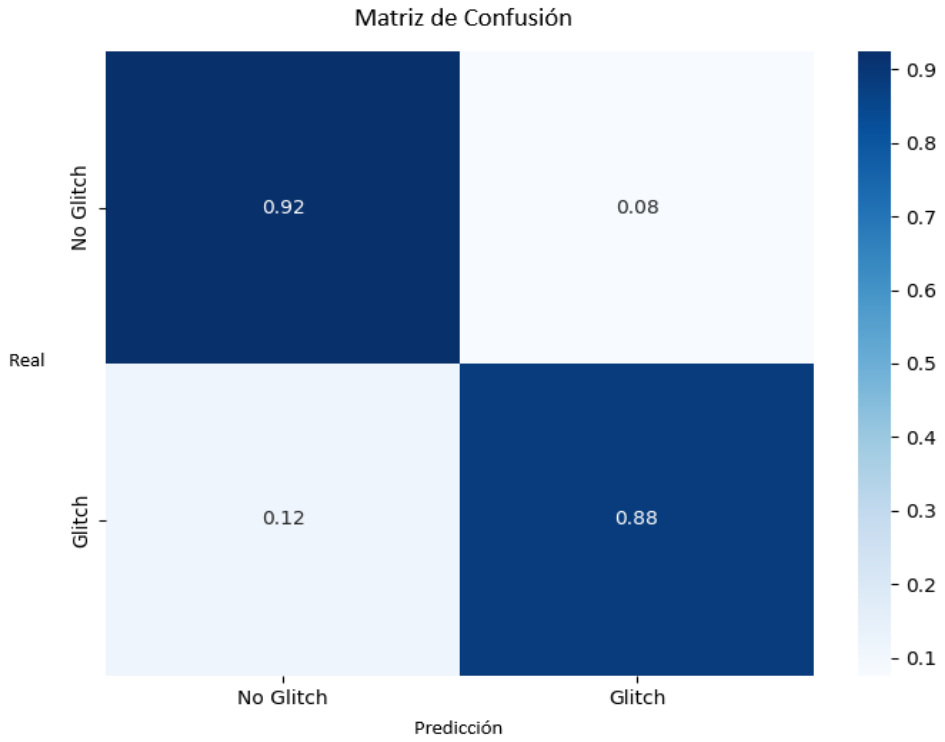

Figura 5.4: *Matriz de confusión obtenida con el set de datos de prueba con glitches con variaciones del orden de magnitud de 10^{-10} Hz*

La matriz de confusión demuestra un buen rendimiento del modelo en la clasificación de ejemplos con y sin *glitches*. En el cuadrante superior izquierdo, observamos que el 92 % de los ejemplos que no contenían irregularidades fueron etiquetadas correctamente, mientras que el 8 % fue erróneamente etiquetado como *glitches*. En el cuadrando inferior derecho, se muestra que el 88 % de los datos que contenían *glitches* fueron correctamente clasificados, mientras que el 12 % tuvieron etiquetas incorrectas.

Estas cifras reflejan la capacidad del modelo para generalizar a nuevos datos, manteniendo una baja tasa de error en ambas clases. Las métricas adicionales obtenidas sobre el conjunto de prueba refuerzan este buen desempeño. La precisión del modelo es de 0.9210, lo que indica que la mayoría de las predicciones positivas son correctas. El recall, con un valor de 0.8843, sugiere que el modelo identifica un porcentaje significativo de las instancias positivas, aunque hay margen de mejora en este aspecto. El F1 Score se sitúa en 0.9023, lo que indica un equilibrio favorable entre precisión y recall. La curva ROC AUC alcanza un valor

de 0.9642, lo que demuestra una sólida capacidad del modelo para discriminar entre las clases. Finalmente, el Log-Loss es de 0.2380, lo que sugiere que el modelo presenta un rendimiento aceptable en términos de probabilidad predicha frente a las clases reales.

El elevado valor de la precisión (0.9210) indica que el modelo es eficaz en la minimización de falsos positivos, clasificando correctamente la mayoría de los ejemplos sin *glitches*. Por su parte, el recall de 0.8843 refleja la capacidad del modelo para detectar eficazmente ejemplo con *glitches*, minimizando los falsos negativos. El F1 Score de 0.9023, que combina precisión y recall, confirma que el modelo mantiene un buen equilibrio entre ambos aspectos.

El ROC AUC de 0.9642 destaca la excelente capacidad del modelo para distinguir entre las dos clases (*glitches* o no *glitches*), mientras que el bajo log-loss de 0.2380 sugiere que el modelo produce predicciones bien calibradas, ayudando su toma de decisiones.

Estos resultados en conjunto muestran que el modelo no está sesgado hacia ninguna de las dos clases y que mantiene un buen rendimiento, tanto la detección de *glitches* y la identificación correcta cuando estos no están.

Estos valores son superiores a las métricas obtenidas durante la validación cruzada, lo que puede explicarse por dos factores clave. Primero, durante la validación cruzada se utilizó solamente el 10 % de los datos de entrenamiento y validación, lo que corresponde a 12,500 ejemplos. En contraste, durante el entrenamiento final, tal como se muestre en la figura 5.3, el conjunto de entrenamiento se incrementó a 150,000 ejemplos, mientras que el conjunto de validación fue de 25,000 ejemplos.

El entrenamiento durante más épocas (40 épocas más que en la validación cruzada) y el uso de una mayor cantidad de datos contribuyeron a mejoras significativas en las métricas, aumentando aproximadamente 0.04 en la mayoría de ellas, con la excepción del log loss, que disminuyó de 0.3400 en la validación cruzada a 0.2380 en el conjunto de prueba terminan.

Finalmente, con el modelo ya optimizado y con buenos resultados, lo guardamos como 'glitch_detector_cnn.pth' para seguir la prueba con *glitches* reales registrados.

Conclusiones

En este trabajo, desarrollamos un modelo de redes neuronales convolucionales (CNN) para la detección automática de *glitches* en faseogramas de púlsares de rayos gamma. Debido a la escasez de datos observacionales de este tipo, el primer paso consistió en desarrollar un código capaz de generar artificialmente la emisión de fotones e inducir irregularidades como *timing noise* y *glitches* en la fase de rotación de los púlsares. Esto permitió simular un total de 600,000 púlsares y sus fases de rotación, con y sin *glitches*, formulando así un problema de clasificación binaria.

Con el objetivo de automatizar el proceso de detección de *glitches* de baja magnitud, construimos y optimizamos una red neuronal, preentrenada con 35,000 datos, de los cuales la mitad correspondía a *glitches* con variaciones en la frecuencia del orden de magnitud de 10^{-9} Hz, mientras que la otra mitad no presentaba esta irregularidad. Esto permitió generar un modelo con una precisión del 98.2 % en la clasificación de un conjunto de prueba de 5,000 datos.

Posteriormente, mediante técnicas de transferencia de aprendizaje y ajuste fino, el modelo fue adaptado para aprender las características de *glitches* con variaciones en la frecuencia del orden de 10^{-10} Hz. Este modelo fue validado y optimizado con un total de 17,500 datos, tras lo cual se entrenó con 175,000 datos adicionales. Al aplicarlo sobre un conjunto de prueba de 25,000 faseogramas, el modelo alcanzó una precisión del 92.1 %, con una tasa de aciertos del 88.4 % en la detección de *glitches*.

Estos resultados demuestran la capacidad de las CNN para automatizar de manera precisa la detección de *glitches* en púlsares, complementando y potencialmente optimizando el análisis visual en la búsqueda de estos eventos en los faseogramas de rayos gamma. No obstante, el modelo enfrenta desafíos en la detección de *glitches* más pequeños, lo que indica margen de mejora tanto en las simulaciones como en la arquitectura del modelo o el enfoque adoptado.

Aunque el modelo final se entrenó con 175,000 datos, su optimización se realizó con solo el 10 % de estos debido a limitaciones de tiempo, ya que cada ciclo de pruebas tomaría aproximadamente 12 horas. Una posible mejora sería realizar

la optimización con un porcentaje mayor de datos, dado que se observó que al aumentar la cantidad de datos, el *recall* del modelo mejoró del 85.6 % al 88.4 %, lo que sugiere que un mayor volumen de datos podría traducirse en un mejor desempeño del modelo. Por lo que, una manera de mejorar nuestro actual trabajo es entrenar el primer modelo con los 400,000 datos preparados y posteriormente aplicar ajuste fino y la optimización del modelo con los 200,000 datos generados para eso.

Por otro lado, no pudimos validar nuestro modelo con *glitches* reales, ya que la documentación sobre la ocurrencia de estos *glitches* pequeños en púlsares de rayos gamma es escasa. Encontramos un caso con una variación del orden de 10^{-11} Hz; sin embargo, el problema fue que ocurrió antes del MJD mínimo permitido para la extracción de datos en Fermi, lo que impidió acceder a esta información.

Debido al problema mencionado anteriormente, una mejora potencial sería haber optado por una clasificación múltiple en lugar de binaria, simulando *glitches* de diferentes órdenes de magnitud. Esto habría permitido una mayor generalización del modelo, que actualmente está limitado a la detección de *glitches* con variaciones en la frecuencia del orden de 10^{-9} y 10^{-10} Hz. Con una clasificación múltiple, el modelo podría detectar *glitches* de diferentes magnitudes, automatizando de manera más robusta el análisis de faseogramas, reduciendo así la necesidad de intervención humana.

En este trabajo observamos cómo, mediante la ingeniería y la aplicación de técnicas de aprendizaje automático, se facilita el estudio de los objetos celestes en astronomía. El análisis automatizado de grandes volúmenes de datos astronómicos, como los obtenidos de telescopios y satélites, es esencial para detectar patrones y fenómenos sutiles que, aunque podrían identificarse manualmente, requerirían mucho más tiempo, lo que resultaría poco eficiente para el estudio de los astros. En particular, el aprendizaje automático permite procesar y clasificar datos masivos de manera rápida y precisa, como en la detección de anomalías en señales de púlsares o la identificación de exoplanetas.

Estos enfoques abren una ventana de posibilidades para optimizar las observaciones astronómicas y la identificación de patrones, proporcionando herramientas más robustas y eficientes que permiten dedicar más tiempo al análisis profundo de los fenómenos detectados.

Bibliografía

[1] IBM. What is machine learning? https://www.ibm.com/topics/machine-learning, 2024. Consultado el 25 de septiembre de 2024.

[2] IBM. What are convolutional neural networks? https://www.ibm.com/es-es/topics/convolutional-neural-networks, 2024. Consultado el 25 de septiembre de 2024.

[3] Mohd Hasan Muhammed Fathin, Zunaira Hasib, and Abdullah Aldrin. Memory and computational efficient convolutional neural networks. In *2017 International Conference On Smart Technology for Smart Nation (SmartTechCon)*, pages 903–907. IEEE, 2017.

[4] Paperspace. Weights and biases. https://machine-learning.paperspace.com/wiki/weights-and-biases, 2024. Consultado el 25 de septiembre de 2024.

[5] Siddharth Sharma, Simone Sharma, and Anidhya Athaiya. Activation functions in neural networks. *International Journal of Engineering Applied Sciences and Technology*, 4(12):310–316, apr 2020. Consultado el 25 de septiembre de 2024.

[6] Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks. *arXiv preprint arXiv:1511.08458v2*, 2015.

[7] IBM. Model tuning parameters. https://www.ibm.com/docs/en/watsonx/saas?topic=model-tuning-parameters, 2024. Consultado el 25 de septiembre de 2024.

[8] IBM. What is regularization? https://www.ibm.com/topics/regularization, 2024. Consultado el 25 de septiembre de 2024.

[9] Ron Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. *International Joint Conference on Artificial Intelligence (IJCAI)*, 14(2):1137–1145, 1995.

[10] David M W Powers. Evaluation: From precision, recall and f-measure to roc, informedness, markedness and correlation. *Journal of Machine Learning Technologies*, 2(1):37–63, 2011.

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

[12] Marina Sokolova and Guy Lapalme. A systematic analysis of performance measures for classification tasks. *Information Processing and Management*, 45(4):427–437, 2009.

[13] Yutaka Sasaki. The truth of the f-measure. *Teach Tutor Mater*, pages 1–5, 2007.

[14] Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective*. MIT Press, 2012.

[15] Tom Fawcett. An introduction to roc analysis. *Pattern Recognition Letters*, 27(8):861–874, 2006.

[16] IBM. What is transfer learning? https://www.ibm.com/topics/transfer-learning, 2024. Consultado el 25 de septiembre de 2024.

[17] IBM. What is fine-tuning? https://www.ibm.com/topics/fine-tuning, 2024. Consultado el 25 de septiembre de 2024.

[18] Adam Paszke, Sam Gross, Soumith Chintala, Edward Yang, Zachary DeVito, Angela Lin, Babak Gharachorloo, and et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in Neural Information Processing Systems*, 32, 2019.

[19] Walter Lewin and Warren Goldstein. *Por amor a la física: Del final del arco iris a la frontera del tiempo. Un viaje por las maravillas de la física*. Penguin Random House Grupo Editorial España, feb 2012. Consultado el 13 de abril de 2019.

[20] A. A. Abdo, M. Ajello, A. Allafort, L. Baldini, J. Ballet, G. Barbiellini, M. G. Baring, D. Bastieri, A. Belfiore, R. Bellazzini, et al. The second fermi large area telescope catalog of gamma-ray pulsars. *The Astrophysical Journal Supplement Series*, 208(2):17, Sep 2013. Consultado el 13 de abril de 2019.

[21] P. S. Ray, M. Kerr, D. Parent, A. A. Abdo, L. Guillemot, S. M. Ransom, N. Rea, M. T. Wolff, A. Makeev, M. S. E. Roberts, F. Camilo, M. Dormody, P. C. C. Freire, J. E. Grove, C. Gwon, A. K. Harding, S. Johnston, M. Keith, M. Kramer, P. F. Michelson, R. W. Romani, P. M. Saz Parkinson, D. J. Thompson, P. Weltevrede, K. S. Wood, and M. Ziegler. Precise gamma-ray timing and radio observations of 17 fermi gamma-ray pulsars. *The Astrophysical Journal Supplement Series*, 194:17, jun 2011. Consultado el 25 de septiembre de 2024.

[22] Felix Patzelt. colorednoise.py. https://github.com/felixpatzelt/colorednoise/blob/master/colorednoise.py, 2016. Consultado el 25 de septiembre de 2024.

[23] Danai Antonopoulou, Brynmor Haskell, and Cristóbal M. Espinoza. Pulsar glitches: observations and physical interpretation. *Reports on Progress in Physics*, 85(12):126901, 2022.

[24] E. V. Sokolova and A. G. Panin. Search for glitches of gamma-ray pulsars with deep learning. *Astronomy & Astrophysics*, may 2021. Consultado el 25 de septiembre de 2024.