Universidad de Valladolid

Escuela de Ingeniería Informática

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática
Mención en Tecnologías de la Información

Técnicas de Detección y Mitigación de Ransomware basadas en Machine Learning

Autor: Adrián Sanz Martín

Universidad de Valladolid

Escuela de Ingeniería Informática

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática
Mención en Tecnologías de la Información

Técnicas de Detección y Mitigación de Ransomware basadas en Machine Learning

Autor: Adrián Sanz Martín

Tutor: Valentín Cardeñoso Payo

A mi familia, por ser mi base, mi impulso y mi mayor apoyo en cada paso de este camino.

Agradecimientos

Han sido muchas las personas que han contribuido, de una forma u otra, a que haya podido completar esta etapa universitaria y llegar hasta el final de la carrera completando así este Trabajo de Fin de Grado.

En primer lugar, quiero agradecer a mi tutor, quien me ha ayudado a orientarme y a mantener una dirección clara durante el desarrollo del Trabajo de Fin de Grado. Su guía ha sido fundamental para poder afrontar esta última etapa con mayor seguridad.

También quiero dar las gracias a mis amigos, que han estado presentes a lo largo de estos años, compartiendo tanto los buenos momentos como las dificultades. Su apoyo y compañía han hecho este camino mucho más llevadero y enriquecedor.

Y, sobre todo, quiero dedicar este logro a mi madre, por estar siempre a mi lado, por su apoyo incondicional y por creer en mí desde el primer día. Su constancia, cariño y ánimo han sido claves durante toda la carrera.

Por último, gracias a mi pareja, por su paciencia, por entender los momentos de ausencia y estrés, y por ser una fuente constante de motivación y tranquilidad.

A todos, gracias.

Resumen

En la actualidad, la detección de amenazas informáticas es un desafío creciente debido al constante aumento en la cantidad y complejidad del malware. Este Trabajo de Fin de Grado se centra en el análisis, diseño y evaluación de diferentes modelos de deep learning aplicados a la detección y atribución de muestras de malware. Para ello, se ha llevado a cabo un proceso exhaustivo de conversión y preprocesamiento de las muestras, transformándolas en representaciones visuales capaces de ser interpretadas por arquitecturas de redes neuronales.

A lo largo del proyecto se han entrenado y evaluado diversos modelos como CNN, Bi-LSTM, CNN-BiLSTM y Bi-LSTM-GN, utilizando técnicas de validación cruzada y métricas estándar para comparar su rendimiento en tareas de clasificación binaria, multi-clase y de atribución por familia. Los resultados obtenidos demuestran una mejora significativa en precisión y rendimiento a medida que se incorporan técnicas como la normalización de gradientes o el uso de arquitecturas híbridas.

Este trabajo no solo contribuye al ámbito académico y técnico con una propuesta efectiva para la detección de malware, sino que también ha permitido al autor adquirir una visión más profunda de los retos de la ciberseguridad y su impacto en la vida cotidiana, destacando la importancia de desarrollar soluciones automatizadas, precisas y escalables ante una amenaza digital en constante evolución.

Abstract

Currently, the detection of computer threats is a growing challenge due to the constant increase in the quantity and complexity of malware. This thesis focuses on the analysis, design, and evaluation of different deep learning models applied to the detection and attribution of malware samples. To this end, an exhaustive process of sample conversion and preprocessing was carried out, transforming them into visual representations capable of being interpreted by neural network architectures.

Throughout the project, various models such as CNN, Bi-LSTM, CNN-BiLSTM, and Bi-LSTM-GN were trained and evaluated using cross-validation techniques and standard metrics to compare their performance in binary, multi-class, and family attribution tasks. The results obtained demonstrate a significant improvement in accuracy and performance as techniques such as gradient normalization or the use of hybrid architectures were incorporated.

This work not only contributes to the academic and technical fields with an effective approach to malware detection, but has also allowed the author to gain a deeper understanding of cybersecurity challenges and their impact on everyday life, highlighting the importance of developing automated, accurate, and scalable solutions in the face of a constantly evolving digital threat.

Índice general

Índice de cuadros

Índice de figuras

Capítulo 1

Introducción

1.1 Contexto

La era contemporánea de la tecnología ha transformado la manera en que nos comunicamos y accedemos a la información. Sin embargo, junto con estos avances han aparecido varios retos, especialmente en el ámbito de la ciberseguridad. El auge de Internet y su difusión global han proporcionado a los cibercriminales una oportunidad sin precedentes para explotar su accesibilidad y el anonimato que permite el mundo digital al realizar ataques contra individuos y empresas [11] .

La creación de la World Wide Web (WWW) [67] facilitó el acceso a nuevas estrategias para los atacantes, dándoles más maneras de introducirse en los sistemas informáticos. También, el crecimiento de los dispositivos conectados multiplicó enormemente el riesgo de ataque y la capacidad de causar interrupciones a gran escala [23]. A medida que el entorno digital ha madurado, las metodologías de los ciberdelincuentes han cambiado, incorporando tácticas más avanzadas, entre las que se destaca el uso de , software malicioso diseñado específicamente para infiltrarse, perjudicar o acceder sin autorización a sistemas y datos [12].

Sin embargo, el *malware* fue solo el inicio del aumento en la complejidad del cibercrimen, alcanzando un punto crítico con la llegada del *ransomware* , un software dañino que se presenta como uno de los mayores retos en ciberseguridad en la actualidad debido a su capacidad destructiva y su modus operandi de extorsión [57].

En este estudio, se busca desarrollar y evaluar estrategias efectivas para la detección y clasificación de *ransomware* , con el fin de superar las limitaciones de los métodos tradicionales de detección basados en firmas. Para lograr esto, se propone un enfoque avanzado que aprovecha el potencial del aprendizaje profundo, particularmente centrado en identificar amenazas nuevas y disfrazadas. Esta investigación tiene como objetivo contribuir a mejorar la precisión y resistencia de los sistemas antivirus modernos, fortaleciendo la defensa ante uno de los ataques más perjudiciales en el ámbito actual de la ciberseguridad [34].

1.2 Motivación

El *ransomware* se ha convertido en una amenaza importante en materia de ciberseguridad, con ataques dirigidos contra individuos, corporaciones y entidades gubernamentales por igual, a menudo con devastadores impactos financieros y operativos. Los atacantes de *ransomware* utilizan diversas técnicas, desde el cifrado y la ex-filtración de datos hasta la ingeniería social, para comprometer y extorsionar a sus víctimas.

Los mecanismos de defensa convencionales, si bien son eficaces contra amenazas conocidas, a menudo resultan insuficientes para detectar nuevos ataques de *ransomware* , ya que muchos dependen en gran medida de técnicas de detección estáticas que carecen de adaptabilidad. Las limitaciones de estos enfoques son evidentes en

su lucha por manejar las tácticas de evasión dinámica del *ransomware* , como la manipulación de algoritmos de cifrado y los métodos antiforenses, que se emplean regularmente para eludir los sistemas de detección estándar.

Estos desafíos demuestran la necesidad de estrategias de detección más avanzadas que puedan adaptarse en tiempo real, alineándose continuamente con la naturaleza cambiante del *ransomware* .

1.3 Revisión del estado actual

La detección y clasificación de *malware* es un campo de investigación ampliamente estudiado debido al aumento constante de amenazas cibernéticas, entre las que destaca el *ransomware* por su impacto económico y social. Tradicionalmente, los mecanismos de detección se han basado en técnicas estáticas, como el análisis de firmas, y dinámicas, como la ejecución del software en entornos controlados (sandboxing). Sin embargo, estos enfoques presentan importantes limitaciones frente a variantes nuevas o técnicas de ofuscación.

Ante estos retos, el uso de métodos basados en aprendizaje automático y, más recientemente, en aprendizaje profundo, han ganado protagonismo en los últimos años. Modelos como las redes neuronales multicapa (MLP), redes convolucionales (CNN) y redes recurrentes (LSTM o Bi-LSTM) han demostrado ser capaces de aprender patrones complejos a partir de grandes volúmenes de datos y, por tanto, mejorar significativamente la detección de *malware*.

Además de la detección binaria (es decir, determinar si un archivo es malicioso o no), se ha empezado a trabajar también en la atribución por familias, lo que permite no solo identificar el *malware*, sino también clasificarlo según su origen o comportamiento. Este tipo de clasificación es especialmente útil en el caso del *ransomware* , ya que conocer a qué familia pertenece una amenaza puede facilitar la aplicación de contramedidas más eficaces.

En la actualidad, algunos estudios han propuesto arquitecturas híbridas como CNN-LSTM o CNN-BiLSTM, que combinan el procesamiento espacial y secuencial de datos para mejorar la precisión. También se han comenzado a explorar modelos más avanzados, como los basados en redes de grafos (Graph Networks), que permiten representar relaciones estructurales entre componentes del ejecutable y, al combinarlas con Bi-LSTM, capturar la evolución temporal del comportamiento del *malware*.

A pesar de estos avances, persisten ciertos retos: muchos modelos se entrenan con datasets poco actualizados o con un número reducido de muestras, lo que afecta a su capacidad de generalización. Por otro lado, pocos trabajos analizan el impacto de cada característica utilizada en la clasificación, lo que limita la interpretabilidad de los modelos y su aplicabilidad en entornos reales.

En este contexto, se hace necesario seguir investigando modelos que no solo logren una alta precisión en tareas de detección y atribución, sino que también sean capaces de adaptarse a nuevas amenazas y de aprovechar al máximo la información contenida en los archivos analizados.

1.3.1 Evaluación del estado

Según un informe publicado por sophos [62] en un estudio realizado sobre más de 5000 encuestados acerca del estado del *ransomware* en 2024, el 59 % de las organizaciones se vieron afectadas por el *ransomware* , un ligero descenso respecto al 66 % de los dos años anteriores. Esto no quiere decir que ya se haya solventado este problema y lo dejemos de lado, al contrario, hay que seguir por esa línea de trabajo y seguir al tanto de los nuevos ciberataques, para que poco a poco y año tras año siga bajando cada vez más ese porcentaje.

A pesar de los avances en ciberseguridad, el *ransomware* continúa siendo una de las amenazas más persistentes y peligrosas para las organizaciones. Resulta alentador que, durante el último año, todas las franjas de ingresos empresariales hayan experimentado una reducción en la tasa de ataques de *ransomware* . No obstante, esta mejora no ha sido uniforme; por ejemplo, en el caso de las empresas con ingresos entre 500 y 1000 millones de dólares, la disminución fue inferior a un punto porcentual.

Figura 1.1: Indice de organizaciones afectadas

La probabilidad de sufrir un ataque de *ransomware* tiende a incrementarse con el nivel de ingresos de la organización. Las empresas con una facturación superior a los 5000 millones de dólares presentan la tasa más elevada de ataques, alcanzando un 67 %. Sin embargo, las pequeñas empresas tampoco están exentas: cerca del 47 % de las que ingresan menos de 10 millones de dólares también fueron víctimas de este tipo de ataques en el último año, como muestra la Figura 1.2.

Además, se ha detectado un cambio en el perfil de los atacantes. Aunque muchos incidentes continúan siendo ejecutados por grupos altamente organizados y con recursos, se está produciendo una democratización del *ransomware* . Cada vez más actores menos cualificados recurren a variantes simples y de bajo coste, lo que amplía el espectro de amenazas y hace que cualquier empresa, independientemente de su tamaño o recursos, pueda convertirse en un objetivo potencial.

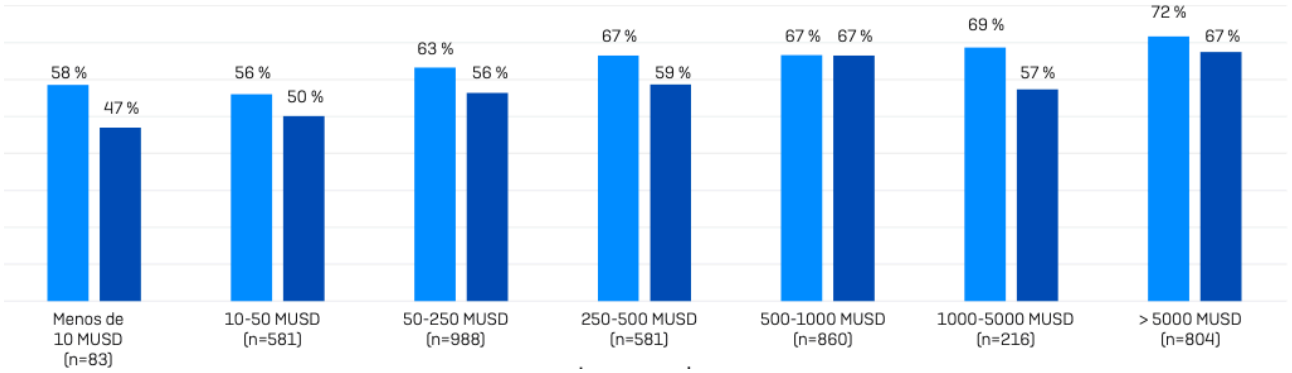

Figura 1.2: Porcentaje de organizaciones afectadas por el *ransomware* en 2024

Respecto a los ataques por industria, con algunas excepciones, las tasas de ataques de *ransomware* fueron bastante similares entre los diferentes sectores, afectando entre el 60 % y el 68 % de las organizaciones en 11 de los 15 sectores evaluados. En la investigación más reciente, los sectores con menor frecuencia de incidentes son el gobierno estatal/local (34 %) y el comercio al por menor (45 %), siendo los únicos en los que menos de la mitad de las entidades consultadas indicaron haber sido víctimas en el último año.

Llama la atención el contraste entre los dos niveles de gobierno: mientras que el sector gubernamental central o federal registró la mayor tasa de ataques (68 %), el gobierno estatal/local mostró la más baja (34 %). Esta diferencia es significativa, siendo el doble de ataques en el gobierno central respecto al local. Aun así, se observa una ligera mejora en el caso del gobierno central, ya que el año anterior este sector registró un 70 % de ataques.

Existen diversas hipótesis para explicar esta disparidad dentro del sector público. Por un lado, el aumento de los ataques al gobierno central podría tener motivaciones políticas, especialmente en un contexto de inestabilidad global. Por otro, la reducción en el sector estatal/local puede reflejar los esfuerzos realizados el año anterior para mejorar su resiliencia frente al *ransomware* . También es posible que los atacantes hayan optado por dirigir sus esfuerzos hacia objetivos con mayor capacidad de pago, alejándose de las administraciones locales, que suelen tener recursos más limitados para pagar rescates.

Según el informe obtenido por sophos [62], en el último año, se han producido otros cambios notables a nivel de sector:

- Se ha reducido el índice individual más alto de ataques registrado, que ha pasado del 80 % (educación primaria y secundaria) al 69 % (gobierno central/federal).
- El sector de la educación ya no presenta los dos índices de ataque más elevados: este año se sitúan en el 66 % (educación superior) y el 63 % (educación primaria y secundaria), frente al 79 % y al 80 %, respectivamente, del año pasado.
- El sector sanitario es uno de los cinco sectores que registraron un aumento del índice de ataques en el último año, pasando del 60 % al 67 %.
- El sector de TI, telecomunicaciones y tecnología ya no ostenta el índice de ataques más bajo: un 55 % de las organizaciones fueron víctimas en el último año, lo que supone un aumento con respecto al 50 % registrado en 2023.

Estos ajustes evidencian que las entidades que han fortalecido sus protocolos de ciberseguridad a través de actualizaciones, capacitación y respaldo de datos han logrado disminuir de manera notable su vulnerabilidad al *ransomware* ; al mismo tiempo, la expansión de los ataques a sectores que anteriormente estaban menos impactados señala que el peligro ya no se centra en un solo ámbito, lo que favorece una distribución más equilibrada de los esfuerzos defensivos; además, el aumento de incidentes en sectores como la salud y la tecnología de la información subraya la necesidad de mantener continuamente las salvaguardias en todas las áreas, resaltando la relevancia de una mejora constante y adaptable en las tácticas de seguridad.

1.3.2 Causas raíz de los ataques

El 99 % de las entidades que sufrieron ataques de *ransomware* pudieron determinar el origen del problema. Por segundo año seguido, la utilización de vulnerabilidades continuó siendo la forma de acceso más habitual, siguiendo una tendencia parecida a la que se vio en la investigación de 2023.

Un 34 % de las personas encuestadas señaló que el ataque se originó a través de técnicas que utilizan el correo electrónico. En esta categoría, la mayoría de los ataques se iniciaron con correos electrónicos dañinos (los cuales incluían enlaces o archivos adjuntos que descargaban software malicioso), en una proporción casi el doble en comparación con los ataques que comenzaron mediante phishing. Aunque ambos enfoques están conectados, es fundamental señalar que el phishing se emplea principalmente para robar credenciales de acceso y, por lo general, representa una etapa inicial en ataques más amplios que buscan violar identidades.

En la siguiente figura 1.3 podemos ver cómo se distribuyen esos porcentajes en los años 2023 y 2024.

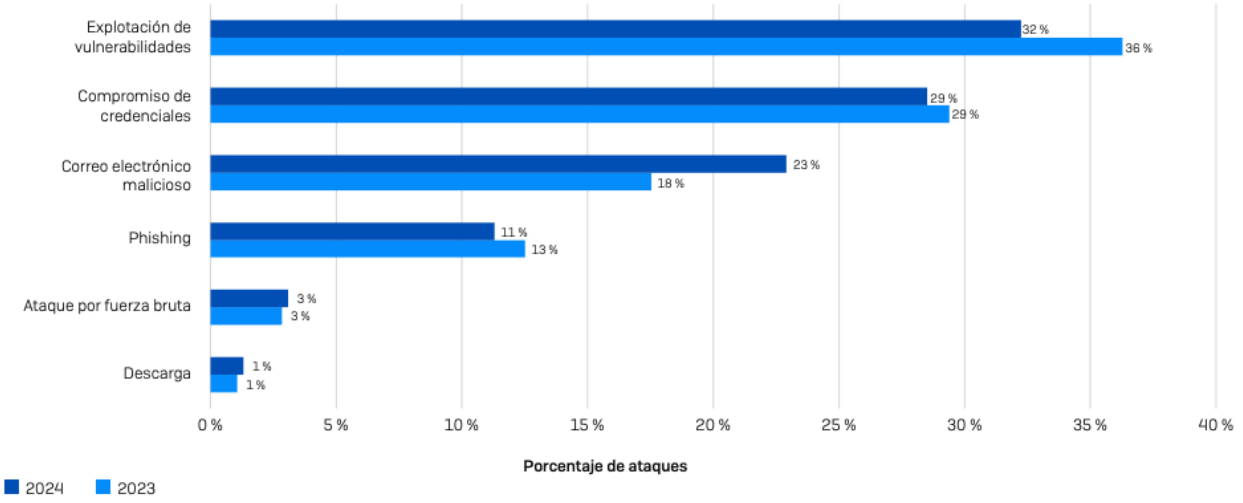

Figura 1.3: Porcentaje de ataques

Desde una perspectiva técnica, localizar el origen preciso del ataque facilita el cierre de vulnerabilidades, fortalece los aspectos débiles y previene futuras incursiones semejantes. En este sentido, esta información es importante porque alimenta de manera directa el desarrollo de modelos de detección más efectivos: comprender cómo y por dónde se produce la intrusión permite elegir las características del conjunto de datos más relevantes, modificar el pretratamiento y capacitar modelos que identifiquen patrones verdaderos de conducta maliciosa.

Además, al identificar la causa principal, las empresas tienen la capacidad de implementar acciones correctivas y preventivas más efectivas, como segmentar redes, ajustar configuraciones de cortafuegos o capacitar al personal para que reconozca campañas de phishing. Esto está vinculado a la etapa de mitigación que discutiremos en la sección final del documento, subrayando la idea de que los modelos de aprendizaje automático no solo son útiles para identificar ataques, sino también para entender y evitar su inicio.

1.4 Entorno de desarrollo y herramientas utilizadas

En este apartado se describen las principales plataformas y herramientas de software empleadas a lo largo de este proyecto, tanto para el desarrollo y ejecución de los modelos de detección de *ransomware* como para la redacción, planificación y coordinación de las tareas del TFG.

1.4.1 Google Colab

Google Colab [24] ha sido la plataforma principal que se ha utilizado para la **ejecución y entrenamiento** de los modelos de aprendizaje profundo y aprendizaje automático. Gracias a su infraestructura basada en la nube, permite el acceso a recursos de cómputo avanzados, como GPU y TPU, sin necesidad de disponer de hardware local. Esto facilita la realización de experimentos con distintos tamaños de lotes, arquitecturas y parámetros, así como la compartición y reproducibilidad del código.

Figura 1.4: Logo Colab [24]

Dado que los modelos no han requerido muchos recursos de cómputo salvo un par de excepciones, esta herramienta me ha permitido ejecutar cada uno de ellos en un tiempo razonable, sin la necesidad de disponer de material computacional externo.

1.4.2 Overleaf

Overleaf [48] se ha utilizado para la **redacción colaborativa** de la memoria del TFG. Este editor en línea colaborativo basado en la nube que se utiliza para escribir, editar y publicar documentos, basado en LaTeX [36] ofrece un entorno integrado de compilación y control de versiones, lo que ha permitido trabajar simultáneamente en el documento, gestionar referencias bibliográficas con BibTeX y mantener un formato uniforme y profesional a lo largo de todo el trabajo.

Figura 1.5: Logo overleaf [48]

1.4.3 Microsoft Teams

Microsoft Teams [65] es la aplicación más sofisticada de mensajería para su organización. Se trata de un espacio de trabajo pensado para la colaboración en tiempo real y la comunicación, las reuniones, el uso compartido de archivos y aplicaciones, e incluso para los ocasionales emoji. Todo en un único lugar, en equipo, y con todo a disposición de todos. Ha servido como la herramienta principal para la **comunicación y coordinación** con el tutor del TFG.

Figura 1.6: Logo Microsoft Teams [65]

A través de sus canales y reuniones virtuales, se ha planificado el trabajo realizado, se han discutido avances, y se han registrado actas de seguimiento. Asimismo, Teams [65] facilitó el intercambio y visionado de archivos.

1.4.4 Microsoft Word

Microsoft Word [45] es un software para procesamiento de textos desarrollado por Microsoft [44] desde 1983 hasta la actualidad. Está incluido en el paquete de aplicaciones Microsoft Office, como parte del software de suscripción en línea Microsoft 365 y Works hasta su descontinuación. Se ha empleado para la **elaboración de la planificación** del proyecto y la documentación de reuniones. Se ha organizado el calendario de trabajo, así como los avances que se realizaban en el TFG. Ha servido también como hoja a sucio donde se iban realizando todas las explicaciones y análisis llevados a cabo a más adelante, antes de incluirlos en la memoria.

Figura 1.7: Logo Microsoft Word [45]

1.4.5 Python

Python [52] es un lenguaje de alto nivel de programación interpretado cuya filosofía hace hincapié en la legibilidad de su código. Se trata de un lenguaje de programación multiparadigma, ya que soporta parcialmente la orientación a objetos, programación imperativa y, en menor medida, programación funcional. Es un lenguaje interpretado, dinámico y multiplataforma.

Administrado por Python Software Foundation [53], posee una licencia de código abierto, denominada Python Software Foundation License. [54] Python [52] se clasifica constantemente como uno de los lenguajes de programación más populares.

Figura 1.8: Logo Python [52]

La decisión de utilizar Python en el desarrollo de este Trabajo de Fin de Grado es debido a las numerosas ventajas que ofrece como lenguaje de programación, especialmente en el contexto del análisis de datos, la inteligencia artificial y la ciberseguridad. Python al ser un lenguaje de alto nivel, interpretado y dinámico, permite escribir código de manera más rápida y clara, facilitando la implementación de soluciones complejas sin necesidad de preocuparse por detalles de bajo nivel como la gestión de memoria.

En este trabajo, esto ha sido particularmente beneficioso, ya que me ha dado la oportunidad de emplear una mezcla de programación imperativa y orientada a objetos para estructurar de forma efectiva el código relacionado con el preprocesamiento de datos, la creación de modelos y la visualización de resultados. La capacidad de implementar también algunos principios de programación funcional, como el uso de funciones lambda o expresiones de mapeo, ha facilitado tareas que se repiten y ha aumentado la claridad del código.

Capítulo 2

Objetivos y Alcance

2.1 Objetivos

El objetivo principal de este estudio es analizar el impacto del *ransomware* en sistemas informáticos y redes, evaluando sus resultados sobre un dataset con más de 20.000 muestras y proponiendo optimizaciones que incrementen su precisión en la detección, clasificación por categorías y atribución por familias de *malware*.

Además, se plantean los siguientes objetivos específicos:

- Investigar y clasificar las principales técnicas de detección de *ransomware* , incluyendo métodos basados en firmas, análisis de comportamiento y modelos de aprendizaje automático.
- Reproducir los modelos del dataset de 2024 [8] y la evaluación de los modelos MLP, CNN, LSTM, CNN-LSTM, CNN-Bi-LSTM y Bi-LSTM-GN tal como aparecen en el paper de referencia.
- Examinar estrategias de mitigación y respuesta, identificando las mejores prácticas para contener y recuperar sistemas afectados.
- Comparar los resultados originales con los modelos optimizados obtenidos.
- Analizar la importancia de las características mediante un modelo Random Forest y un estudio de ablación, para identificar qué atributos del dataset aportan más valor a cada modelo.
- Elaborar una guía de buenas prácticas orientada a la prevención del *ransomware* en entornos corporativos y personales.
- Realizar una revisión bibliográfica y estado del arte, recopilando información sobre *ransomware* , técnicas de detección y estrategias de mitigación.
- Analizar ataques reales y su impacto, estudiando casos relevantes para identificar patrones y metodologías de ataque.
- Optimizar los modelos MLP y CNN, ajustando hiperparámetros y arquitecturas para mejorar mínimamente su rendimiento.
- Comparar y validar técnicas de detección, contrastando su efectividad en términos de precisión y rendimiento.

2.1.1 Tareas a realizar

Para alcanzar los objetivos establecidos en este estudio, llevaré a cabo una serie de tareas organizadas en seis fases principales: **Carga y preprocesamiento de datos**, **Reproducción de los modelos**, **Optimización de modelos**, **Análisis de importancia de características** y **Documentación y comparación**. A continuación, se describen en detalle cada una de estas tareas:

- **Carga y preprocesamiento de datos**
  - Se parte del dataset 2024 [8] y se cargan los datos en la plataforma de Google Colab por medio de Google Drive.
  - Se describen los campos, se realiza limpieza y normalización sobre los datos.
- **Reproducción de los modelos del paper [26]**
  - Implementar los modelos de machine learning según la especificación original.
  - Ejecutar esos modelos en la detección, clasificación y atribución por familias en Google Colab [24].
- **Optimización de modelos**
  - Ajustar hiperparámetros del MLP (número de neuronas, tasa de aprendizaje, épocas) para mejorar la precisión.
  - Modificar la arquitectura CNN (capas, filtros, tamaños de kernel) y reentrenar para obtener ganancias de rendimiento.
  - Aplicar técnicas similares al modelo Bi-LSTM-GN, refinando su configuración.
- **Análisis de importancia de características y ablación**
  - Entrenar un Random Forest y extraer la característica del dataset [8] que determinase las features.
  - Realizar pruebas de ablación: eliminar selectivamente columnas clave y medir el impacto en cada uno de los tres modelos.
- **Documentación y comparación**
  - Redactar en el Capítulo8 los detalles de las modificaciones de cada modelo: qué se cambió y por qué.
  - En el Capítulo 6, presentar tablas y gráficas que comparen los resultados originales vs. los optimizados, resaltando mejoras en precisión, pérdida y eficiencia.

Capítulo 3

Metodología

En este capítulo se describe la metodología utilizada para alcanzar los objetivos planteados en el estudio. La estrategia empleada en este estudio va a ser la de Cross Industry Standard Process for Data Mining [16]. Se trata de un modelo estándar abierto del proceso que describe los enfoques comunes que utilizan los expertos en minería de datos. Se fundamenta en un enfoque práctico, enfocado en la replicación de hallazgos, la alteración y optimización de modelos previos, así como en su análisis metódico en diferentes grupos de datos.

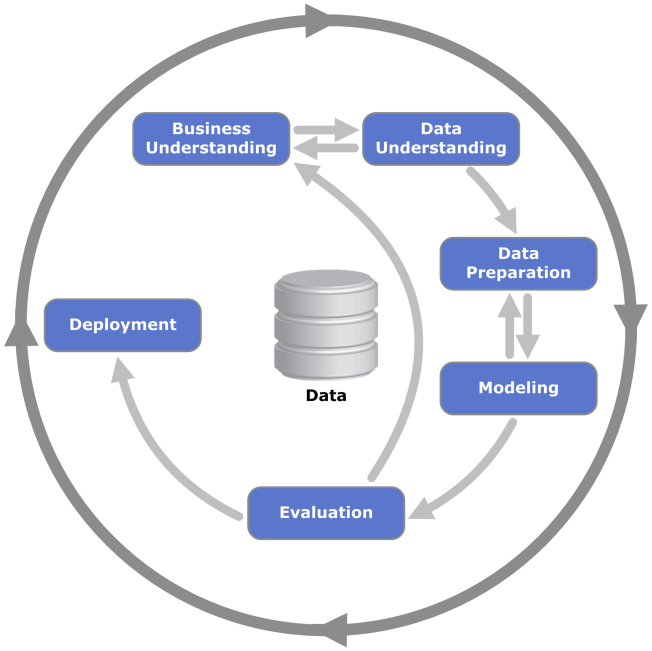

Figura 3.1: Cross Industry Standard Process for Data Mining [16]

La metodología CRISP-DM [16] es comúnmente empleada en iniciativas de análisis de datos y extracción de información. Este enfoque ofrece un marco cíclico y versátil que consta de seis etapas clave, las cuales se han ajustado al ámbito de evaluación y optimización de modelos para identificar y clasificar *ransomware* .

A continuación, se describe la aplicación de cada etapa:

- **Comprensión del negocio:** Se reconoció como el problema más importante la creciente peligrosidad del *ransomware* , así como la urgencia de implementar sistemas automáticos para la detección y clasificación,

utilizando métodos de aprendizaje automático y profundo. El propósito del proyecto se enfocó en estudiar los modelos actuales, perfeccionarlos y medir su eficacia en diversas situaciones y conjuntos de datos.

- **Comprensión de los datos:** Se examinaron detenidamente dos colecciones de información: la primera, originada del documento fundamental (OMM-2022)[17], y la segunda, una recopilación de muestras del propio paper. Se investigó su organización, clases de variables, calidad de la información y distribución de categorías. Esto facilitó la identificación de posibles inconvenientes de desbalance y ruido, así como la definición de criterios para la selección de características.

- **Preparación de los datos:** Se llevó a cabo un proceso de preparación que se ajustó a las exigencias de cada actividad (detección, clasificación y asignación de familia), aunque manteniendo una estructura similar. Este proceso abarcó la depuración, conversión, normalización y codificación de las características. Con el fin de evitar repeticiones, en el texto únicamente se describen las variaciones entre los preprocesamientos efectuados para cada método.

- **Modelado:**Se llevaron a cabo seis tipos de modelos (MLP, CNN, LSTM, CNN-LSTM, CNN-Bi-LSTM y Bi-LSTM-GN), comenzando con la reproducción de la estructura inicial del artículo de referencia y luego realizando modificaciones para mejorar su rendimiento. Se emplearon métodos como la optimización de hiperparámetros, la regularización y cambios en la disposición de las capas. Todos los modelos se crearon en Python, debido a su efectividad en el análisis de datos, su claridad en la sintaxis, y su compatibilidad con diversas plataformas.

- **Evaluación:** Todos los modelos fueron analizados utilizando métricas estándar (exactitud, recuperación, puntuación F1, tablas de confusión), tanto en la tarea de detección binaria como en la de clasificación múltiple. Se implementó el mismo proceso de evaluación para asegurar consistencia en la comparación. Además, se examinaron los resultados a través de gráficos que muestran la tendencia de la pérdida y la precisión a lo largo del entrenamiento.

- **Despliegue y conclusiones:** Aunque no se llevó a cabo una implementación en el entorno productivo, se registraron y examinaron los resultados obtenidos para sacar conclusiones sobre la eficacia de cada modelo, las repercusiones de los ajustes efectuados y las características más significativas. Estos descubrimientos podrían ser fundamentales para investigaciones futuras o para su aplicación en escenarios reales.

3.1 Fases y duración

Las fases y duración prevista de cada una (Semana inicial y final) son las que se detallan en el cuadro 3.1.

SIni	SFin	FIni	FFin	Tarea
1	2	27 de enero	9 de febrero	Búsqueda y selección de bibliografía y datasets
3	5	10 de febrero	1 de marzo	Revisión y análisis de estudios previos sobre *malware*
6	8	2 de marzo	22 de marzo	Preprocesamiento de datos para los seis enfoques
9	11	23 de marzo	12 de abril	Implementación de modelos de detección y clasificación
12	14	13 de abril	3 de mayo	Implementación de modelo de atribución
15	17	4 de mayo	22 de mayo	Evaluación de los modelos y análisis comparativo
18	20	23 de mayo	10 de junio	Redacción del TFG y elaboración de gráficas/tablas
21	21	10 de junio	19 de junio	Revisión, corrección y preparación de la defensa

Cuadro 3.1: Fases y calendario del desarrollo del proyecto.

De acuerdo con lo establecido en la guía del Trabajo Fin de Grado, se exige que el estudiante invierta unas 300 horas aproximadamente. Para satisfacer esta necesidad, se ha creado un plan detallado que abarca

desde finales de enero hasta principios de junio, dividido en diferentes etapas con objetivos específicos. Esta planificación se presenta en la Tabla 3.1, donde se muestran las semanas asignadas a cada tarea, además de sus fechas de inicio y finalización.

Durante este lapso, la dedicación ha sido continua y se ha repartido a lo largo de un periodo de alrededor de cuatro meses de trabajo efectivo. Con una estimación de entre 15 y 18 horas por semana, dependiendo de la carga de trabajo de cada etapa, se logra cumplir con las horas requeridas según el plan académico. En las etapas iniciales, como la recopilación de bibliografía o el análisis de antecedentes, la dedicación fue más ligera (aproximadamente entre 10 y 12 horas semanales), mientras que en momentos críticos como el desarrollo de modelos, el preprocesamiento y la redacción final, la intensidad se incrementó notablemente, con semanas que alcanzaron hasta 20 horas de trabajo.

Además, esta planificación ha permitido subdividir el proyecto en bloques temáticos claramente definidos: investigación del estado del arte, análisis de datos, implementación de modelos de aprendizaje automático y redacción final del informe. Esta organización no solo ha facilitado una gestión más eficaz del tiempo, sino también una mejor disposición de los contenidos y los resultados obtenidos.

3.2 Costes

Aunque se trata de un proyecto académico, es importante realizar una estimación aproximada de los recursos que se habrían necesitado si este Trabajo de Fin de Grado se hubiera desarrollado en un entorno profesional. Para ello, se han considerado los costes más relevantes, como el tiempo dedicado (gasto de personal) y el uso de equipos informáticos, aplicando un prorrateo estimado.

3.2.1 Gastos de personal

Se estima que el desarrollo del proyecto ha requerido aproximadamente 300 horas de trabajo, incluyendo fases de documentación, investigación, desarrollo, pruebas, redacción de la memoria y preparación de la defensa. Suponiendo una tarifa de 15 €/hora (equivalente a una beca técnica o práctica académica), el coste de personal sería:

$$300 horas \times 15€/hora = 4{,}500€ \tag{3.1}$$

3.2.2 Gastos equipo informático

El proyecto se ha desarrollado utilizando un ordenador personal con una configuración media (CPU multinúcleo, 8 GB de RAM, GPU dedicada) cuyo coste de adquisición se estima en 1.200 €. Considerando un uso prorrateado durante la duración del proyecto (6 meses), el coste imputado sería:

$$(1{,}200€/36 meses) \times 6 meses \approx 200€ \tag{3.2}$$

En este caso, no se han requerido licencias de software de pago ni servidores externos, ya que se han utilizado herramientas gratuitas y de código abierto como Python, Jupyter Notebook, TensorFlow/Keras y Google Colab.

Concepto	**Coste estimado**
Gastos de personal	4.500 €
Prorrateo equipo	200 €
Licencias / software	0 €
Total	**4.700 €**

Cuadro 3.2: Presupuesto total estimado del proyecto

Capítulo 4

Marco Conceptual

En este apartado se exponen y explican las ideas principales que son esenciales para entender el trabajo realizado durante este proyecto. Comprender adecuadamente estos términos es crucial, ya que forman la base teórica que sustenta el análisis, la valoración y las sugerencias de mejora que se presentan.

A lo largo del capítulo se abordarán temas relacionados con el funcionamiento de los modelos de aprendizaje automático, las técnicas más comunes de preprocesamiento de datos, así como conceptos específicos relacionados con la detección y mitigación de *ransomware* . Además, se explicarán las métricas utilizadas para evaluar los modelos y otros elementos necesarios para entender el desarrollo y los resultados obtenidos.

Antes de seguir hablando sobre el *ransomware* voy a empezar definiéndolo: es un tipo de *malware* que bloquea el acceso a los archivos de sus víctimas o los cifra a cambio de un rescate para recuperar los datos bloqueados o cifrados [68]. Con la invención de las técnicas de ofuscación, se ha vuelto cada vez más difícil detectar sus nuevas variantes, complicando aún más los procesos de defensa. Identificar la categoría y la familia exactas de un *malware* es fundamental para preparar respuestas efectivas ante posibles ataques y para poder minimizar el impacto.

Es por ello que se ha convertido en una de las amenazas más relevantes en el ámbito de la ciberseguridad debido a su capacidad para cifrar archivos y exigir un rescate a cambio de su recuperación. A lo largo de los años, la aparición de nuevas variantes de *ransomware* , muchas de ellas altamente ofuscadas, ha dificultado su detección mediante técnicas tradicionales basadas en firmas o coincidencia de patrones.

Ante este escenario, el uso de técnicas de aprendizaje automático y, especialmente, de aprendizaje profundo, ha cobrado gran relevancia en la detección y clasificación de este tipo de *malware*. Estos enfoques permiten analizar grandes volúmenes de datos y detectar patrones complejos que pasan desapercibidos para los métodos convencionales.

Los enfoques tradicionales basados en aprendizaje automático no han logrado detectar ni clasificar con precisión las variantes avanzadas de *ransomware* ofuscado, debido a las limitaciones de las técnicas de detección basadas en firmas y coincidencia de patrones existentes. Sin embargo, los modelos basados en aprendizaje profundo han demostrado ser útiles tanto en la detección como en la clasificación de este tipo de amenazas, al permitir un análisis más exhaustivo y detallado del *ransomware* ofuscado.

A pesar de los avances en este campo, la mayoría de las investigaciones se han centrado en la detección y no tanto en la atribución a familias específicas de *ransomware* , lo que deja un área de mejora importante. Este proyecto tiene como objetivo abordar precisamente este desafío, proponiendo un enfoque de clasificación de múltiples clases que aprovecha el potencial del aprendizaje profundo. En concreto, se plantea el uso de una arquitectura basada en redes de memoria a corto y largo plazo bidireccionales, combinadas con la técnica de

normalización de grupos (Bi-LSTM-GN), con el fin de detectar y clasificar variantes de *ransomware* con alta precisión.

4.1 Contexto

El *ransomware* tiene una historia que se remonta a finales de la década de 1980, cuando los cibercriminales comenzaron a utilizar técnicas de cifrado para exigir pagos en efectivo a través de servicios postales. Uno de los primeros ejemplos documentados fue el *ransomware* AIDS, creado en 1989, donde las víctimas debían enviar 180 dólares a un apartado postal en Panamá para recuperar el acceso a sus sistemas. [26]

Sin embargo, este tipo de ataque no tuvo una gran notoriedad hasta 2009. La situación cambió con la introducción de Bitcoin en 2010, lo que impulsó significativamente los delitos en internet. Las criptomonedas brindaban a los delincuentes una forma segura y anónima de recibir pagos, evitando el rastreo de transacciones y dificultando la persecución legal, ya que podían realizar pagos sin que nadie supiese quiénes eran. [7]

El *ransomware* más conocido de la década de 2010 fue CryptoLocker, que apareció en 2013 y atacaba sistemas operativos Windows. Este *ransomware* utilizaba claves criptográficas avanzadas, empleando un par de claves pública y privada para cifrar y descifrar los archivos de la víctima [47]. Su impacto fue masivo, afectando a una gran cantidad de usuarios y estableciendo un precedente para futuros ataques de *ransomware* .

En 2017, surgió WannaCry, una de las versiones más devastadoras del *ransomware* . Este ataque afectó a más de 300.000 sistemas en múltiples países, causando interrupciones en infraestructuras críticas y empresas de todo el mundo [64].

El aumento del *ransomware* persistió, y durante la primera parte de 2022 se registraron 10.666 nuevas variantes, de acuerdo con un estudio de FortiGuard en 2023 [21]. Este crecimiento se atribuye, en gran medida, a la expansión del modelo *ransomware* -as-a-Service (RaaS), que proporciona a los ciberdelincuentes un fácil acceso a variaciones alteradas y camufladas de *ransomware* , lo que hace más sencillo su manejo y propagación sin requerir habilidades avanzadas de programación.

A través del tiempo, los científicos han creado diversas metodologías para identificar y categorizar el *malware*, mientras que las compañías de antivirus comerciales han confiado sobre todo en la detección que se basa en las firmas. Este enfoque implica extraer las firmas de los archivos ejecutables a través de un análisis estático y conservarlas en una base de datos. Cuando se revisa un archivo que se considera sospechoso, se comparan sus firmas con las que ya están guardadas para verificar si es dañino o inofensivo. Aunque este procedimiento es veloz y efectivo para reconocer *malware* ya conocido, tiene limitaciones notables en la identificación de nuevas variantes, ya que los delincuentes digitales pueden evitarlo usando métodos de ofuscación. [5].

Para sortear estas restricciones, se ha creado un método de detección de *malware* que se centra en el comportamiento, donde un software se categoriza como seguro o perjudicial según sus actos dentro del sistema. Este método examina las llamadas al sistema, las invocaciones de API, los cambios en archivos, los registros y la actividad de red, lo que facilita la clasificación de un programa en función de su comportamiento. Su operativa se sustenta en tres etapas clave:

- **Extracción del comportamiento:** Se identifican y registran las acciones del programa en el sistema, como llamadas al sistema, invocaciones de API, modificaciones de archivos, registros y actividad en la red.
- **Generación de propiedades:** A partir de los datos extraídos, se generan características que describen el comportamiento del software, permitiendo su análisis y clasificación.
- **Implementación de modelos de aprendizaje automático:** Se aplican modelos de machine learning para analizar las propiedades generadas y determinar si una aplicación es segura o maliciosa.

Este método es más efectivo contra *malware* recién descubierto, ya que detecta patrones de comportamiento en lugar de depender únicamente de su código fuente. Incluso si el *malware* es modificado, su funcionalidad

maliciosa sigue siendo detectable. Por ello, la mayoría de las variantes recientes de software malicioso son identificadas mediante este enfoque.

Sin embargo, la detección basada en el comportamiento también presenta desafíos. Algunos tipos de *malware* están diseñados para detectar entornos protegidos (como máquinas virtuales o entornos de prueba) y modificar su ejecución para evitar ser clasificados como maliciosos [33]. Además, aunque se ha avanzado considerablemente en la detección, la clasificación de *ransomware* ha recibido menos atención, lo que representa una barrera para una mitigación y prevención más efectiva. Una clasificación precisa de las familias de *ransomware* permitiría desarrollar estrategias más específicas para combatir cada tipo de amenaza.

4.2 Tipos de

Dentro del ecosistema del *ransomware* , se pueden identificar dos tipos principales: **crypto *ransomware*** y **locker *ransomware*** , cada uno con características y niveles de peligrosidad distintos. Y de manera más secundaria el **Scareware** [26].

4.2.1 Crypto *ransomware*

El crypto *ransomware* opera cifrando los archivos y datos de la víctima, impidiendo su acceso a menos que se pague un rescate a los atacantes para obtener la clave de descifrado. Este tipo de *ransomware* es especialmente dañino, ya que la recuperación de los archivos es prácticamente imposible sin la clave, lo que obliga a las víctimas a recurrir a copias de seguridad (si las tienen) o, en el peor de los casos, a aceptar la pérdida de su información.

4.2.2 Locker *ransomware*

Por otro lado, el locker *ransomware* bloquea o deshabilita el acceso a la computadora de la víctima sin cifrar los archivos almacenados en ella. En comparación con el crypto *ransomware* , este tipo de ataque es menos peligroso, ya que no compromete directamente los datos del usuario. La eliminación de la infección permite restaurar el acceso al dispositivo sin alterar la información almacenada. Además, si el dispositivo de almacenamiento (generalmente un disco duro) se traslada a otro equipo en funcionamiento, los datos pueden recuperarse incluso si el virus persiste en el sistema original. Debido a esta vulnerabilidad, el locker *ransomware* tiene menos éxito al exigir el pago de un rescate, ya que las víctimas pueden restaurar el acceso a sus archivos sin necesidad de cumplir con las demandas de los atacantes.

4.2.3 Scareware

Otro tipo de *ransomware* es el scareware, diseñado para intimidar a las víctimas y obligarlas a pagar un rescate. Este tipo de *malware* se presenta comúnmente como mensajes falsos que pretenden provenir de autoridades gubernamentales, alegando que el usuario ha cometido algún delito y debe pagar una multa para evitar consecuencias legales. Algunas variantes, como el leakware, aumentan la presión psicológica al amenazar con exponer información privada o sensible de la víctima, utilizando la intimidación y la vergüenza como herramientas de coacción.

4.3 Algoritmos de aprendizaje automático (ML)

Con el progreso en las estrategias de aprendizaje automático, numerosos investigadores han comenzado a emplearlas para reconocer *ransomware* . Los algoritmos de identificación de *ransomware* que utilizan aprendizaje automático son capaces de modelar patrones de datos más sofisticados en comparación con los métodos tradicionales basados en firmas [56]. Esto les da la capacidad de identificar exitosamente nuevas variaciones de

malware antiguo, incluso aquel que no se había descubierto previamente [22]. En líneas generales, existen tres categorías de algoritmos de aprendizaje automático: **Supervisados**, **No supervisados** y los **Semisupervisados**.

4.3.1 Supervisados

En los modelos de aprendizaje supervisado, se desarrolla un modelo predictivo utilizando datos ya clasificados con el fin de estimar correctamente las etiquetas de la información nueva que ingresa [71].

Este conjunto de datos de entrenamiento incluye entradas y salidas correctas, que permiten al modelo aprender con el tiempo. El algoritmo mide su precisión a través de la función de pérdida, ajustando hasta que el error se haya minimizado lo suficiente. El *Random Forest* es un modelo de este tipo que veremos más adelante y se emplea para tareas de clasificación y predicción. El término "$Forest$" se refiere a un grupo de árboles de decisión independientes, que posteriormente se combinan para minimizar la variabilidad y generar estimaciones de datos más precisas [30].

Es un desafío para los modelos detectar *ransomware* que nunca se ha identificado o que no pertenece a familias conocidas con las que se los ha entrenado. Sin embargo, este tipo de modelos en general tienen menos desafíos y pueden detectarlo con mayor facilidad.

4.3.2 No supervisados

Por el contrario, el aprendizaje no supervisado utiliza datos no etiquetados. Esto quiere decir que descubren agrupaciones de datos sin necesidad de la intervención humana [27].

Algunos de los enfoques de este tipo de aprendizaje serían el agrupamiento, la asociación y la reducción de dimensionalidad. En cuanto a las agrupaciones, destacan las siguientes [27]:

- **Agrupación exclusiva y superpuesta:** La clasificación en clústeres exclusivos es un método de agrupación que establece que un dato solamente puede pertenecer a un clúster. A esto también se le conoce como agrupamiento "$rigido$". Un ejemplo de este tipo de agrupación es el algoritmo K-means. La técnica de agrupamiento K-means es un ejemplo típico de un método de clasificación que separa los datos en K grupos, donde K indica cuántas agrupaciones hay, basándose en la proximidad a los centroides de cada grupo. Los datos que se encuentran más cerca de un centroide particular se colocan en la misma categoría. Un número mayor de K sugiere agrupaciones más reducidas y detalladas, en tanto que un número menor de K resultará en agrupaciones más amplias y menos detalladas.

- **Agrupación jerárquica:** La organización en jerarquías, que también recibe el nombre de análisis de agrupamiento jerárquico (HCA), es un método de agrupamiento sin supervisión que puede clasificarse de dos maneras: aglomerativo o divisivo.

  El método aglomerativo es visto como un enfoque de abajo hacia arriba. En este proceso, los puntos de datos se consideran inicialmente como grupos individuales y luego se combinan de manera sucesiva basándose en su similitud hasta formar un solo grupo.

- **Agrupación probabilística:** Un modelo probabilístico es un enfoque no supervisado que contribuye a abordar desafíos relacionados con la estimación de densidad o el agrupamiento "suave". En la agrupación basada en probabilidades, los datos se agrupan según la probabilidad de su pertenencia a una distribución específica. El modelo de mezcla gaussiana (GMM) se encuentra entre los métodos más comunes de agrupamiento probabilístico.

4.3.3 Semisupervisados

En este algoritmo, los datos etiquetados y no etiquetados se combinan con el aprendizaje semisupervisado durante la fase de entrenamiento[35].

Los enfoques de aprendizaje semisupervisado son de gran importancia en contextos donde conseguir una cantidad adecuada de datos etiquetados resulta extremadamente complicado o caro, mientras que es más sencillo obtener grandes volúmenes de datos sin etiquetar. En estos casos, ni las técnicas de aprendizaje completamente supervisadas ni las no supervisadas ofrecerán respuestas satisfactorias [29].

El aprendizaje semisupervisado se fundamenta en ciertas premisas respecto a los datos no etiquetados que se utilizan para formar el modelo y la manera en que los datos de distintas categorías se conectan entre sí. Una condición esencial del aprendizaje semisupervisado (SSL) es que los ejemplos no etiquetados que se emplean en el entrenamiento del modelo deben tener relación con la tarea para la que se está desarrollando dicho modelo.

En términos más específicos, el SSL exige que la distribución p(x) de los datos de entrada incluya información sobre la distribución posterior p(y|x), es decir, la probabilidad condicionada de que un punto de datos particular (x) pertenezca a una clase específica (y). Por ejemplo, si se utilizan datos no etiquetados para formar un clasificador de imágenes que distinga entre fotos de gatos y perros, el conjunto de datos utilizado para entrenar debe incluir imágenes de ambos, pero las imágenes de caballos o motocicletas no aportarán nada útil. [29].

4.3.4 Aprendizaje profundo (DL)

El aprendizaje profundo, una nueva área dentro de la inteligencia artificial y un tipo de aprendizaje automático, ha incrementado su popularidad y se ha convertido en una técnica fundamental de aprendizaje automático en diversos sectores. Estos modelos hacen uso de redes neuronales artificiales y aprenden a partir de múltiples capas ocultas y ejemplos previos. Cuentan con varias capas de neuronas artificiales cuyos pesos se modifican de manera constante para alcanzar los resultados deseados. Este proceso de ajuste se lleva a cabo para garantizar que el optimizador pueda reducir al mínimo la pérdida. La pérdida se refiere al error en las predicciones de la red neuronal y se puede determinar mediante una función de pérdida.

Las redes neuronales recurrentes, las percepciones multicapa y las redes neuronales convolucionales son los modelos de aprendizaje automático más reconocidos. Las arquitecturas de los modelos de aprendizaje profundo son complejas y cuentan con numerosas capas de procesamiento, lo que les permite aprender características más complejas de forma automática y con diferentes niveles de abstracción, además de gestionar datos de alta dimensión. Se requiere más investigación para evaluar de manera adecuada el aprendizaje profundo, el cual ha mostrado resultados muy precisos en varios ámbitos, pero su implementación en la detección de *ransomware* , la clasificación por categorías y la asignación de familias aún es limitada [20].

La diferencia clave entre el aprendizaje automático y el aprendizaje profundo radica en la configuración de la red neuronal que utilizan. Los enfoques tradicionales de aprendizaje automático, que son menos complejos, utilizan redes neuronales básicas que tienen una o dos capas de procesamiento. En contraste, los modelos de aprendizaje profundo utilizan tres o más capas, frecuentemente llegando a tener cientos o miles de capas para su entrenamiento.

Los modelos de aprendizaje supervisado necesitan datos de entrada organizados y etiquetados para producir resultados confiables, mientras que los métodos de aprendizaje profundo pueden funcionar con aprendizaje no supervisado. Gracias al aprendizaje no supervisado, estos modelos son capaces de identificar características, funciones y relaciones que requieren para generar resultados precisos a partir de datos sin procesar y no estructurados. Además, tienen la capacidad de analizar y mejorar sus resultados para optimizar la precisión [28].

Capítulo 5

Soluciones Existentes

En este capítulo se va a abordar el estudio de las soluciones que han llevado a cabo otros investigadores acerca de las técnicas y los enfoques en la detección del *ransomware* , con el objetivo de comprender la problemática actual y las maneras de llevarlo a cabo.

Varios estudios han analizado diferentes métodos y estrategias en la investigación para identificar *ransomware* . Con el fin de entender el estado actual de las soluciones presentadas, llevamos a cabo un examen detallado de la literatura, reconociendo las motivaciones clave y las cuestiones tratadas en cada investigación [26].

En nuestro análisis, examinamos las estrategias utilizadas para la detección de *ransomware* , evaluando el conjunto de datos empleado, las características extraídas o seleccionadas, y los modelos de aprendizaje automático entrenados. Además, recopilamos información sobre la precisión reportada en cada estudio, las limitaciones identificadas y la efectividad de las soluciones propuestas.

Un aspecto clave en nuestra revisión es la clasificación multiclase, determinando si cada investigación ha considerado la diferenciación entre múltiples tipos de *ransomware* o si se ha limitado a una clasificación binaria (malicioso/benigno). Este análisis nos permite identificar vacíos en la literatura y proponer mejoras en los métodos de detección y clasificación de *ransomware* .

5.1 Machine Learning

El aprendizaje automático ha sido ampliamente utilizado en la detección de *ransomware* , con diversos enfoques propuestos en la literatura. Según las investigaciones que se citan en el paper [26]: Zhang, Liu y Jiang [70] introdujeron un método basado en la evaluación de relevancia blanda para analizar la relación entre las características y las etiquetas de *malware*. Utilizando el conjunto de datos Microsoft Malware Classification Challenge (BIG 2015) y modelos como Naïve Bayes (NB), Árbol de Decisión (DT), Random Forest (RF) y Support Vector Machine (SVM), lograron una precisión del 98,8 %.

Por otro lado, Khan y otros [20] aplicaron un enfoque innovador basado en la secuenciación de ADN digital, utilizando algoritmos como Regresión Lineal (LR), RF, NB y Optimización Mínima Secuencial (SMO), alcanzando una precisión del 87,9 % en un conjunto de datos compuesto por 582 muestras de *ransomware* y 942 muestras benignas.

En otro estudio relevante, Ficco [40] propuso la técnica Alpha-Count, un método de promedio ponderado que integra sistemas de detección de intrusiones, antivirus, entropía de archivos y análisis de tráfico de red. Su modelo basado en redes neuronales profundas (DNN), entrenado con 10.634 muestras maliciosas y 2.000 benignas, obtuvo una precisión del 93,28 %.

Asimismo, Poudyal y Dasgupta [50] investigaron la actividad del *ransomware* mediante el análisis de llamadas del sistema, tráfico de red y modificaciones en archivos. Entrenaron modelos de aprendizaje automático

como SVM, LR y RF, con SVM logrando una precisión del 99,72 %.

Mail, Ab Razak y Ab Rahman [42] presentaron un sistema de sandbox en la nube que emplea algoritmos de ML (RF, J-48 y NB), obteniendo una precisión del 99,8 % en un conjunto de datos de 9.600 muestras de *malware*. De manera similar, Ganfure y otros (2023) desarrollaron RTrap, una herramienta dinámica de detección de *ransomware* en entornos controlados, que alcanzó una precisión del 99,8 % con modelos DT, RF y SVM.

Finalmente, Molina y colaboradores [58] propusieron una técnica para la atribución de familias de *ransomware* basada en el análisis del comportamiento previo al ataque. Recolectaron 129.500 muestras de *malware* de VirusTotal y VirusShare entre 2010 y 2019, de las cuales 19.499 pertenecían a 21 familias de *ransomware* . Se implementaron modelos como Bernoulli BN, K-Nearest Neighbors (KNN), Artificial Neural Networks (ANN), Long Short-Term Memory (LSTM) y RF, con este último alcanzando una precisión del 94,92 %.

Estos estudios reflejan la efectividad de las técnicas de aprendizaje automático en la detección y clasificación de *ransomware* , con enfoques que abarcan desde modelos tradicionales hasta redes neuronales avanzadas.

5.2 Deep Learning

Por otro lado, los algoritmos de aprendizaje profundo han demostrado una efectividad considerable en la detección de *malware*. Zhang, Wang y Zhu [70] propusieron una técnica dual basada en Redes Generativas Antagónicas (GAN) para distinguir entre archivos cifrados y no cifrados, obteniendo una precisión del 98,1 % en los conjuntos de datos KDD99, SWaT y WADI.

Aslan y Yilmaz [12] utilizaron modelos de redes neuronales profundas como AlexNet y ResNet152 en los conjuntos de datos Microsoft BIG 2015, Malimg y Malevis, alcanzando precisiones del 97,78 %, 94,88 % y 96,5 %, respectivamente.

Li, Ríos y Trajković [37] adoptaron una estrategia diferente, evaluando los datos de enrutamiento del protocolo de puerta de enlace fronteriza (BGP) utilizando modelos como LightGBM, CNN y RNN, logrando una precisión del 64,74 %.

Darem y otros [3] introdujeron un algoritmo de detección de *malware* adaptativo basado en el conjunto de datos Drebin para Android, alcanzando una precisión del 99,41 %.

Yazdinejad y colaboradores [2] utilizaron un enfoque basado en redes neuronales LSTM para la detección de *malware*, con una validación cruzada de 10 veces, obteniendo una precisión del 98 % al distinguir muestras maliciosas de benignas.

Hwang y otros [13] propusieron una técnica de detección de *malware* basada en redes neuronales profundas (DNN) y entrenaron su modelo con un conjunto de datos de 10.000 muestras maliciosas y 10.000 benignas. Su enfoque logró una precisión del 94 %.

Recientemente, los investigadores también han abordado la clasificación multiclase en la detección de familias de malware. Roy y Chen (2021) utilizaron un enfoque basado en BiLSTM-CRF para identificar *ransomware* y categorizar eventos anómalos, logrando una precisión del 99,87 % en la identificación de *ransomware* y del 96,5 % en la categorización de eventos anómalos.

Keyes y otros [19] presentaron EntropyLyzer, una herramienta de análisis de comportamiento basada en entropía, que clasificó 147 familias y 12 categorías de *malware* para Android con una precisión del 98,4

Lashkari y otros [6] crearon el conjunto de datos CIC-AndMal2017 para *malware* en Android y utilizaron el algoritmo KNN para identificar *malware*, alcanzando una precisión del 85,4 %, aunque el modelo tuvo dificultades con la atribución de familias, con una precisión del 27,24 %.

Por último, Rahali y otros [1] emplearon aprendizaje profundo para clasificar *malware* en 12 categorías y 191 familias, obteniendo una precisión del 93,36 %.

Estos estudios evidencian el creciente éxito de las técnicas de aprendizaje profundo en la detección y clasificación de *malware*, especialmente en el análisis de familias y categorías diversas.

5.3 CIC-MalMem-2022 Dataset

El CIC-MalMem-2022 dataset [17] es una colección de información desarrollada por el Instituto canadiense de Ciberseguridad para simular una situación tan auténtica como sea posible con software malicioso que es común en el entorno real. Está formado por spyware, *ransomware* y troyanos, y ofrece un conjunto de datos equilibrado que puede ser utilizado para evaluar sistemas de detección de *malware* oculto. Este conjunto de información emplea el modo de depuración durante el procedimiento de volcado de memoria, lo que impide que este se refleje en los volcados de memoria. Esto proporciona un ejemplo más fiel de lo que un usuario típico tendría en funcionamiento durante un ataque de *malware*.

Como se expone en diversos estudios recientes [26], la detección de *malware* ofuscado, especialmente en forma de *ransomware* polimórfico, sigue siendo un reto, debido a la naturaleza dinámica y cambiante de estas amenazas. Frente a esta problemática, el conjunto de datos CIC-MalMem-2022 [17] ofrece una base realista y actualizada para abordar estas limitaciones, ya que recopila muestras de *malware* en memoria en diferentes escenarios, permitiendo evaluar modelos capaces de generalizar más allá de firmas estáticas o características manuales.

La detección de *malware* ofuscado ha sido un desafío crítico que ha sido abordado por varios investigadores en los últimos años. Carrier [63] utilizó el marco VolMemLyzer con 26 funciones de memoria, logrando una precisión del 99 % en diversos clasificadores, como SVM, DT, RF y KNN.

Smith, Khorsandroo y Roy [61] emplearon siete algoritmos de clasificación y consiguieron una precisión del 99 % utilizando la correlación de Pearson.

Naeem y otros [25] aplicaron métodos de agrupamiento como K-Means, DBSCAN y GMM en el conjunto de datos *malware*-Exploratory y usaron siete clasificadores adicionales, alcanzando un promedio de precisión del 99 %.

Mezina y Burget [43] utilizaron Redes Neuronales Convolucionales Dilatadas (CNN dilatadas) para la clasificación de múltiples clases, logrando una precisión del 99 %.

Roy y otros [59] propusieron MalHyStack, un modelo de clasificación multiclase basado en el aprendizaje de conjuntos apilados, que alcanzó una precisión del 99,98 %, 85,04 % y 70,20 % en la detección, clasificación y atribución familiar, respectivamente.

Dang [55] utilizó el algoritmo CatBoost para clasificar *malware* ofuscado, realizando tanto clasificación binaria como multiclase. El modelo alcanzó una precisión del 99,9 % en la detección.

Dener, Oklahoma y Orman [18] llevaron a cabo una clasificación binaria utilizando diversos algoritmos como RF, DT, GBT, LR, NB, SVM lineales, MLP, redes neuronales de avance profundo y LSTM. El algoritmo LR logró la precisión más alta, alcanzando un 99,97

Al-Qudah y otros [41] propusieron un clasificador de clases basado en SVM (OCSVM) con análisis de componentes principales (PCA), alcanzando una precisión del 99,4 % en la clasificación de una clase utilizando el modelo PCA (OCC-PCA).

Abualhaj y otros [39] propusieron un parámetro métrico de distancia mejorado de la KNN con validación cruzada de K-fold para la detección de *malware*. Lograron una precisión del 99,97 % en la detección, del 82,21 % en la clasificación y del 66,93 % en la atribución de familias.

Shafin, Karmakar y Mareels [60] propusieron un enfoque basado en CNN-BiLSTM, concretamente CompactCBL (Compact CNN-BiLSTM) y RobustCBL (Robust CNN-BiLSTM) para detectar el ataque binario, su tipo y familia. Los métodos lograron una precisión del 99,96 % y el 99,92 % en la detección de ataques binarios, del 84,56 % y el 84,22 % en ataques por familia, y del 72,60 % y el 71,41 % en tipos de ataques, respectivamente.

Smith, Khorsandroo y Roy [61] utilizaron un método de conjunto apilado basado en CNN como un aprendiz base y MLP para el metaaprendizaje. Se identificó y categorizó *malware* basado en IoT mediante la identificación de características de imágenes y la detección de actividades sospechosas. Esto ayudó a clasificar familias de *malware*, y los modelos lograron una precisión del 99,01 % en la detección de *malware*.

Muchos de los métodos de detección de *ransomware* existentes se basan principalmente en características creadas a mano o firmas estáticas, lo que los hace menos adaptables a las familias de *malware* nuevas y

emergentes. Las cepas de *ransomware* polimórficas y metamórficas son particularmente difíciles de identificar utilizando métodos basados en firmas. Una gran mayoría de las investigaciones se concentran en características de comportamiento estáticas o de corto plazo, acciones de archivos o tráfico de red durante un período determinado. Sin embargo, el *ransomware* con frecuencia demuestra tendencias dinámicas que evolucionan. Los métodos que ignoran estos patrones temporales no logran capturar toda la gama de operaciones de *ransomware* , lo que quizás genere más falsos negativos. Muchos algoritmos de aprendizaje automático empleados en estudios anteriores funcionan bien en conjuntos de datos conocidos, pero sufren cuando se exponen a *malware* nuevo. Este problema es resultado del sobreajuste a patrones específicos en los datos de entrenamiento, lo que reduce la capacidad del modelo para generalizarse en diferentes familias de *ransomware* y ataques innovadores.

Nuestro proceso de análisis de la literatura nos lleva a la conclusión de que es fundamental identificar el *ransomware* ofuscado analizándolo en profundidad y clasificándolo en su categoría correcta (*ransomware* , troyano, spyware, etc.), y atribuir sus familias mediante técnicas avanzadas basadas en Deep Learning (DL) para preparar la defensa proporcionando la solución y la ruta correctas.

5.3.1 Resultado del análisis

Para entender un poco lo que se ha expuesto con anterioridad, se ofrece un análisis detallado del rendimiento de los modelos de aprendizaje automático implementados. Para evaluar sus capacidades, se llevan a cabo tres tareas de detección diferentes: la primera es la detección de *malware*, la segunda es la clasificación por categorías y la tercera es la atribución de familias. Además, se realiza un análisis comparativo con la literatura existente sobre el mismo conjunto de datos para validar el rendimiento del enfoque propuesto.

Los ensayos se llevaron a cabo en una workstation HP Z230 que cuenta con un procesador Core(TM) i7-4790 de 3,60 GHz (8 núcleos), con el sistema operativo Windows 10 Professional de 64 bits y 16 GB de memoria RAM. También se usó la plataforma Google Colab, donde se configuraron unidades de procesamiento tensor de nube (TPU) y unidades de procesamiento gráfico (GPU) para evaluar el rendimiento computacional. Para la ejecución del proyecto práctico, se optó por el conocido software Anaconda 2. 4. 2 junto con Jupyter Notebook 6. 4. 12 y bibliotecas esenciales como Sklearn, TensorFlow, Numpy y Pandas, que se utilizan para tareas de aprendizaje automático (ML) y aprendizaje profundo (DL). El conjunto de datos se segmenta en un 80 % para el entrenamiento y un 20 % para las pruebas, utilizando validación cruzada K-Fold para asegurarse de que el entrenamiento y las pruebas se efectúen en todas las clases.

El cuadro 5.1 detalla los elementos fundamentales fijados para cada uno de los modelos de aprendizaje profundo durante sus fases de entrenamiento y evaluación. Con el fin de minimizar los sesgos, los modelos de aprendizaje profundo se configuran con un tamaño de lote de 32, una tasa de aprendizaje de 0,001, empleando el optimizador Adam para la mejora del modelo y una función de entropía cruzada categórica para monitorear la pérdida y abordar problemas de clasificación de múltiples categorías.

Primero, experimentaron con la detección de *malware* entrenando modelos DL en el conjunto de datos utilizando 20, 30, 50, 100 y 150 épocas. Todos los modelos se evalúan en función del promedio.

A continuación se muestran en los cuadros 5.1 y 5.2 los parámetros que tuvieron en cuenta en el experimento sobre el conjunto de datos de CIC-MALMEN 2022, y los tiempos de entrenamiento, pérdida y precisión sobre cada uno de los modelos.

Parameters	**OMM-2022 dataset**
Batch size	32
Epochs	20,30,50,100,150
Learning rate	0.001
Loss function	Categorical cross entropy
Optimization algorithm	Adam
Normalization	Standard
Randomization	42
Number of classes	50,51,52
Cross-validation	K-Fold
Number of splits	5

Cuadro 5.1: Parámetros del conjunto de datos OMM-2022

Model	**Average time (Minutes)**	**Average loss**	**Average accuracy**
CNN	41	0.0011	99.99 %
MLP	18	0.0007	99.99 %
LSTM	74	0.0010	99.99 %
CNN-LSTM	61	0.0011	99.97 %
CNN-BiLSTM	95	0.0012	99.97 %
GN-BiLSTM	131	0.0002	99.99 %

Cuadro 5.2: Tiempo promedio de entrenamiento, pérdida y precisión en el conjunto de datos OMM-2022.

En cuanto a la precisión, el tiempo de entrenamiento y la pérdida, se observó que los modelos de aprendizaje profundo implementados presentaron un rendimiento sobresaliente en ambas fases de entrenamiento, en el conjunto de datos OMM-2022.

El CNN logró una precisión del 99,99 % con una pérdida de 0,0011, completando el entrenamiento en 41 minutos.

El MLP obtuvo una precisión del 99,99 % con una pérdida de 0,0007, con un tiempo de entrenamiento de 18 minutos.

El LSTM alcanzó una precisión del 99,99 % con una pérdida de 0,0010 en 74 minutos de entrenamiento.

El modelo CNN-LSTM logró una precisión del 99,97 % con una pérdida de 0,0011, completando el entrenamiento en 61 minutos.

El modelo CNN-BiLSTM alcanzó una precisión del 99,97 % con una pérdida de 0,0012 en 95 minutos de entrenamiento para el conjunto de datos OMM-2022. Finalmente, el modelo propuesto, el GN-BiLSTM, alcanzó una precisión del 99,99 %.

El rendimiento de los modelos en el conjunto de datos OMM-2022,fue excelente, con todos los modelos alcanzando precisiones cercanas al 99,99 %. Los modelos GN-BiLSTM, CNN y MLP lograron una precisión sobresaliente del 99,99 %, mientras que los modelos CNN-LSTM y CNN-BiLSTM tuvieron una precisión ligeramente inferior del 99,97 %. Aunque la diferencia en precisión es mínima, esta variación destaca la robustez del modelo GN-BiLSTM, que mantiene una mayor precisión a pesar de su diseño más complejo.

En los siguientes capítulos analizaremos cómo afectan esos modelos al conjunto de datos de 2024 [8].

Capítulo 6

Análisis del Dataset y preprocesamiento

En este capítulo se abordará el análisis del dataset que se ha escogido para el análisis, así como las técnicas de preprocesamiento de datos, normalización y categorización que se han llevado a cabo sobre este, antes de realizar la detección, clasificación y atribución por categoría de *malware*.

Para comprobar la eficacia del modelo propuesto en un entorno realista, se han empleado datos obtenidos de ejecuciones controladas en memoria, que incluyen tanto muestras benignas como maliciosas de diversas familias. Las características empleadas provienen del análisis dinámico del comportamiento, abarcando aspectos como llamadas a la API, accesos a archivos y registros del sistema.

Los resultados obtenidos demuestran que el modelo es efectivo en la detección de *ransomware* y en la realización de la clasificación categórica y facilita la atribución a familias, incluso en situaciones donde se presentan muestras no vistas anteriormente.

Este análisis abarca lo siguiente:

- **Método basado en Deep Learning:** Se propone un enfoque basado en aprendizaje profundo (DL) capaz de detectar las últimas variantes de *ransomware* ofuscado con alta precisión.
- **Clasificación multiclase:** Se desarrolla un modelo que permite detectar y clasificar el *malware* según su categoría o familia.
- **Creación de un conjunto de datos actualizado:** Se recopilan las muestras más recientes de *ransomware* para construir un conjunto de datos, entrenar modelos en él y validar su rendimiento.

6.1 Descripción del dataset

El presente dataset ha sido diseñado para el análisis de *malware*, con un enfoque especial en el *ransomware* . Como se ha detallado anteriormente, contiene un total de 21,752 muestras, distribuidas equitativamente entre archivos maliciosos (10,876) y benignos (10,876), lo que garantiza un equilibrio adecuado para su uso en modelos de aprendizaje automático y análisis estadísticos.

Al mismo dataset se le ha hecho un balanceo de los datos para poder determinar de manera más precisa las categorías de *malware*, que serían las siguientes:

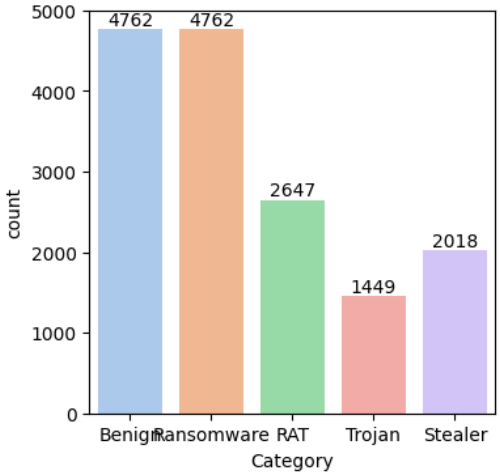

Figura 6.1: Categorías de malware

Asimismo, se ha balanceado también el dataset para mostrar las muestras de *malware* que están organizadas en 26 familias distintas, incluyendo algunas de las variantes más relevantes en ataques recientes, como Cerber, DarkSide, Dharma, GandCrab, LockBit, Maze, Phobos, REvil, Ragnar Locker, Ryuk, Shade y WannaCry como se muestran a continuación:

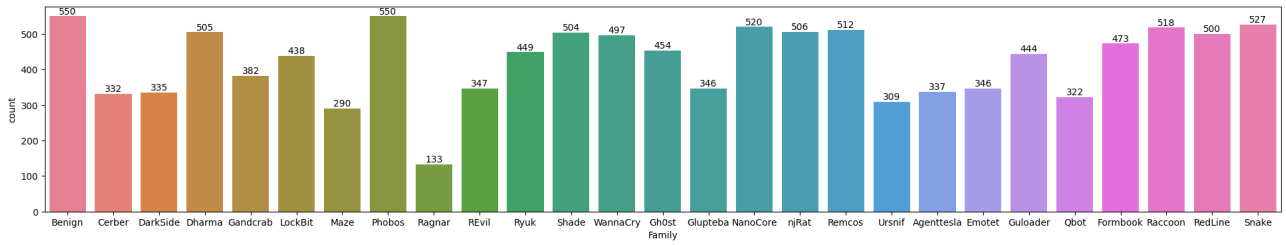

Figura 6.2: Familias de malware

El dataset consta de 77 características (features), que abarcan una amplia variedad de atributos técnicos, tales como:

- **Identificadores y metadatos del archivo**
  - `md5`, `sha1`: Hashes del archivo que permiten su identificación única.
  - `file_extension`: Extensión del archivo, útil para determinar su formato.
- **Estructura del ejecutable (PE Header)**
  - `EntryPoint`: Dirección de entrada del ejecutable.
  - `PEType`: Tipo de Portable Executable (PE).
  - `MachineType`: Tipo de arquitectura de la máquina para la que fue compilado el archivo.
  - `magic_number`: Identificador del formato del archivo.
  - `bytes_on_last_page`, `pages_in_file`: Número de bytes en la última página y cantidad de páginas en el archivo.
  - `relocations`: Cantidad de registros de reubicación.
  - `size_of_header`: Tamaño del encabezado del ejecutable.

- `min_extra_paragraphs`, `max_extra_paragraphs`: Párrafos extra mínimos y máximos requeridos por el ejecutable.
- `init_ss_value`, `init_sp_value`: Valores iniciales del segmento de pila (SS) y del puntero de pila (SP).
- `init_ip_value`, `init_cs_value`: Valores iniciales del puntero de instrucción (IP) y del segmento de código (CS).
- `over_lay_number`: Número de overlay en el archivo.
- `oem_identifier`: Identificador del fabricante OEM.
- `address_of_ne_header`: Dirección del encabezado New Executable (NE).

- **Estructura interna del PE**
  - `Magic`: Número mágico del ejecutable.
  - `SizeOfCode`: Tamaño de la sección de código.
  - `SizeOfInitializedData`, `SizeOfUninitializedData`: Tamaños de datos inicializados y no inicializados.
  - `AddressOfEntryPoint`: Dirección de entrada del código ejecutable.
  - `BaseOfCode`, `BaseOfData`: Dirección base de las secciones de código y datos.
  - `ImageBase`: Dirección base de la imagen cargada en memoria.
  - `SectionAlignment`, `FileAlignment`: Alineación de secciones y archivos en memoria.
  - `OperatingSystemVersion`, `ImageVersion`: Versión del sistema operativo y del archivo ejecutable.
  - `SizeOfImage`: Tamaño total del ejecutable en memoria.
  - `SizeOfHeaders`: Tamaño de los encabezados del ejecutable.
  - `Checksum`: Suma de verificación del ejecutable.
  - `Subsystem`: Tipo de subsistema donde se ejecutará el programa.
  - `DllCharacteristics`: Características de seguridad y configuración de la DLL.
  - `SizeofStackReserve`, `SizeofStackCommit`: Tamaño reservado y comprometido para la pila.
  - `SizeofHeapCommit`, `SizeofHeapReserve`: Tamaño reservado y comprometido para el heap.
  - `LoaderFlags`: Indicadores de carga del ejecutable.

- **Características de las secciones del ejecutable**
  - `text_VirtualSize`, `text_VirtualAddress`: Tamaño y dirección virtual de la sección de código (`.text`).
  - `text_SizeOfRawData`, `text_PointerToRawData`: Tamaño en disco y puntero a los datos en crudo de `.text`.
  - `text_PointerToRelocations`, `text_PointerToLineNumbers`: Punteros a reubicaciones y números de línea en `.text`.
  - `text_Characteristics`: Características de la sección `.text`.
  - `rdata_VirtualSize`, `rdata_VirtualAddress`: Tamaño y dirección virtual de la sección de datos (`.rdata`).
  - `rdata_SizeOfRawData`, `rdata_PointerToRawData`: Tamaño en disco y puntero a los datos en crudo de `.rdata`.

  - `rdata_PointerToRelocations`, `rdata_PointerToLineNumbers`: Punteros a reubicaciones y números de línea en `.rdata`.
  - `rdata_Characteristics`: Características de la sección `.rdata`.

- **Registros de actividad del sistema**
  - `registry_read`, `registry_write`, `registry_delete`: Número de operaciones sobre el registro de Windows.
  - `registry_total`: Total de accesos al registro.

- **Comportamiento en red**
  - `network_threats`: Indicadores de actividad sospechosa en la red.
  - `network_dns`: Número de consultas DNS realizadas.
  - `network_http`: Cantidad de peticiones HTTP detectadas.
  - `network_connections`: Número total de conexiones establecidas.

- **Actividad de procesos**
  - `processes_malicious`: Cantidad de procesos clasificados como maliciosos.
  - `processes_suspicious`: Cantidad de procesos con actividad sospechosa.
  - `processes_monitored`: Número total de procesos observados.
  - `total_processes`: Cantidad total de procesos en ejecución.

- **Interacciones con archivos**
  - `files_malicious`: Número de archivos creados/modificados con comportamiento malicioso.
  - `files_suspicious`: Número de archivos considerados sospechosos.
  - `files_text`: Número de archivos de texto manipulados.
  - `files_unknown`: Número de archivos sin clasificar.

- **Llamadas a funciones y API**
  - `dlls_calls`: Número de llamadas a librerías dinámicas (DLLs).
  - `apis`: Cantidad de funciones de la API del sistema invocadas.

- **Etiquetas de clasificación**
  - `Class`: Categoría del archivo (benign o *malware*).
  - `Category`: Tipo de *malware* (Ejemplo: *ransomware* , Trojan).
  - `Family`: Familia específica del *malware* (Ejemplo: WannaCry, REvil).

6.2 Preprocesamiento de datos

El preprocesamiento de datos es una etapa fundamental en cualquier tarea de aprendizaje profundo, ya que garantiza que los datos de entrada sean de alta calidad y estén en un formato adecuado para los modelos. En este proceso, se aplican diversas técnicas como la limpieza de datos, normalización, eliminación de valores atípicos y conversión de características, con el objetivo de reducir el ruido y mejorar la capacidad del modelo para identificar patrones relevantes. En el caso de la detección de *malware*, el preprocesamiento es crucial para transformar los datos en representaciones que faciliten la clasificación y detección de amenazas, asegurando así un entrenamiento más eficiente y preciso de los modelos implementados [51].

Antes de empezar el preprocesamiento, se va a explicar las librerías de python que se han tenido en cuenta para poder ejecutar los modelos:

6.2.1 Importación de bibliotecas

```
import pandas as pd
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import accuracy_score, confusion_matrix, ConfusionMatrixDisplay
from torch.utils.data import DataLoader, TensorDataset
from tqdm import tqdm
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
import time
import seaborn as sns
import plotly.express as px
```

- **pandas**: Utilizada para la carga, manipulación y análisis de estructuras de datos en forma de tablas (*DataFrames*). Ha sido esencial para trabajar con los archivos CSV del conjunto de datos, permitiendo tareas como limpieza, filtrado y transformación de datos.
- **numpy**: Permite trabajar de forma eficiente con arrays y realizar operaciones matemáticas sobre matrices. Se ha usado para transformar datos y realizar cálculos numéricos previos al entrenamiento.
- **torch** y **torch.nn**: Librerías del framework *PyTorch* utilizadas para construir y entrenar el modelo de red neuronal (MLP). `torch.nn` proporciona las capas, funciones de activación y herramientas necesarias para definir arquitecturas neuronales.
- **torch.optim**: Se ha utilizado para definir el algoritmo de optimización (*Adam*) encargado de minimizar la función de pérdida durante el entrenamiento del modelo.
- **torch.utils.data**: Ofrece estructuras como `TensorDataset` y `DataLoader`, fundamentales para dividir los datos en lotes (*batching*) y aplicarles aleatorización (*shuffling*) de forma eficiente.
- **sklearn.model_selection.train_test_split**: Utilizada para dividir el conjunto de datos en conjuntos de entrenamiento y prueba, garantizando una evaluación justa del modelo.
- **sklearn.model_selection.StratifiedKFold**: Aplicada para implementar validación cruzada estratificada, asegurando que la proporción de clases se mantenga en cada partición.
- **sklearn.metrics**: Se han usado funciones como `accuracy_score` y `confusion_matrix` para evaluar el rendimiento del modelo. También se ha empleado `ConfusionMatrixDisplay` para visualizar la matriz de confusión.
- **tqdm**: Librería empleada para mostrar barras de progreso en tiempo real durante el entrenamiento, lo que facilita el seguimiento de las ejecuciones largas.
- **matplotlib.pyplot** y **seaborn**: Utilizadas para la generación de gráficos y visualizaciones. *Matplotlib* ofrece gráficos básicos, mientras que *Seaborn* mejora la estética y proporciona herramientas estadísticas para la representación visual.
- **plotly.express**: Permite generar gráficos interactivos en 2D y 3D. Ha sido útil para representar visualmente la distribución de familias de *malware* o resultados de análisis exploratorios.
- **sklearn.preprocessing.StandardScaler**: Se ha utilizado para estandarizar las variables numéricas, lo cual mejora el rendimiento del modelo al normalizar los datos con media cero y desviación estándar uno.

- **time**: Librería estándar de Python usada para medir el tiempo de ejecución de fragmentos del código, especialmente durante el entrenamiento del modelo.

Posteriormente, para todos los modelos se ha realizado el siguiente tratamiento de datos:

6.2.2 Carga del dataset

Carga los datos desde un archivo CSV:

```
data = pd.read_csv(BDIR)
```

- **BDIR**: Variable que contiene la ruta del archivo CSV que se encuentra en la carpeta especificada en Google Drive.
- **pd.read_csv**: Función de pandas que lee el archivo CSV desde la ruta proporcionada en el BDIR.
- **data**: El archivo CSV se almacena en la variable `data` como un DataFrame de pandas, lo que permite manipular y analizar los datos fácilmente.

El siguiente paso es eliminar los identificadores y datos irrelevantes para el estudio.

6.2.3 Eliminación de identificadores y datos irrelevantes

```
    data = data.drop(['md5'], axis=1)
    data = data.drop(['sha1'], axis=1)
    data = data.drop(['Category'], axis=1)
    data = data.drop(['Family'], axis=1)
    data = data.drop(['PEType'], axis=1)
    data = data.drop(['file_extension'], axis=1)
```

- Elimina múltiples columnas innecesarias con data.drop(). Cada llamada a drop() elimina una columna específica basada en su nombre, con el parámetro axis=1 indicando que se están eliminando columnas (no filas).
- **md5, sha1**: Son hashes criptográficos para identificar archivos, pero no aportan información útil para detección de *malware*.
- **Category, Family**: Para este caso no las necesitaremos, ya que queremos analizar la detección.
- **PEType, fileExtension**: La extensión de archivo y el tipo de Portable Executable tampoco aportan información útil.

6.2.4 Eliminación de atributos técnicos del ejecutable

```
    data = data.drop(['MachineType'], axis=1)
    data = data.drop(['magic_number'], axis=1)
    data = data.drop(['Magic'], axis=1)
    data = data.drop(['OperatingSystemVersion'], axis=1)
    data = data.drop(['ImageVersion'], axis=1)
    data = data.drop(['Subsystem'], axis=1)
    data = data.drop(['DllCharacteristics'], axis=1)
    data = data.drop(['AddressOfEntryPoint'], axis=1)
```

- Estas características, al igual que las anteriores, no aportan mucho valor al análisis de *malware*.
- Por ejemplo, **MachineType**, **OperatingSystemVersion**, **Subsystem** y **ImageVersion** son muy variadas y no diferencian bien entre archivos benignos y *malware*.

6.2.5 Eliminación de características de bajo impacto o redundantes

```
    data = data.drop(['network_threats'], axis=1)
    data = data.drop(['text_Characteristics'], axis=1)
    data = data.drop(['rdata_Characteristics'], axis=1)
    data = data.drop(['oem_identifier'], axis=1)
    data = data.drop(['LoaderFlags'], axis=1)
```

- **network_threats** podría ser una métrica derivada de otros datos de red y, por lo tanto, redundante.
- **text_Characteristics** y **rdata_Characteristics** pueden ser poco útiles o contener información que ya está representada en otras columnas.
- **oem_identifier**: Código del fabricante, probablemente irrelevante.
- **LoaderFlags**: No suele usarse en análisis de *malware*.

6.2.6 Eliminación de registros de direcciones y valores internos

```
    data = data.drop(['rdata_PointerToRelocations'], axis=1)
    data = data.drop(['init_ss_value'], axis=1)
    data = data.drop(['init_ip_value'], axis=1)
    data = data.drop(['init_cs_value'], axis=1)
    data = data.drop(['text_PointerToLineNumbers'], axis=1)
    data = data.drop(['rdata_PointerToLineNumbers'], axis=1)
    data = data.drop(['text_PointerToRelocations'], axis=1)
```

- Son valores internos de cómo está estructurado el ejecutable en memoria.
- No aportan información clave para determinar si un archivo es *malware* o no.

6.2.7 Transformación de datos

Tras haber eliminado una serie de características que realmente van a aportar algo más negativo que positivo al análisis, el siguiente paso es transformar los datos:

```
# Load the dataset
df = data
# Convert hexadecimal values to numeric
df['EntryPoint'] = df['EntryPoint'].apply(lambda x: int(x, 16))
df['bytes_on_last_page'] = df['bytes_on_last_page'].apply(lambda x: int(x, 16))
df['pages_in_file'] = df['pages_in_file'].apply(lambda x: int(x, 16))
df['relocations'] = df['relocations'].apply(lambda x: int(x, 16))
df['size_of_header'] = df['size_of_header'].apply(lambda x: int(x, 16))
df['min_extra_paragraphs'] = df['min_extra_paragraphs'].apply(lambda x: int(x, 16))
df['max_extra_paragraphs'] = df['max_extra_paragraphs'].apply(lambda x: int(x, 16))
#df['init_ss_value'] = df['init_ss_value'].apply(lambda x: int(x, 16))
df['init_sp_value'] = df['init_sp_value'].apply(lambda x: int(x, 16))
#df['init_ip_value'] = df['init_ip_value'].apply(lambda x: int(x, 16))
#df['init_cs_value'] = df['init_cs_value'].apply(lambda x: int(x, 16))
df['over_lay_number'] = df['over_lay_number'].apply(lambda x: int(x, 16))
#df['oem_identifier'] = df['oem_identifier'].apply(lambda x: int(x, 16))
df['address_of_ne_header'] = df['address_of_ne_header'].apply(lambda x: int(x, 16))
df['SizeOfCode'] = df['SizeOfCode'].apply(lambda x: int(x, 16))
df['SizeOfInitializedData'] = df['SizeOfInitializedData'].apply(lambda x: int(x, 16))
df['SizeOfUninitializedData'] = df['SizeOfUninitializedData'].apply(lambda x: int(x, 16))
#df['AddressOfEntryPoint'] = df['AddressOfEntryPoint'].apply(lambda x: int(x, 16))
df['BaseOfCode'] = df['BaseOfCode'].apply(lambda x: int(x, 16))
```

```
df['BaseOfData'] = df['BaseOfData'].apply(lambda x: int(x, 16))
df['ImageBase'] = df['ImageBase'].apply(lambda x: int(x, 16))
df['SectionAlignment'] = df['SectionAlignment'].apply(lambda x: int(x, 16))
df['FileAlignment'] = df['FileAlignment'].apply(lambda x: int(x, 16))
df['SizeOfImage'] = df['SizeOfImage'].apply(lambda x: int(x, 16))
df['SizeOfHeaders'] = df['SizeOfHeaders'].apply(lambda x: int(x, 16))
df['Checksum'] = df['Checksum'].apply(lambda x: int(x, 16))
df['SizeofStackReserve'] = df['SizeofStackReserve'].apply(lambda x: int(x, 16))
df['SizeofStackCommit'] = df['SizeofStackCommit'].apply(lambda x: int(x, 16))
df['SizeofHeapCommit'] = df['SizeofHeapCommit'].apply(lambda x: int(x, 16))
df['SizeofHeapReserve'] = df['SizeofHeapReserve'].apply(lambda x: int(x, 16))
#df['LoaderFlags'] = df['LoaderFlags'].apply(lambda x: int(x, 16))
df['text_VirtualSize'] = df['text_VirtualSize'].apply(lambda x: int(x, 16))
df['text_VirtualAddress'] = df['text_VirtualAddress'].apply(lambda x: int(x, 16))
df['text_SizeOfRawData'] = df['text_SizeOfRawData'].apply(lambda x: int(x, 16))
df['text_PointerToRawData'] = df['text_PointerToRawData'].apply(lambda x: int(x, 16))
#df['text_PointerToRelocations'] = df['text_PointerToRelocations'].apply(lambda x: int(x,
    16))
#df['text_PointerToLineNumbers'] = df['text_PointerToLineNumbers'].apply(lambda x: int(x,
    16))
df['rdata_VirtualSize'] = df['rdata_VirtualSize'].apply(lambda x: int(x, 16))
df['rdata_VirtualAddress'] = df['rdata_VirtualAddress'].apply(lambda x: int(x, 16))
df['rdata_SizeOfRawData'] = df['rdata_SizeOfRawData'].apply(lambda x: int(x, 16))
df['rdata_PointerToRawData'] = df['rdata_PointerToRawData'].apply(lambda x: int(x, 16))
#df['rdata_PointerToRelocations'] = df['rdata_PointerToRelocations'].apply(lambda x: int(
    x, 16))
#df['rdata_PointerToLineNumbers'] = df['rdata_PointerToLineNumbers'].apply(lambda x: int(
    x, 16))
```

- Para cada columna, están utilizando una función lambda que convierte los valores hexadecimales a decimales.
- La función lambda que se usa es:

$$lambda\ x : int(x, 16) \tag{6.1}$$

- `int(x, 16)`: Convierte el valor `x` de hexadecimal (base 16) a decimal (base 10).
- Esto es importante porque los valores en algunas columnas están representados en formato hexadecimal (por ejemplo, direcciones de memoria o tamaños de memoria en los archivos ejecutables), y para su análisis y uso posterior, es necesario convertirlos a formato numérico.
- Aplicación de la conversión en varias columnas:
  - `EntryPoint`: El punto de entrada de un ejecutable (dirección en memoria).
  - `bytes_on_last_page`: Número de bytes en la última página de un archivo.
  - `pages_in_file`: Número de páginas en un archivo ejecutable.
  - `relocations`: Número de reubicaciones (relocations) en el archivo.
  - `size_of_header`: Tamaño del encabezado.
  - `min_extra_paragraphs` y `max_extra_paragraphs`: Pueden ser parámetros relacionados con la estructura interna del archivo ejecutable.
  - Y muchas otras, como `BaseOfCode`, `BaseOfData`, `SizeOfCode`, `SizeOfInitializedData`, `SizeOfUninitializedData`, `Checksum`, etc.

```
    categorical_columns = df.select_dtypes(include=['object', 'category']).columns
    categorical_columns = [ 'Class']
    df = pd.DataFrame(df)
```

La codificación categórica es el proceso de convertir valores categóricos en valores numéricos y debe convertirse antes de que se alimenten a los modelos ML o DL. Por lo general, se utilizan dos enfoques: codificación one-hot y simplemente sustituir los datos de categoría con valores numéricos. La codificación one-hot es útil cuando los valores son mínimos y no es adecuada para tareas de clasificación y atribución de familias.

El dataset incluye tres columnas idénticas que deben transformarse en datos numéricos. La primera columna es **Clase**, que tiene 2 valores únicos (Malicioso y Benigno) y se convierte en 0 y 1.

La segunda columna, **Categoria**, contiene cinco valores únicos (Benigno, Troyano, *ransomware* , Stealer y RAT).

La tercera columna, **Familia**, contiene 27 entradas únicas.
Todos los valores categóricos en ambas columnas (Categoría y Familia) se convierten en codificación mecanizable.

- Selecciona las columnas del DataFrame `df` que tienen un tipo de datos categórico, es decir, aquellas que son de tipo `object` o `category`.
- Las columnas de tipo `object` son generalmente las que contienen texto o cadenas de caracteres.
- Las columnas de tipo `category` son aquellas que contienen variables que toman un número limitado de valores distintos, generalmente usadas para datos cualitativos.
- `columns`: Extrae los nombres de las columnas que cumplen con el criterio anterior (columnas de tipo `object` o `category`).
- Después de ejecutar esta línea, `categorical_columns` contendrá una lista con los nombres de las columnas categóricas.

```
from sklearn.preprocessing import LabelEncoder
# Initialize the label encoder
label_encoder = LabelEncoder()

# Apply label encoding to each categorical column
for col in categorical_columns:
    df[col] = label_encoder.fit_transform(df[col])
```

- `LabelEncoder`: Es una clase de la biblioteca scikit-learn que se utiliza para convertir variables categóricas en valores numéricos. Es particularmente útil cuando las categorías son etiquetas que no tienen un orden intrínseco.
- `categorical_columns`: Esta lista contiene las columnas que se definieron previamente como categóricas. En este caso, solo contiene la columna `'Class'`.
- `fit_transform(df[col])`: Este método de `LabelEncoder` realiza dos pasos:
  - `fit()`: Encuentra las categorías únicas en la columna especificada.
  - `transform()`: Convierte cada categoría en un número único, asignando a cada categoría un número entero. Por ejemplo, si las categorías son "A", "B" y "C", podrían ser codificadas como 0, 1, y 2, respectivamente.
- El resultado de `fit_transform()` se asigna nuevamente a la columna `df[col]`, reemplazando los valores originales categóricos con los nuevos valores numéricos.

Esto es sumamente importante, ya que los modelos de aprendizaje automático (como árboles de decisión, redes neuronales, etc.) no pueden trabajar con datos categóricos directamente; necesitan datos numéricos. El Label Encoding convierte las categorías en números para que los algoritmos puedan procesarlas.

6.2.8 Normalización

La normalización, también conocida como escalado de características, es un proceso que ajusta los valores de las variables dentro de un rango determinado. En los conjuntos de datos utilizados en esta investigación, las características presentan diferentes escalas numéricas, con algunas variables que tienen rangos altos y otras más reducidos.

Para mejorar la eficiencia y estabilidad de los modelos de aprendizaje automático y aprendizaje profundo, se aplica una transformación que ajusta los valores a un rango estándar de 0 a 1. En este caso, se utiliza la técnica StandardScaler, la cual reescala los datos de manera que cada característica tenga una media de 0 y una desviación estándar de 1, asegurando así una distribución uniforme y optimizada para el entrenamiento de los modelos.

En este paso los datos serán normalizados de la siguiente manera:

```
from sklearn.preprocessing import MinMaxScaler, StandardScaler
# Create a Min-Max scaler instance
scaler = StandardScaler()
# Select the columns you want to scale (exclude the target variable if needed)
columns_to_scale = df.columns[:-1]  # You can select specific columns here

# Fit the scaler on the selected columns and transform the data
df[columns_to_scale] = scaler.fit_transform(df[columns_to_scale])
```

El código aplica una técnica de normalización utilizando StandardScaler de la biblioteca sklearn.preprocessing. Este método ajusta los valores de las características seleccionadas para que tengan una media de 0 y una desviación estándar de 1, lo que mejora el rendimiento de los modelos de aprendizaje automático al evitar que ciertas características dominen sobre otras debido a diferencias en la escala de los valores.

Para ello, primero se crea una instancia de StandardScaler, luego se seleccionan las columnas a normalizar (excluyendo la variable objetivo) y, finalmente, se ajustan y transforman los datos con fitTransform(). Como resultado, todas las características seleccionadas quedan escaladas de manera uniforme, facilitando el entrenamiento de los modelos y asegurando que los algoritmos que dependen de la magnitud de las variables, como redes neuronales o modelos basados en distancia, funcionen de manera óptima.

6.2.9 Separación de características (features) y etiquetas (labels)

La separación de características y etiquetas es un paso fundamental en el preprocesamiento de datos para el entrenamiento de modelos de aprendizaje automático.

En este proceso, el conjunto de datos (df) se divide en dos partes:

```
    # Separate features and labels
X = df.iloc[:, :-1].values  # Features
y = df.iloc[:, -1].values   # Class labels
```

Características (X): Son las variables de entrada que el modelo utilizará para hacer predicciones. En el código, se seleccionan todas las columnas excepto la última (df.iloc[:, :-1].values), ya que se asume que la última columna contiene la etiqueta o la variable objetivo.

Etiquetas (y): Representan la salida esperada o la clase objetivo que el modelo debe predecir. En este caso, se extrae la última columna del conjunto de datos (df.iloc[:, -1].values).

Esta separación es esencial porque los modelos de aprendizaje automático aprenden patrones a partir de las características (X) y se entrenan para predecir las etiquetas (y).

Para el caso de los modelos Bi-LSTM y Bi-LSTM-GN se aplicó como parte de una estrategia de mejora del rendimiento lo siguiente:

```
from sklearn.preprocessing import LabelEncoder
from imblearn.over_sampling import SMOTE
le = LabelEncoder()
y_encoded = le.fit_transform(y)
smote = SMOTE(random_state=42)
X_smote, y_smote = smote.fit_resample(X, y_encoded)
X_smote.shape
```

Exclusivamente en esos modelos se aplican LabelEncoder y SMOTE para mejorar el rendimiento al tratar con clases desbalanceadas. El LabelEncoder transforma las etiquetas categóricas en valores numéricos, necesarios para que el modelo pueda procesarlas correctamente. Por otro lado, SMOTE genera ejemplos sintéticos de la clase minoritaria, equilibrando el conjunto de datos y evitando que el modelo se sesgue hacia la clase mayoritaria, lo cual es especialmente importante en arquitecturas más complejas y sensibles como las Bi-LSTM.

Aquí terminaría la fase de preprocesamiento de los datos para la detección de *malware*, a continuación se explicará el preprocesamiento llevado a cabo tanto para la clasificación de *malware* como para la atribución por familias de *malware*.

6.3 Preprocesamiento - Clasificación de malware

Con el fin de evitar redundancias en la explicación del preprocesamiento, en esta sección se detallarán únicamente las diferencias específicas entre el preprocesamiento realizado para la clasificación de malware y el utilizado en la sección anterior (detección de *malware*). Dado que ambos procesos comparten una base común en la limpieza y transformación de los datos, se omite la repetición de pasos idénticos y se destacan únicamente las variaciones relevantes entre ambos enfoques.

6.3.1 Carga del dataset

```
data = pd.read_csv('balanceado.csv')
```

Esta vez la carga de datos va a ser a través de un fichero diferente, en este caso un dataset balanceado [9]. Esto es porque el dataset original [8] muestra un significativo desbalance en la distribución de las clases, lo que puede influir negativamente en la efectividad del modelo de clasificación. De esta manera, la clase **Benign** que contaba con un total de 10876 muestras, tras balancearlo cuenta con un total de 4.762 muestras , mientras que las clases asociadas a comportamientos maliciosos están repartidas de manera igualada y en menor cantidad: ***ransomware*** tiene 4.762 muestras, **RAT** cuenta con 2. 647, **Stealer** con 2.018 y **Trojan** con 1.449.

Este desequilibrio que había anteriormente entre las clases puede generar sesgos en el proceso de entrenamiento, haciendo que el modelo favorezca la clase más abundante y dificultando la identificación precisa de las clases menos representadas. Para mitigar este inconveniente y fomentar una distribución más equilibrada, se ha decidido llevar a cabo un muestreo aleatorio sobre la clase **Benign**, reduciendo su cantidad hasta que sea igual a la de la clase maliciosa que tiene más representación.

6.3.2 Eliminación de identificadores y datos irrelevantes

```
#data = data.drop(['Category'], axis=1)
```

En este caso se van a eliminar los identificadores mencionados en la anterior sección, excepto la feature de **Category**, por eso se muestra comentada en la línea de código, ya que es la que vamos a tener en cuenta en el análisis de clasificación de *malware*.

6.3.3 Transformación de datos

```
categorical_columns = df.select_dtypes(include=['object', 'category']).columns
categorical_columns =[ 'Class', 'Category']
df = pd.DataFrame(df)
```

- `categorical_columns = df.select_dtypes(include=['object', 'category'])` busca dentro del DataFrame df todas las columnas que contienen datos categóricos o de tipo texto (tipo object o category). Ya que los modelos como redes neuronales o modelos basados en árboles no pueden trabajar directamente con texto o categorías. Estas columnas necesitan ser codificadas (por ejemplo, con one-hot encoding o label encoding).
- `categorical_columns =[ 'Class', 'Category']` sobrescribe la variable anterior y especifica manualmente que las columnas categóricas a trabajar son 'Class' y 'Category'.Puede que el select_dtypes anterior no haya detectado correctamente las columnas porque tal vez están como int pero en realidad representan clases (por ejemplo, 0 = benigno, 1 = malware).
- `df = pd.DataFrame(df)` hace una copia superficial del dataframe por si se ha modificado en algún paso anterior.

6.4 Preprocesamiento - Atribución por Familias

6.4.1 Carga del dataset

Para este caso en el que se va a clasificar por atribucion de familias de *malware*, el dataset del que se va a partir es el de *familias.csv*[10].

```
    data = pd.read_csv('familias.csv')
```

La razón de utilizar este dataset es porque también están desbalanceadas las familias de *malware* en el dataset original. Es por ello que como de benigno había 10876 familias y en comparación con el resto que se sitúan en torno a 500, este desequilibrio puede causar problemas al entrenar modelos de machine learning o deep learning, ya que el modelo aprende a priorizar la clase mayoritaria (Benign) y no aprender correctamente a identificar las familias con pocas muestras, como se ha explicado anteriormente.

6.4.2 Eliminación de identificadores y datos irrelevantes

Al igual que se ha explicado antes, en este caso, como queremos atribuir por familias de *malware*, tan solo eliminaremos las características eliminadas antes exceptuando la Category y la Famliy que son las que van a interesar en esta clasificación.

```
#data = data.drop(['Category'], axis=1)
#data = data.drop(['Family'], axis=1)
```

6.4.3 Transformación de datos

```
# Assuming you have a DataFrame named 'df'
categorical_columns = df.select_dtypes(include=['object', 'category']).columns
categorical_columns =[ 'Class', 'Category', 'Family']
df = pd.DataFrame(df)
```

Misma transformación de datos que la utilizada en la clasificación de *malware*, pero en este caso en atribución por familias.

6.5 Estudio de ablación

Como se ha comprobado en secciones anteriores, el uso de las features determina en gran medida cómo se van a comportar unos modelos frente a otros.

Con el objetivo de identificar la relevancia de ciertas características en el rendimiento del modelo, se ha realizado un análisis de ablación eliminando distintas combinaciones de variables clave del conjunto de datos y observando la variación en la precisión del modelo CNN tras 20, 30 y 50 épocas de entrenamiento.

6.5.1 Análisis

Para llevarlo a cabo se ha probado con 3 algoritmos diferentes para ver qué relevancia tienen en el entrenamiento de los modelos.

El primer algoritmo utilizado es el *RandomForestClassifier*:

```
from sklearn.ensemble import RandomForestClassifier
import matplotlib.pyplot as plt
import seaborn as sns

# Entrenamos con todas las columnas
X_full = data.drop('Class', axis=1)
y_full = data['Class']

model = RandomForestClassifier(random_state=42)
model.fit(X_full, y_full)

# Obtenemos importancias
importances = model.feature_importances_
feature_names = X_full.columns # Aquí está el cambio

# Crear DataFrame
feature_importance_df = pd.DataFrame({
'Feature': feature_names,
'Importance': importances
}).sort_values(by='Importance', ascending=False)

# Visualización
plt.figure(figsize=(10, 6))
sns.barplot(data=feature_importance_df.head(20), x='Importance', y='Feature')
plt.title('Top 20 Features más importantes')
plt.tight_layout()
plt.show()
```

RandomForest genera múltiples árboles de decisión autónomos, cada uno basado en un grupo aleatorio de variables y datos (bagging). Después, combina los resultados de todos los árboles en un promedio.[14].

Para calcular la importancia de una variable, se calcula observando cuánto mejora la impureza (por ejemplo, el Gini impurity) cada vez que se usa esa variable en un nodo de un árbol. Si una variable aparece muchas veces y produce divisiones efectivas, se considera importante.

Los resultados obtenidos son los siguientes:

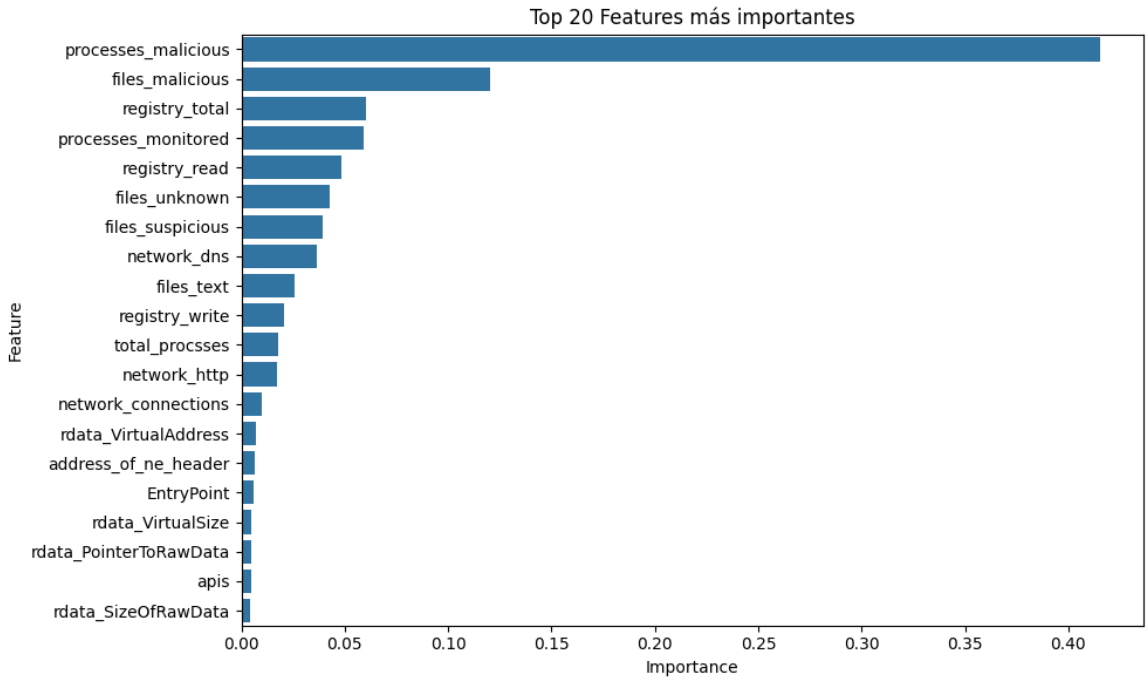

Figura 6.3: Top 20 features with RandomForestClassifier

Como se aprecia en la figura 6.3 `processes_malicious` domina con un 40 % de importancia, lo que indica que los árboles la utilizan mucho y muy temprano para dividir los datos.

Además, hay un bloque de variables con importancias entre 3 % y 8 %,que muestra que hay varias variables que ayudan al modelo, pero ninguna tanto como la principal.

Es por eso qué según este algoritmo, la variable `processes_malicious` es la más importante y más adelante se comprobará cómo afecta al modelo.

El siguiente algoritmo que se ha tenido en cuenta es el de *LGBMClassifier*:

```
from lightgbm import LGBMClassifier
import matplotlib.pyplot as plt
import seaborn as sns

# Entrenamos con todas las columnas
X_full = data.drop('Class', axis=1)
y_full = data['Class']

model = LGBMClassifier(random_state=42)
model.fit(X_full, y_full)

importances = model.feature_importances_
feature_names = X_full.columns

feature_importance_df = pd.DataFrame({
'Feature': feature_names,
'Importance': importances
}).sort_values(by='Importance', ascending=False)


plt.figure(figsize=(10, 6))
sns.barplot(data=feature_importance_df.head(20), x='Importance', y='Feature')
plt.title('Top 20 Features más importantes (LightGBM)')
plt.tight_layout()
plt.show()
```

LightGBM es un método que se fundamenta en el aumento de gradientes, pero está creado para ser veloz y eficaz. Forma árboles de decisión de manera consecutiva, donde cada árbol nuevo busca rectificar los fallos del que lo precede. Emplea un enfoque denominado crecimiento hoja por hoja, lo que frecuentemente lleva a la creación de modelos muy efectivos [38].

Para calcular la importancia puede emplear dos formas:

- **Frecuencia de uso** de la feature (cuántas veces aparece en los árboles).
- **Ganancia de información** (cuánto mejora el modelo al usarla, acumulado).

Para este caso en concreto se está utilizando la frecuencia de uso de la feature.

Los resultados obtenidos son los siguientes:

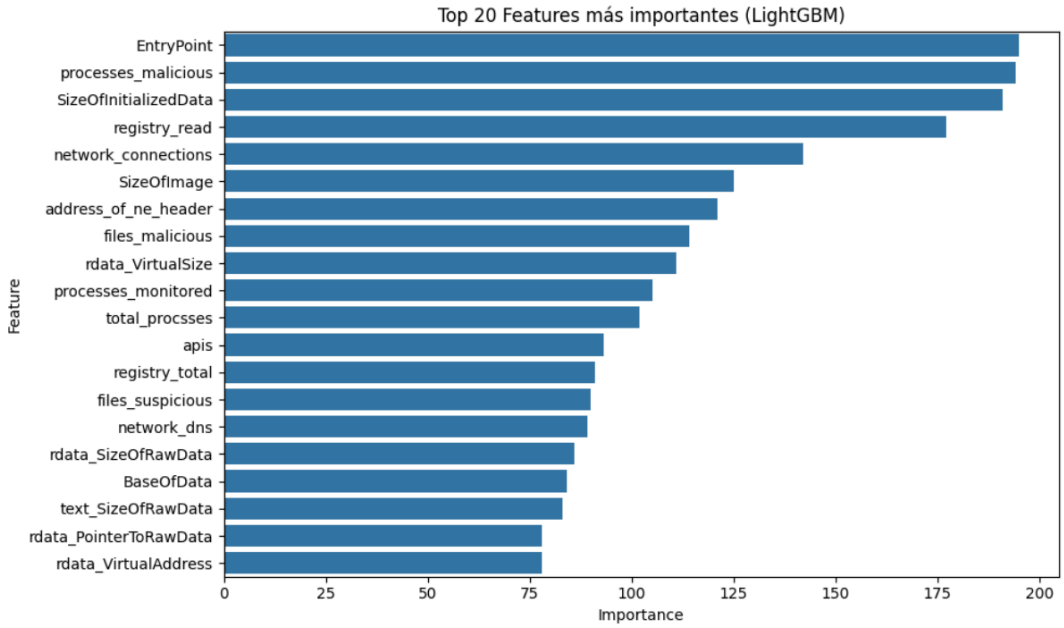

Figura 6.4: Top 20 features with LGBM Classifier

En LightGBM, cuando se mira la importancia por número de veces que se usa una feature: No mide directamente lo importante que es para mejorar la predicción. Mide cuántas veces esa feature fue utilizada en algún ”split”de los árboles.

El problema de esto es que una feature puede aparecer muchísimas veces en splits poco relevantes (que casi no ayudan a separar clases). Mientras que otra feature puede aparecer muy pocas veces, pero en splits críticos que deciden cosas muy importantes. Es por eso que hay que tener en cuenta que la **Cantidad de veces ≠ Importancia real en el modelo**.

En este caso `EntryPoint, processes_malicious, SizeOfInitializedData`, entre otros, muestran cifras significativamente altas. Esto indica que estas variables son elegidas a menudo en varios árboles y poseen un notable poder de predicción desde el punto de vista del algoritmo.

También se observan variables estructurales del ejecutable (por ejemplo, SizeOfImage o BaseOfData), lo que indica que LightGBM identifica correctamente los patrones técnicos del archivo.

Como se puede observar, la variable `processes_malicious` se vuelve a repetir como una de las más importantes.

Por último, se va a utilizar el algoritmo de *XGBoostClassifier:*

```
from xgboost import XGBClassifier
import matplotlib.pyplot as plt
import seaborn as sns

# Codificar las etiquetas
y_encoded = y_full.map({'Benign': 0, 'Malware': 1})

# Ahora entrenar
model = XGBClassifier(random_state=42, use_label_encoder=False, eval_metric='logloss')
model.fit(X_full, y_encoded)

importances = model.feature_importances_
feature_names = X_full.columns
feature_importance_df = pd.DataFrame({
'Feature': feature_names,
'Importance': importances
}).sort_values(by='Importance', ascending=False)

plt.figure(figsize=(10, 6))
sns.barplot(data=feature_importance_df.head(20), x='Importance', y='Feature')
plt.title('Top 20 Features más importantes (XGBoost)')
plt.tight_layout()
plt.show()
```

Este algoritmo también usa gradient boosting, como LightGBM, pero su estrategia de crecimiento de árboles y su sistema de regularización son más conservadores. Tiende a penalizar la complejidad del modelo para evitar sobreajuste [69].

Para calcular la importancia lo hace de la siguiente manera:

- Gain (cuánto reduce el error la variable).
- Weight (cuántas veces se ha usado).
- Cover (cantidad de datos cubiertos cuando se usa la variable).

Por defecto usa gain, los resultados son los siguientes:

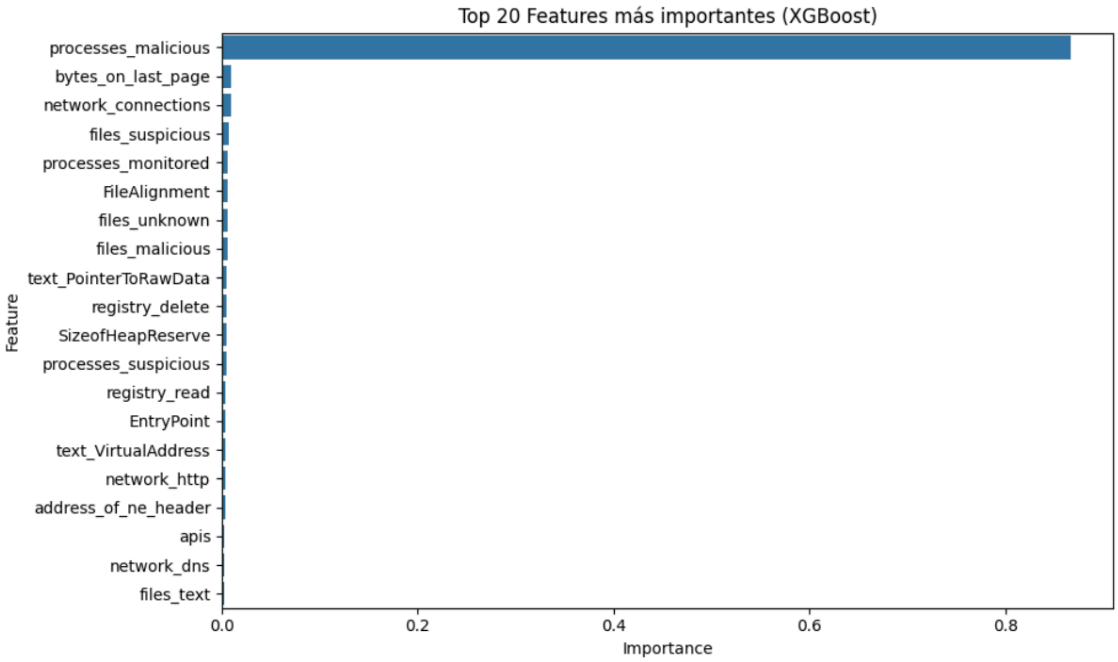

Figura 6.5: Top 20 features with XGBoost Classifier

Como se ve en la figura 6.5 `processes_malicious` tiene una importancia exagerada (0.8). Muchas variables tienen importancia 0, lo que indica que el modelo:

- Encontró que una sola variable le bastaba para clasificar la mayoría de los datos.
- O bien no entrenó bien debido a alguna limitación: clases muy desbalanceadas, correlaciones, pocos árboles o profundidad baja.

Lo que le caracteriza a este modelo también es que es muy sensible a clases no binarias (como *Malware* o *Benign*):

En cualquier caso, `process_malicious` sigue siendo la feature elegida por todos ellos.

6.5.2 Resultados

Para comprobar la veracidad de los algoritmos en cuanto a su selección de las features, se va a comprobar cómo afectan en cada uno de ellos el tenerlas o no en cuenta.

```
important_features = [
'processes_malicious'
]
# Creamos una copia sin esas columnas
df = data.drop(columns=important_features)

X = df.drop('Class', axis=1)
y = df['Class']

input_size = X.shape[1]
```

El código anterior se encarga de eliminar la feature `processes_malicious` y los resultados son los siguientes:

- Precisión del modelo CNN con 20 épocas: 91,59 % frente al 98,88 % que se obtuvo sin eliminar esa feature.

- Precisión del modelo CNN con 30 épocas: 92,09 % frente al 98,92 % original.
- Precisión del modelo CNN con 50 épocas: 93,07 % frente al 98,98 % original.

Si eliminamos solamente `files_malicious` que es lo que hace el siguiente código:

```
important_features = [
'files_malicious'
]
# Creamos una copia sin esas columnas
df = data.drop(columns=important_features)
X = df.drop('Class', axis=1)

y = df['Class']

input_size = X.shape[1]
```

Lo que se obtiene es:

- Precisión del modelo CNN con 20 épocas: 98,88 % frente al 98,88 % original.
- Precisión del modelo CNN con 30 épocas: 98,90 % frente al 98,92 % original.
- Precisión del modelo CNN con 50 épocas: 98,95 % frente al 98,98 % del original.

Si eliminamos ambas features:

```
important_features = [
'processes_malicious', 'files_malicious'
]

# Creamos una copia sin esas columnas
df = data.drop(columns=important_features)

X = df.drop('Class', axis=1)
y = df['Class']

input_size = X.shape[1]
```

El resultado es:

- Precisión del modelo CNN con 20 épocas: 91,06 % frente al 98,88 % original.
- Precisión del modelo CNN con 30 épocas: 98,99 % frente al 98,92 % original.
- Precisión del modelo CNN con 50 épocas: 99,04 % frente al 98,98 % original.

Por último, eliminando el top 20 de features más importantes:

```
    important_features = [
'processes_malicious', 'files_malicious', 'registry_total', 'registry_read',
'processes_monitored', 'files_unknown', 'files_suspicious', 'network_dns',
'files_text', 'network_http', 'registry_write', 'total_procsses',
'network_connections', 'rdata_VirtualAddress', 'address_of_ne_header',
'EntryPoint', 'rdata_VirtualSize', 'rdata_PointerToRawData', 'rdata_SizeOfRawData'

]
# Creamos una copia sin esas columnas
df = data.drop(columns=important_features)

X = df.drop('Class', axis=1)
y = df['Class']

input_size = X.shape[1]
```

Los resultados son:

- Precisión del modelo CNN con 20 épocas: 72,61 % frente al 98,88 %.
- Precisión del modelo CNN con 30 épocas: 72,09 % frente al 98,92 %.
- Precisión del modelo CNN con 50 épocas: 74,08 % frente al 98,98 %.

6.5.3 Conclusiones

Los resultados obtenidos permiten extraer las siguientes conclusiones:

- **Importancia crítica de la característica `processes_malicious`**:
  - La eliminación de esta variable provocó una caída significativa en la precisión del modelo, especialmente con 20 épocas (de 98,88 % a 91,59 %).
  - Incluso tras 50 épocas, el modelo no logra recuperar su rendimiento original (93,07 % vs. 98,98 %).

  `processes_malicious` es una característica altamente informativa y fundamental para el modelo. Su presencia ayuda a distinguir eficazmente entre muestras benignas y maliciosas.

- **Impacto moderado de la característica `files_malicious`**:
  - La eliminación de esta variable produce un impacto leve o casi nulo. La precisión del modelo apenas varía (menos de 0,1 % en la mayoría de los casos).

  Aunque `files_malicious` puede aportar información útil, su impacto es secundario. El modelo puede compensar su ausencia utilizando otras variables relacionadas.

- **Efecto combinado de eliminar ambas (`processes_malicious` + `files_malicious`)**:
  - Al eliminar ambas características, el modelo muestra una caída inicial en la precisión (91,06 % con 20 épocas), pero tras más entrenamiento (30 y 50 épocas), supera ligeramente el rendimiento original, llegando a un 99,04 %.

  El modelo puede reestructurar el aprendizaje utilizando otras características complementarias. Sin embargo, el tiempo de entrenamiento requerido aumenta. Este fenómeno puede estar relacionado con una mayor generalización o menor sobreajuste al eliminar dos variables dominantes.

- **Eliminación del Top 20 de características más relevantes**:
  - El rendimiento del modelo cae drásticamente a un 72,61 % con 20 épocas, y apenas alcanza el 74,08 % tras 50 épocas.

  Las 20 variables eliminadas constituyen el núcleo informativo del modelo. Su exclusión provoca una degradación severa del rendimiento, lo que evidencia su altísimo valor predictivo. La red neuronal no logra recuperar su capacidad discriminativa con las características restantes.

El estudio revela que ciertas variables son fundamentales para la efectividad del modelo CNN, que es el que hemos evaluado en este caso y el de cualquier otro modelo al clasificar *malware*, en particular aquellas vinculadas a la actividad de procesos y registros. Aunque algunas de estas variables podrían considerarse innecesarias o sustituidas por otras, la inclusión de todas las características principales asegura una representación más sólida del comportamiento del *malware*.

Asimismo, se nota que una eliminación cuidadosa de las características más influyentes puede mejorar la capacidad de generalización del modelo, aunque esto implique un mayor costo computacional (más ciclos de entrenamiento). Por el contrario, suprimir de manera indiscriminada las variables más significativas afecta gravemente el desempeño del sistema.

Capítulo 7

Evaluación de los modelos a estudiar

En esta sección se presenta un análisis detallado del rendimiento de los modelos de aprendizaje automático implementados. Para evaluar su efectividad, se llevan a cabo tres tareas de detección distintas: detección de *malware*, clasificación por categorías de malware y atribución de familias de *malware*. Además, se realiza un análisis comparativo con estudios previos utilizando el mismo conjunto de datos, con el objetivo de validar el enfoque propuesto.

Los experimentos se ejecutan sobre la plataforma Google Colab, aprovechando tanto unidades de procesamiento tensorial (TPU) como unidades de procesamiento gráfico (GPU) para evaluar el rendimiento computacional. Se habló la posibilidad de ejecutarlas en una máquina proporcionada por el tutor en local, pero debido a que los modelos tampoco requerían muchos recursos de cómputo, con esta plataforma se podía realizar perfectamente. La implementación se realiza en Anaconda 2.4.2 con Jupyter Notebook 6.4.12, utilizando bibliotecas clave como Scikit-learn, TensorFlow, NumPy y Pandas para las tareas de aprendizaje automático y profundo.

7.1 Evaluación de los modelos

Para el desarrollo y evaluación del modelo, el conjunto de datos se ha dividido en un 80 % para entrenamiento y un 20 % para prueba, garantizando así una separación adecuada entre los datos utilizados para ajustar los pesos del modelo y aquellos empleados para validar su capacidad de generalización. Además, se ha aplicado una validación cruzada estratificada mediante K-Fold, lo cual permite evaluar el rendimiento del modelo de forma más robusta y equitativa entre las distintas clases, especialmente en contextos con clases desbalanceadas.

Durante el entrenamiento, se han utilizado los siguientes parámetros clave: un tamaño de lote (batch size) de 32, una tasa de aprendizaje de 0.001, y el optimizador Adam, ampliamente utilizado por su eficiencia en la actualización de pesos en redes neuronales profundas. Para la función de pérdida, se ha empleado la entropía cruzada categórica, adecuada para problemas de clasificación multiclase como el presente caso [26].

El primer factor que se utiliza para medir el rendimiento de un modelo es comprobar su exactitud. Se revisa y verifica observando la exactitud y la pérdida de un modelo DL en cada época y calculando la precisión promedio o media al final para predecir la precisión del modelo:

$$\text{Accuracy} = \frac{TP + TN}{TP + FP + TN + FN} \tag{7.1}$$

El segundo factor utilizado para medir el rendimiento es la precisión. La precisión se evalúa midiendo la

proporción de positivos identificados correctamente por modelo y el número total de positivos identificados. La precisión se presenta:

$$\text{Precision} = \frac{TP}{TP + FP} \tag{7.2}$$

El tercer factor importante es la recuperación. También se denomina sensibilidad y representa la relación entre las instancias vinculadas recuperadas y el número total de instancias recuperadas. Se representa:

$$\text{Recall} = \frac{TP}{TP + FN} \tag{7.3}$$

El cuarto factor importante es el F1-Score, que se mide considerando tanto la precisión como la recuperación. Se supone que es el peso promedio de todos los valores y se presenta:

$$\text{F1-Score} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{7.4}$$

Dado que la evaluación de los modelos sigue la misma lógica y utiliza las mismas métricas en los tres casos de estudio: **detección de malware**, **clasificación por tipo de malware** y **atribución por familia**, tanto el código utilizado como la explicación de las métricas empleadas serán exactamente iguales para cada uno de ellos. Por este motivo, se presentará una única sección de evaluación, válida para todos los modelos desarrollados, evitando así repeticiones innecesarias y facilitando una visión unificada del rendimiento de los modelos aplicados en las diferentes tareas.

El código que van a implementar todos los modelos es el que se presenta a continuación:

```
# Convert data to PyTorch tensors
X = torch.tensor(X, dtype=torch.float32)
y = torch.tensor(y, dtype=torch.long)

# Define the number of folds for cross-validation
num_splits = 5  # You can adjust the number of folds as needed

# Initialize lists to store accuracy scores, training and testing loss, and times
fold_accuracies = []
train_losses = []
test_losses = []
train_times = []
test_times = []

all_train_accuracies = []
all_test_accuracies = []
all_train_losses = []
all_test_losses = []

all_true_labels = []
all_predicted_labels = []

# Initialize the cross-validator
kf = StratifiedKFold(n_splits=num_splits, shuffle=True, random_state=42)


# Specify the model hyperparameters
input_size = X.shape[1]  # Number of input features
hidden_size = 128  # Number of hidden units
num_classes = len(np.unique(y))  # Number of classes

# Loop over the folds
for fold, (train_index, test_index) in enumerate(kf.split(X, y)):
    X_train, X_test = X[train_index], X[test_index]
```

```
        y_train, y_test = y[train_index], y[test_index]

        # Create DataLoader for training and testing
        train_dataset = TensorDataset(X_train, y_train)
        test_dataset = TensorDataset(X_test, y_test)

        batch_size = 32
        train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
        test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)

        # Create an instance of the XX model
        model = XXModel(input_size, hidden_size, num_classes)

        # Define loss function and optimizer
        criterion = nn.CrossEntropyLoss()
        optimizer = optim.Adam(model.parameters(), lr=0.001)

        # Training loop
        num_epochs = 20
        train_losses_fold = []
        test_losses_fold = []
        train_accuracies = []
        test_accuracies = []

        start_time = time.time()

        for epoch in tqdm(range(num_epochs), desc=f'Fold {fold + 1}/{num_splits}'):
            model.train()
            correct_train = 0
            total_train = 0
            running_train_loss = 0.0

            for inputs, labels in train_loader:
                optimizer.zero_grad()
                outputs = model(inputs)
                loss = criterion(outputs, labels)
                loss.backward()
                optimizer.step()
                running_train_loss += loss.item()
                _, predicted = torch.max(outputs.data, 1)
                total_train += labels.size(0)
                correct_train += (predicted == labels).sum().item()

                # Append true labels and predicted labels for this batch
                all_true_labels.extend(labels.tolist())
                all_predicted_labels.extend(predicted.tolist())

            train_accuracy = correct_train / total_train
            train_losses_fold.append(running_train_loss / len(train_loader))
            train_accuracies.append(train_accuracy)

            # Evaluation on the test set
            model.eval()
            correct_test = 0
            total_test = 0
            running_test_loss = 0.0

            for inputs, labels in test_loader:
                outputs = model(inputs)
                loss = criterion(outputs, labels)
                running_test_loss += loss.item()
                _, predicted = torch.max(outputs.data, 1)
```

```
                total_test += labels.size(0)
                correct_test += (predicted == labels).sum().item()

            test_accuracy = correct_test / total_test
            test_losses_fold.append(running_test_loss / len(test_loader))
            test_accuracies.append(test_accuracy)

        # Store the accuracy of this fold
        fold_accuracies.append(test_accuracies[-1])
        train_losses.append(train_losses_fold)
        test_losses.append(test_losses_fold)

        end_time = time.time()
        train_time = end_time - start_time
        train_times.append(train_time)

        # Calculate the testing time for the fold
        start_time = time.time()
        for _ in range(len(test_loader)):
            pass
        end_time = time.time()
        test_time = end_time - start_time
        test_times.append(test_time)

        # Append accuracy and loss for this fold
        all_train_accuracies.append(train_accuracies)
        all_test_accuracies.append(test_accuracies)
        all_train_losses.append(train_losses_fold)
        all_test_losses.append(test_losses_fold)

    # Calculate and print the mean accuracy across all folds
    mean_accuracy = np.mean(fold_accuracies)
    print(f'Mean Accuracy: {mean_accuracy * 100:.2f}%')

    # Calculate and print the total test loss across all folds
    total_test_loss = sum([sum(loss) for loss in all_test_losses])

    # Calculate and print the total average test loss (across all epochs and folds)
    total_average_test_loss = total_test_loss / (num_splits * num_epochs)

    # Calculate and print the mean training and testing times across all folds
    mean_train_time = np.sum(train_times)
    mean_test_time = np.sum(test_times)
    print(f'Mean Training Time (seconds): {mean_train_time:.2f}')
    print(f'Mean Testing Time (seconds): {mean_test_time:.2f}')

    # Calculate and print the mean training loss across all epochs and folds
    mean_training_loss = np.mean([np.mean(loss) for loss in all_train_losses])
    print(f'Mean Training Loss: {mean_training_loss:.4f}')

    # Calculate and print the mean test loss across all epochs and folds
    mean_test_loss = np.mean([np.mean(loss) for loss in all_test_losses])
    print(f'Mean Test Loss: {mean_test_loss:.4f}')

    # Plot training and test accuracy and loss
    plt.figure(figsize=(12, 5))
    plt.subplot(1, 2, 1)
    mean_train_accuracies = np.mean(all_train_accuracies, axis=0)
    mean_test_accuracies = np.mean(all_test_accuracies, axis=0)
    plt.plot(range(num_epochs), mean_train_accuracies, label="Train")
    plt.plot(range(num_epochs), mean_test_accuracies, label="Test")
    plt.title("Accuracy vs. Epoch")
```

```
    plt.xlabel("Epoch")
    plt.ylabel("Accuracy")
    plt.legend()
    plt.savefig('/content/drive/MyDrive/10-Docencia/TFG/2024-2025/TFG-AdrianSanzMartin/
    datasets/Final_Dataset_without_duplicate.csvresults/XX/XX/XX_epochs_accuracy.pdf',
    format='pdf')
    plt.show()
    plt.close()
    plt.figure(figsize=(12, 5))
    plt.subplot(1, 2, 2)
    mean_train_losses = np.mean(all_train_losses, axis=0)
    mean_test_losses = np.mean(all_test_losses, axis=0)
    plt.plot(range(num_epochs), mean_train_losses, label="Train")
    plt.plot(range(num_epochs), mean_test_losses, label="Test")
    plt.title("Loss vs. Epoch")
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.legend()
    plt.savefig('/content/drive/MyDrive/10-Docencia/TFG/2024-2025/TFG-AdrianSanzMartin/
    datasets/Final_Dataset_without_duplicate.csvresults/XX/XX/XX_epochs_loss.pdf', format
    ='pdf')
    plt.show()
    plt.close()
```

Como se muestra en la Tabla 7.1, los parámetros utilizados en el modelo son los siguientes:

Parameters	**Self-created dataset**
Batch size	32
Epochs	20, 30, 50, 100, 150
Learning rate	0.001
Loss function	Categorical cross entropy
Optimization algorithm	Adam
Normalization	Standard
Randomization	42
Number of classes	46, 47, 48
Cross-validation	K-Fold
Number of splits	5

Cuadro 7.1: Parameter detail of the implemented models.

Este código implementa un modelo genérico que será utilizado por todos los modelos utilizando PyTorch para tareas de clasificación, con la aplicación de validación cruzada estratificada (StratifiedKFold). Primero, los datos se convierten en tensores, y se define una arquitectura de red según el modelo con una capa oculta y una capa de salida. A continuación, se realiza una validación cruzada utilizando 5 particiones. Para cada partición, los datos se dividen en conjuntos de entrenamiento y prueba, y el modelo se entrena usando el optimizador Adam y la función de pérdida CrossEntropyLoss.

Durante el proceso de entrenamiento, se evalúa el rendimiento del modelo en cada época, registrando métricas como la precisión, las pérdidas tanto de entrenamiento como de prueba, así como los tiempos de entrenamiento y prueba. Al finalizar la validación cruzada, se calcula y presenta la precisión media, la pérdida total de prueba y las medias de los tiempos de entrenamiento y prueba. Además, se generan gráficos que muestran la evolución de la precisión y la pérdida a lo largo de las épocas, permitiendo una visualización clara del rendimiento del modelo durante el proceso de entrenamiento. Se aplica la función de activación `ReLU` (`self.relu = nn.ReLU()`) entre ambas capas para introducir no linealidad y permitir al modelo aprender patrones más complejos.

El código descrito con anterioridad lo van a utilizar cada uno de los siguientes modelos:

7.1.1 MLP

A continuación se muestra el modelo utilizando MLP:

```
class MLPModel(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(MLPModel, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        return x
```

Este fragmento de código implementa un modelo de red neuronal multicapa (*MLP*, por sus siglas en inglés) utilizando la biblioteca **PyTorch**. La clase *MLPModel* hereda de *nn.Module*, lo que permite definir una arquitectura personalizada compuesta por capas totalmente conectadas y funciones de activación.

El constructor `__init__` recibe como parámetros:

- `input_size`: número de características de entrada.
- `hidden_size`: número de neuronas en la capa oculta.
- `num_classes`: número de clases a predecir.

Dentro del constructor, se inicializan dos capas lineales:

- `self.fc1 = nn.Linear(input_size, hidden_size)`: conecta la entrada con la capa oculta.
- `self.fc2 = nn.Linear(hidden_size, num_classes)`: conecta la capa oculta con la de salida.

Además, este modelo constituye un clasificador básico basado en perceptrón multicapa, adecuado para tareas de clasificación donde se requiera una arquitectura sencilla pero funcional.

El método `forward` define el flujo de datos a través de la red:

- Se aplica la capa `fc1`.
- Luego, la activación `ReLU`.
- Finalmente, la capa `fc2`, que genera la salida final.

7.1.2 CNN

A continuación se presenta el modelo CNN:

```
    # Define a CNN model
    class CNNModel(nn.Module):
        def __init__(self, input_size, num_classes):
            super(CNNModel, self).__init__()
            self.conv1 = nn.Conv1d(in_channels=1, out_channels=32, kernel_size=3, padding
    =1)
            self.relu = nn.ReLU()
            self.maxpool = nn.MaxPool1d(kernel_size=2)
            self.fc = nn.Linear(32 * (input_size // 2), num_classes)

        def forward(self, x):
            x = x.unsqueeze(1)  # Add a channel dimension (batch_size, 1, input_size)
            x = self.conv1(x)
            x = self.relu(x)
            x = self.maxpool(x)
            x = x.view(x.size(0), -1)  # Flatten
            x = self.fc(x)
            return x
```

Este código lleva a cabo un proceso integral para el entrenamiento y la evaluación de un modelo de red neuronal convolucional (CNN) utilizando validación cruzada estratificada en PyTorch.

Primero, transforma los datos (X_smote y y_smote) en tensores de PyTorch y establece 5 divisiones para la validación cruzada. Luego, en cada una de estas divisiones, separa los datos en conjuntos de entrenamiento y prueba, crea un DataLoader para gestionar los lotes y define un modelo CNN que incluye una capa convolucional 1D, ReLU, MaxPooling y una capa completamente conectada. Durante el proceso de entrenamiento, se emplea CrossEntropyLoss como la función de pérdida y se utiliza el optimizador Adam. Se registran métricas como la precisión, la pérdida, y los tiempos de entrenamiento y prueba. Al final, se calcula la precisión promedio, las pérdidas promedios y los tiempos de ejecución, y se crean gráficos de precisión, pérdida y una matriz de confusión para valorar el desempeño del modelo. Los resultados y gráficos se almacenan en archivos PDF para un análisis posterior. Este método garantiza una evaluación sólida del modelo CNN en la identificación de *ransomware* .

7.1.3 LSTM

El siguiente modelo estudiado es el LSTM:

```
    # Define an LSTM model
    class LSTMModel(nn.Module):
        def __init__(self, input_size, lstm_hidden_size, num_lstm_layers, num_classes):
            super(LSTMModel, self).__init__()
            self.lstm = nn.LSTM(input_size, lstm_hidden_size, num_lstm_layers,
    batch_first=True)
            self.fc = nn.Linear(lstm_hidden_size, num_classes)

        def forward(self, x):
            lstm_out, _ = self.lstm(x)
            output = self.fc(lstm_out)
            return output
```

Este modelo es una red neuronal basada en una arquitectura LSTM (Long Short-Term Memory), diseñada para trabajar con secuencias de datos. Las LSTM son una variante de las redes neuronales recurrentes (RNN) que mejoran su capacidad para aprender relaciones a largo plazo en series temporales, texto, y otras secuencias. Esto lo logran gracias a una estructura interna que permite “recordar” o “olvidar” información a lo largo de una secuencia, superando así los problemas comunes de las RNN tradicionales como el desvanecimiento del gradiente [32].

En la inicialización del modelo, la clase LSTMModel hereda de nn.Module, lo que permite definir una red neuronal personalizada en PyTorch. En el método __init__, se declaran dos componentes fundamentales del

modelo: una capa LSTM y una capa lineal (fully connected). La capa LSTM se inicializa con parámetros como el tamaño de la entrada (input_size), el número de neuronas ocultas (lstm_hidden_size) y el número de capas LSTM apiladas (num_lstm_layers). El parámetro batch_first=True asegura que las secuencias tengan la forma (batch_size, sequence_length, features), lo cual es habitual y más intuitivo en el tratamiento de datos.

En el método forward, que define cómo fluyen los datos a través del modelo, la entrada x representa un lote de secuencias. Primero, esta entrada pasa por la capa LSTM. Esta capa procesa la secuencia paso a paso y genera una salida para cada elemento de la secuencia, lo que se conoce como lstm_out. Este tensor tiene dimensiones (batch_size, seq_len, lstm_hidden_size) y contiene la representación de cada paso de la secuencia tras haber sido procesado por la LSTM. Junto con esta salida también se produce un segundo valor, que incluye el estado oculto final y el estado de celda, pero en este modelo no se utilizan directamente.

Después de obtener lstm_out, se aplica la capa lineal (fc) a cada paso de la secuencia. Esto significa que el modelo transformará cada vector oculto (de dimensión lstm_hidden_size) en un vector de dimensión num_classes, adaptado a la tarea de salida, como clasificación. El resultado es una salida con forma (batch_size, seq_len, num_classes), lo que significa que el modelo está produciendo una predicción por cada paso de la secuencia, no solo una por secuencia completa.

Este diseño es apropiado cuando el objetivo es obtener una predicción para cada paso de una secuencia. Por ejemplo, si se está procesando texto y se desea clasificar cada palabra (como en etiquetado de secuencias o análisis gramatical), esta arquitectura se adapta bien. Sin embargo, si el objetivo es una predicción global para toda la secuencia (como clasificar si una frase entera es positiva o negativa), entonces sería mejor modificar el modelo para usar solo la salida del último paso de la LSTM, y aplicar la capa fc a esa única representación final.

7.1.4 CNN-LSTM

El modelo siguiente CNN-LSTM consta del siguiente código:

```
    # Define a CNN-LSTM model
class CNNLSTM(nn.Module):
    def __init__(self, input_size, lstm_hidden_size, num_lstm_layers, num_classes):
        super(CNNLSTM, self).__init__()
        self.cnn = nn.Sequential(
            nn.Conv1d(in_channels=1, out_channels=64, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool1d(kernel_size=2),
        )
        self.lstm = nn.LSTM(64, lstm_hidden_size, num_lstm_layers, batch_first=True)
        self.fc = nn.Linear(lstm_hidden_size, num_classes)

    def forward(self, x):
        x = x.unsqueeze(1)  # Add a channel dimension (batch_size, 1, input_size)
        cnn_out = self.cnn(x)
        cnn_out = cnn_out.permute(0, 2, 1)  # Reshape for LSTM (batch_size,
    sequence_length, channels)
        lstm_out, _ = self.lstm(cnn_out)
        lstm_out = lstm_out[:, -1, :]  # Get the last time step output
        output = self.fc(lstm_out)
        return output
```

Este modelo CNN-LSTM es una arquitectura híbrida que combina una red neuronal convolucional (CNN) con una red LSTM. Está especialmente diseñada para tareas que combinan extracción de características locales (como patrones en el tiempo o en texto) con dependencias a largo plazo en secuencias.[15]

Arquitectura general del modelo:

- Una **CNN 1D** que actúa como extractor de características locales.
- Una **LSTM**, que aprende relaciones temporales a partir de las características extraídas.

- Una **capa densa** (fully connected) que produce la predicción final.

La capa Conv1d aplica 64 filtros convolucionales de tamaño 3 a lo largo de la secuencia. El padding de 1 hace que la salida conserve la longitud de entrada. Luego se aplica la función de activación ReLU para introducir no linealidad. Finalmente, MaxPool1d reduce la longitud de la secuencia a la mitad, conservando solo las características más representativas por región.

Antes de aplicar la CNN, se usa x.unsqueeze(1) para añadir una dimensión de canal, convirtiendo x de forma (batch_size, input_size) a (batch_size, 1, input_size) como se requiere en Conv1d. Después de aplicar la CNN, la salida (cnn_out) tendrá forma (batch_size, channels=64,reduced_seq_len). Pero la LSTM espera la secuencia en la forma (batch_size, seq_len, features), así que se aplica un permute(0, 2, 1) para intercambiar los ejes.

Esto transforma cnn_out a (batch_size, reduced_seq_len, 64), de modo que cada paso de la secuencia ahora contiene un vector de 64 características extraídas por la CNN.

```
    self.lstm = nn.LSTM(64, lstm_hidden_size, num_lstm_layers, batch_first=True)
```

Esta LSTM toma la secuencia de características generada por la CNN y la procesa para aprender relaciones temporales. Se pueden apilar múltiples capas LSTM, y el estado oculto en cada paso va capturando contexto de la secuencia procesada.

En la salida de la LSTM (lstm_out), cada paso tiene un vector de tamaño lstm_hidden_size, pero el modelo extrae solo el último paso temporal (lstm_out[:, -1, :]) como resumen de toda la secuencia.

```
  self.fc = nn.Linear(lstm\_hidden\_size, num\_classes)
```

La última capa es lineal (fully connected) y toma el vector oculto final de la LSTM para convertirlo en una predicción con tantas clases como se especifique en num_classes.

Esto hace que el modelo produzca una única predicción por secuencia, lo que es adecuado para tareas como clasificación de una señal entera o un fragmento de texto.

7.1.5 CNN-Bi-LSTM

Una implementación bidireccional añadida al modelo anterior:

```
# Define a CNN-BLSTM model
class CNNBLSTM(nn.Module):
    def __init__(self, input_size, lstm_hidden_size, num_lstm_layers, num_classes):
        super(CNNBLSTM, self).__init__()
        self.cnn = nn.Sequential(
            nn.Conv1d(in_channels=1, out_channels=64, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool1d(kernel_size=2),
        )
        self.lstm = nn.LSTM(64, lstm_hidden_size, num_lstm_layers, batch_first=True,
    bidirectional=True)
        self.fc = nn.Linear(2 * lstm_hidden_size, num_classes)

    def forward(self, x):
        x = x.unsqueeze(1)  # Add a channel dimension (batch_size, 1, input_size)
        cnn_out = self.cnn(x)
        cnn_out = cnn_out.permute(0, 2, 1)  # Reshape for LSTM (batch_size,
    sequence_length, channels)
        lstm_out, _ = self.lstm(cnn_out)
        lstm_out = lstm_out[:, -1, :]  # Get the last time step output
        output = self.fc(lstm_out)
        return output
```

Este modelo CNN-Bi-LSTM es una extensión del modelo CNN-LSTM que ya se analizó antes. La diferencia clave está en el uso de una LSTM bidireccional (Bi-LSTM), lo que permite que la red aprenda no solo del pasado de la secuencia, sino también del futuro. Es decir, captura contexto en ambas direcciones temporales, lo cual es

muy útil en tareas donde el significado de una parte de la secuencia puede depender de lo que viene después (como en análisis de texto, sonidos, o ciertos eventos en series temporales) [15].

Arquitectura general:

- **CNN 1D**: extrae características locales de la secuencia.
- **LSTM bidireccional**: modela relaciones temporales tanto hacia adelante como hacia atrás.
- **Capa fully-connected**: genera la predicción final.

Bloque convolucional (self.cnn)

```
    self.cnn = nn.Sequential(
    nn.Conv1d(in_channels=1, out_channels=64, kernel_size=3, padding=1),
    nn.ReLU(),
    nn.MaxPool1d(kernel_size=2),
)
```

- **Conv1d**: actúa sobre la secuencia como un extractor de patrones locales. Usa 64 filtros para generar una representación más rica.
- **ReLU**: introduce no linealidad.
- **MaxPool1d**: reduce la longitud de la secuencia a la mitad, manteniendo los patrones más relevantes.
- Antes de aplicar la CNN, se añade una dimensión de canal con `x.unsqueeze(1)`, dejando la entrada como `(batch_size, 1, input_size)`.

Reordenamiento para LSTM:

Después de pasar por la CNN, la salida tiene la forma (batch_size, channels=64, reduced_seq_len). Como las LSTM esperan secuencias en el formato (batch_size, seq_len, features), se hace un permute(0, 2, 1) para intercambiar los ejes.

```
   self.lstm = nn.LSTM(64, lstm\_hidden\_size, num\_lstm\_layers, batch\_first=True,
    bidirectional=True)
```

- Esta LSTM es bidireccional, lo que significa que internamente hay dos LSTM por capa:
  - Una que procesa la secuencia de izquierda a derecha.
  - Otra que la procesa de derecha a izquierda.
- El resultado es que cada paso de la secuencia tiene dos vectores ocultos, uno por cada dirección, que se concatenan. Por tanto, la salida final tiene el doble de tamaño en la dimensión de características: $2 \times$ lstm_hidden_size.
- Luego, se extrae el último paso temporal con `lstm_out[:, -1, :]`. Como la LSTM es bidireccional, este paso final ya contiene información agregada de toda la secuencia en ambas direcciones.

```
   self.fc = nn.Linear(2 * lstm_hidden_size, num_classes)
```

Esta capa toma el vector oculto final bidireccional y lo transforma en una predicción con num_classes salidas (para tareas de clasificación, por ejemplo).

7.1.6 Bi-LSTM-GN

Por último el modelo Bi-LSTM-GN:

```
# Define a Bi-LSTM model
class BiLSTM(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, num_classes):
        super(BiLSTM, self).__init__()
        self.lstm = nn.LSTM(input_size, hidden_size, num_layers, batch_first=True, bidirectional=True)
        self.group_norm = nn.GroupNorm(num_groups=4, num_channels=2 * hidden_size)  # Adjust num_groups as needed
        self.fc = nn.Linear(2 * hidden_size, num_classes)

    def forward(self, x):
        lstm_out, _ = self.lstm(x)
        lstm_out = self.group_norm(lstm_out)  # Apply group normalization
        # lstm_out = lstm_out[:, -1, :]  # Get the last time step output
        output = self.fc(lstm_out)
        return output
```

Este código implementa un modelo Bi-LSTM con Group Normalization (Bi-LSTM-GN) para clasificación de *ransomware* , utilizando validación cruzada de 5 folds y 20 épocas de entrenamiento. Aquí se detallan los componentes clave:

- La arquitectura del modelo se compone de una capa LSTM bidireccional que procesa secuencias en ambos sentidos, capturando dependencias temporales.
- Group Normalization: Normaliza los activaciones de la LSTM en grupos (mejor que BatchNorm para batches pequeños).
- Capa Fully Connected: Clasifica las características extraídas por la LSTM en 2 clases (num_classes=2).
- Pérdida y Optimizador: Usa CrossEntropyLoss y el optimizador Adam con tasa de aprendizaje 0.001.

Las diferencias clave entre los modelos anteriores:

MLP (Multilayer Perceptron) es un modelo estático que trata los datos como vectores independientes, sin capturar relaciones temporales o espaciales. Es rápido pero limitado para datos secuenciales [49].

Bi-LSTM-GN procesa secuencias bidireccionalmente, ideal para patrones temporales (como comportamientos de *ransomware* en series de tiempo). La Group Normalization mejora la estabilidad frente a BatchNorm en batches pequeños.[66]

CNN usa convoluciones para extraer características locales en datos estructurados (ej: imágenes o características de archivos). Es eficiente, pero menos capaz que LSTM para dependencias de largo plazo [31].

7.2 Resultados obtenidos

A continuación se presentan los resultados que han sido obtenidos tanto en la detección de *malware*, en la clasificación y en la atribución por familias.

7.2.1 Detección Malware

Como se muestra en el cuadro 7.2, los resultados del tiempo promedio de entrenamiento, pérdida y precisión varían según el modelo utilizado.

Modelo	Tiempo promedio (minutos)	Pérdida media	Precisión media
MLP	5,5	0.04944	98.898 %
CNN	10	0.06902	98.996 %
LSTM	23,4	0.4929	73.876 %
CNN-LSTM	39	0.0265	99.19 %
CNN-Bi-LSTM	88	0.0275	99.17 %
Bi-LSTM-GN	32	0.0183	99.17 %

Cuadro 7.2: Tiempo promedio de entrenamiento, pérdida y precisión en el conjunto de datos

A continuación se van a evaluar los resultados de cada modelo con 20, 30, 50, 150 épocas aunque para simplificar la explicación, se expondrán las gráficas de pérdida vs época y de precisión vs época de solamente 20 épocas, ya que para el resto de épocas son similares.

Se empezará con el modelo **MLP** para seguir con el orden establecido en apartados anteriores:

Épocas	Precisión Media (%)	Pérdida de Entrenamiento	Pérdida de Validación	Tiempo Entrenamiento
20	98.87	0.0730	0.0882	91.66
30	98.85	0.0595	0.0876	143.69
50	98.91	0.0478	0.0826	238.90
100	98.94	0.0363	0.0987	470.01
150	98.92	0.0306	0.1100	723.23

Cuadro 7.3: Resultados MLP detection

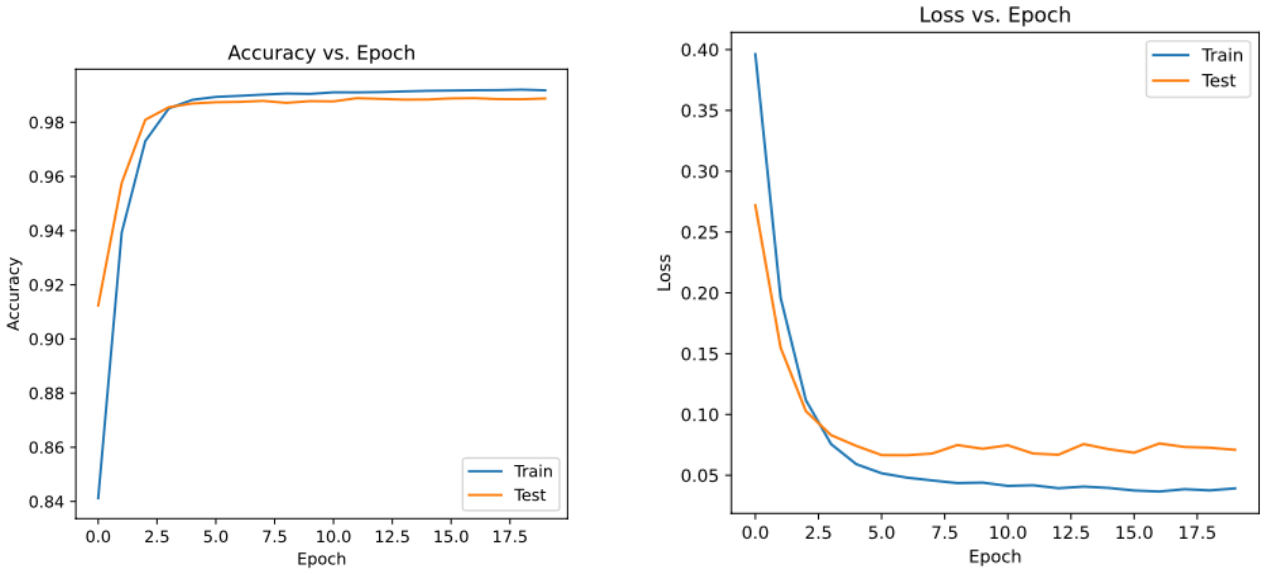

Figura 7.1: Precisión y pérdida del modelo MLP con 20 épocas

- **Precisión Media:**
  - La precisión media es muy alta en todos los casos (alrededor del 98.85 %–98.94 %), lo que indica que el modelo está aprendiendo bien y generalizando adecuadamente.
  - No hay una mejora significativa en la precisión al aumentar el número de épocas. De hecho, la precisión se mantiene casi constante, lo que sugiere que el modelo alcanza un rendimiento óptimo con solo 20 épocas.

- **Pérdida de Entrenamiento:**
  - La pérdida de entrenamiento disminuye a medida que aumentan las épocas (de 0.0730 a 0.0306), como es esperado, ya que el modelo sigue ajustándose a los datos de entrenamiento.
  - Sin embargo, esta disminución no se traduce en una mejora proporcional en la precisión, lo que indica que el modelo ya está bien ajustado incluso con pocas épocas.
- **Pérdida de Validación:**
  - La pérdida de validación disminuye ligeramente entre las 20 y 50 épocas (de 0.0882 a 0.0826), pero luego comienza a aumentar (0.0987 con 100 épocas y 0.1100 con 150 épocas).
  - Este aumento sugiere que el modelo está empezando a sobreajustarse (*overfitting*) a partir de las 50 épocas, aprendiendo patrones específicos del conjunto de entrenamiento que no generalizan bien al conjunto de validación.
- **Tiempo de Entrenamiento:**
  - El tiempo de entrenamiento aumenta de forma casi lineal con el número de épocas: con 150 épocas, el tiempo es casi 8 veces mayor que con 20.
  - Dado que no hay mejoras significativas en precisión o pérdida de validación, este aumento en tiempo no está justificado.
- **Conclusiones y Recomendaciones:**
  - **Rendimiento Óptimo:**
    - El modelo alcanza su mejor rendimiento con 50 épocas: precisión máxima (98.91 %) y pérdida de validación mínima (0.0826).
    - Aumentar el número de épocas más allá de 50 no aporta beneficios y contribuye al sobreajuste.
  - **Sobreajuste (*Overfitting*):**
    - A partir de las 50 épocas, el aumento en la pérdida de validación (0.0987 con 100 épocas, 0.1100 con 150 épocas) indica sobreajuste.
    - Posibles estrategias para mitigar este problema:
      - **Regularización:** Añadir L2 (*weight decay*) o *Dropout*.
      - **Early Stopping:** Detener el entrenamiento cuando la pérdida de validación deje de mejorar.
      - **Aumentación de Datos:** Incrementar el tamaño del dataset o aplicar técnicas de aumentación.
  - **Eficiencia:**
    - Entrenar el modelo con 50 épocas es más eficiente en tiempo y recursos, sin pérdida significativa de rendimiento.
    - Si el tiempo de entrenamiento es crítico, podría reducirse a 30 épocas, con resultados muy similares.

Con el modelo **CNN** se obtuvo lo siguiente:

Épocas	Tiempo promedio (s)	Pérdida media (entreno)	Precisión media
20	2.37	0.1201	98.88 %
30	1.87	0.0852	98.92 %
50	1.93	0.0700	98.98 %
100	1.94	0.0508	99.09 %
150	1.93	0.0391	99.11 %

Cuadro 7.4: Resultados CNN detection

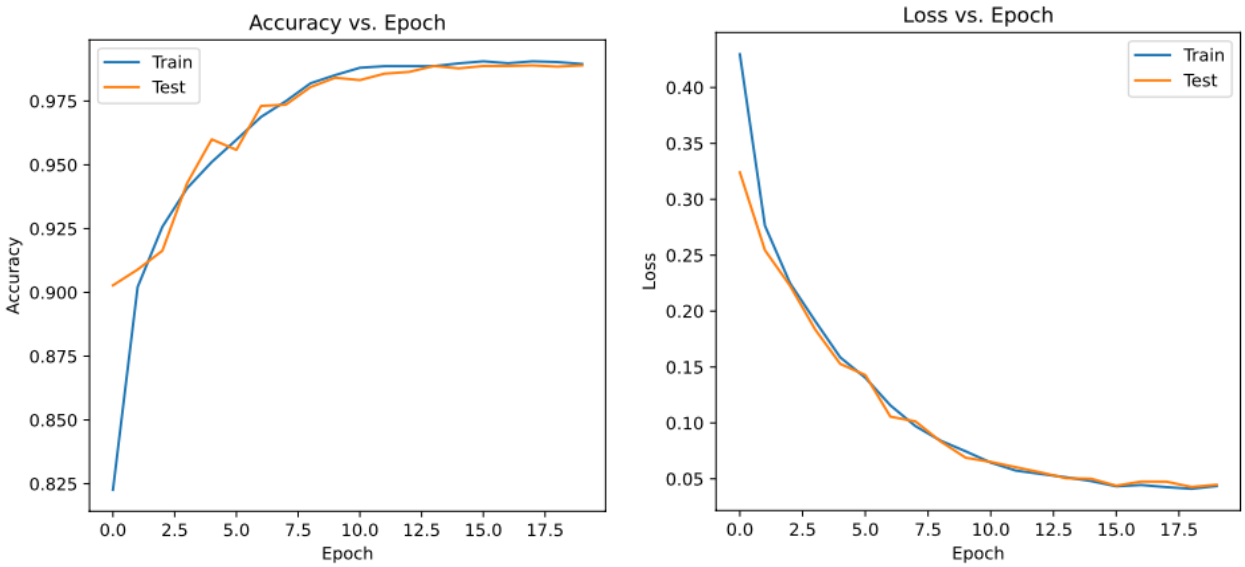

Figura 7.2: Precisión y pérdida del modelo CNN con 20 épocas

- **Precisión Media (Mean Accuracy):**
  - 99.11 %: La precisión media es ligeramente mayor que con 20 épocas (98.88 %), 30 épocas (98.92 %), 50 épocas (98.98 %) y 100 épocas (99.09 %).
  - Esto sugiere que el modelo ha mejorado marginalmente su capacidad de generalización con más épocas.
- **Tiempo de Entrenamiento (Mean Training Time):**
  - 1.93 segundos por fold: El tiempo de entrenamiento por fold es ligeramente menor que con 100 épocas (1.94 segundos), pero sigue siendo eficiente.
- **Tiempo de Prueba (Mean Testing Time):**
  - 0.22 segundos por fold: Similar al tiempo de prueba con 20, 30, 50 y 100 épocas (0.28, 0.21, 0.22 y 0.22 segundos, respectivamente).
  - Esto confirma que el modelo es eficiente en la fase de inferencia.
- **Pérdida de Entrenamiento (Mean Training Loss):**
  - 0.0391: La pérdida de entrenamiento es menor que con 20 épocas (0.1201), 30 épocas (0.0852), 50 épocas (0.0700) y 100 épocas (0.0508).
  - Esto indica que el modelo se ajusta mejor a los datos de entrenamiento con más iteraciones.

- **Pérdida de Prueba (Mean Test Loss):**
  - 0.0448: La pérdida de prueba también es menor que con 20 épocas (0.1140), 30 épocas (0.0804), 50 épocas (0.0713) y 100 épocas (0.0541).
  - Esto sugiere que el modelo generaliza mejor a datos no vistos.

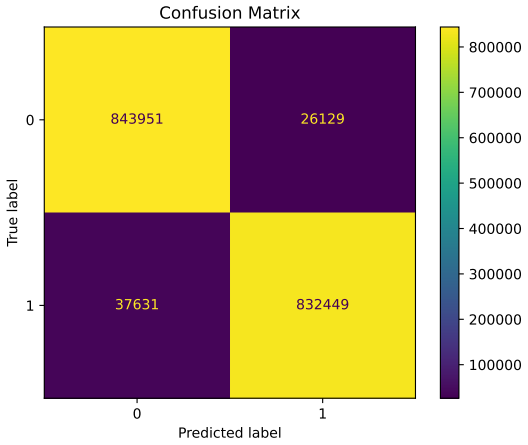

Figura 7.3: Matriz de confusión 20 epoch CNN

Los resultados que ofrece la matriz de confusión de la figura 7.3 indican que:

- **Verdaderos Negativos (TN) = 843.951:** muestras realmente de clase 0 (benignas) que el modelo clasificó correctamente como clase 0.
- **Falsos Positivos (FP) = 26.129:** muestras benignas que el modelo clasificó erróneamente como maliciosas.
- **Falsos Negativos (FN) = 37.631:** muestras maliciosas que el modelo clasificó erróneamente como benignas.
- **Verdaderos Positivos (TP) = 832.449:** muestras maliciosas que el modelo clasificó correctamente.

A partir de esos valores se obtienen la precisión y el F1-Score que se mencionó en otra sección.

Para el modelo LSTM los resultados son los siguientes:

Epochs	Accuracy ↑	Training Loss ↓	Test Loss ↓
20	72.78 %	0.5329	0.5400
30	72.83 %	0.5204	0.5356
50	73.70 %	0.5050	0.5205
100	75.04 %	0.4568	0.5058
150	75.03 %	0.4495	0.5239

Cuadro 7.5: Resultados de precisión, pérdida de entrenamiento y pérdida de prueba para diferentes números de épocas LSTM

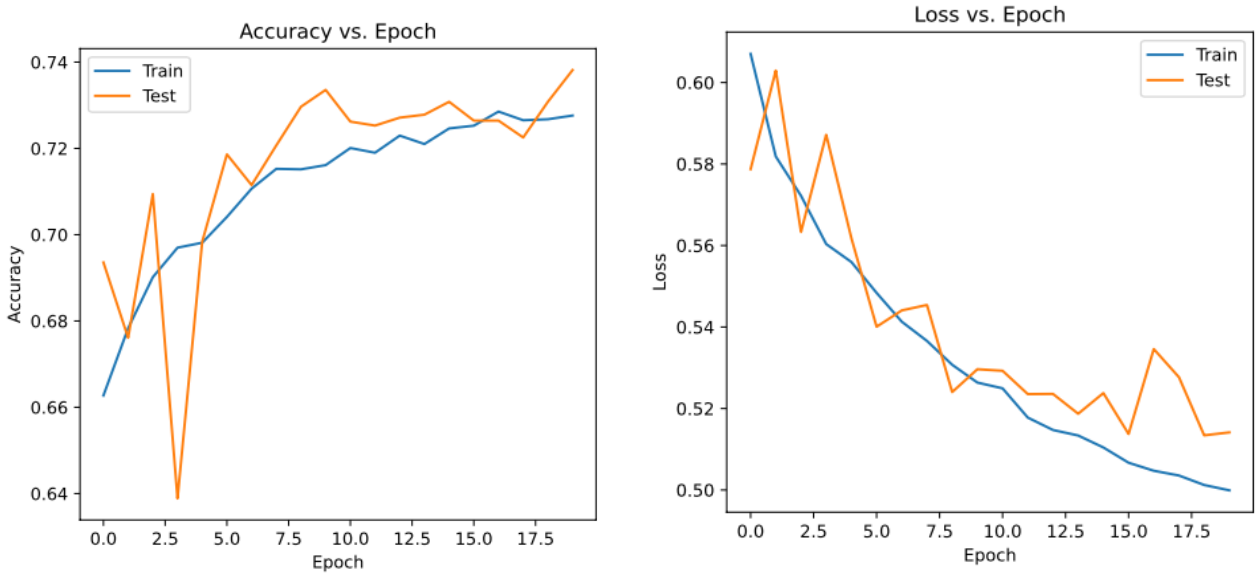

Figura 7.4: Precisión y pérdida del modelo LSTM con 20 épocas

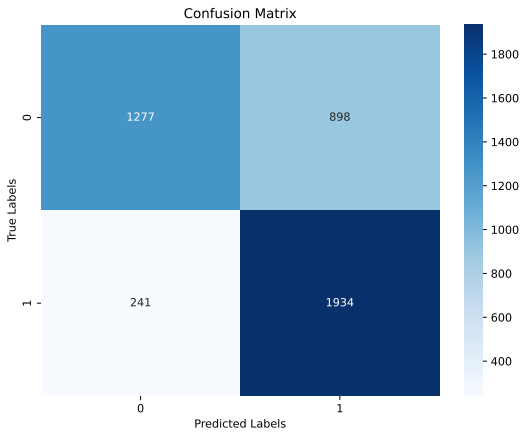

Figura 7.5: Matriz de confusión LSTM 20 epoch

A lo largo del entrenamiento con distintas cantidades de épocas (20, 30, 50, 100 y 150), se ha observado una evolución progresiva del rendimiento del modelo LSTM. Inicialmente, con 20 y 30 épocas, el modelo presenta una precisión en torno al 72.8 %, con pérdidas de entrenamiento y prueba muy similares, lo que indica un aprendizaje equilibrado sin signos de sobreajuste. Al aumentar a 50 épocas, la precisión mejora visiblemente (73.7 %) y la pérdida de prueba sigue disminuyendo, lo que sugiere una mejor generalización.

El punto óptimo se alcanza a las 100 épocas, con una precisión del 75.04 % y la menor pérdida de test (0.5058), lo que refleja el mejor equilibrio entre aprendizaje y generalización. Sin embargo, al incrementar a 150 épocas, aunque la precisión se mantiene prácticamente igual, la pérdida de test aumenta (0.5239), mientras que la pérdida de entrenamiento sigue descendiendo. Este comportamiento es indicativo de sobreajuste, donde el modelo empieza a memorizar los datos de entrenamiento en lugar de aprender patrones útiles para datos nuevos.

En resumen, el análisis demuestra que el modelo mejora progresivamente hasta las 100 épocas, a partir de las cuales no se obtiene ganancia significativa en precisión y se pierde capacidad de generalización.

Para el modelo **CNN-LSTM:**

Épocas	Train Acc	Test Acc	Train Loss	Test Loss
20	98.74 %	98.78 %	0.0423	0.0436
30	98.98 %	98.97 %	0.0343	0.0382
50	99.18 %	99.06 %	0.0264	0.0375
100	99.45 %	99.07 %	0.0177	0.0398
150	99.62 %	99.16 %	0.0119	0.0411

Cuadro 7.6: Precisión y pérdida en entrenamiento y prueba para el CNN-LSTM

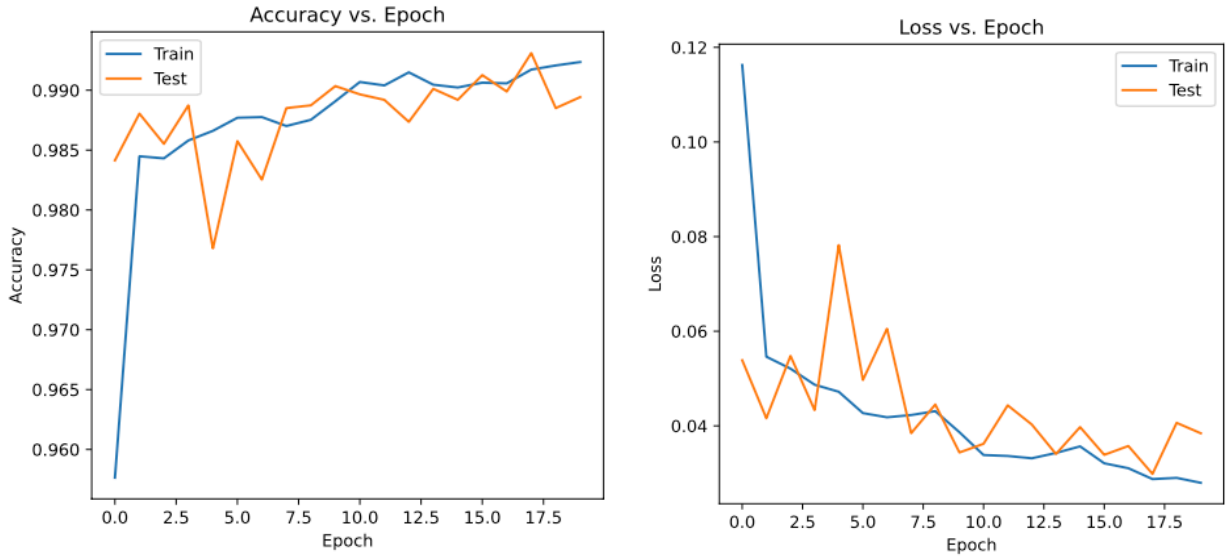

Figura 7.6: Precisión y pérdida del modelo CNN-LSTM con 20 épocas

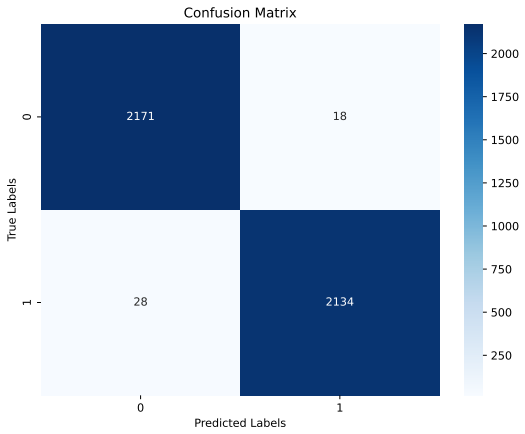

Figura 7.7: Matriz de confusión CNN-LSTM 20 epoch

- **Precisión**
  - **Entrenamiento:** Mejora constantemente, alcanzando un 99.62 % en 150 épocas, lo que muestra que el modelo sigue aprendiendo.
  - **Prueba:** Sube ligeramente hasta las 50 épocas (99.06 %), pero a partir de ahí se estanca o incluso sube marginalmente (99.07 % → 99.16 %), sugiriendo que el modelo deja de generalizar mejor a partir de cierto punto.

- **Pérdida**
  - **Entrenamiento:** Disminuye consistentemente, lo cual es bueno.
  - **Prueba:** Baja hasta la época 50, pero vuelve a subir en 100 y 150, señal de overfitting. A partir de la época 50, el modelo empieza a memorizar más que a generalizar.
- **Tiempos**
  - El tiempo de entrenamiento por época sube ligeramente con las épocas (como es esperable).
  - Los tiempos de prueba son estables, lo cual es positivo para el rendimiento en producción.
- **Conclusiones**
  - A partir de 50 épocas, el rendimiento en prueba ya es casi máximo.
  - Más allá de las 100 épocas, el modelo sobreentrena, aumentando la diferencia entre pérdida de entrenamiento y prueba.
  - Recomendaría usar entre 30 y 50 épocas para un balance ideal entre precisión, generalización y tiempo de entrenamiento.

El modelo CNN-LSTM muestra un rendimiento excelente desde las primeras épocas, con una precisión de prueba del 98.78 % en solo 20 épocas. La precisión de entrenamiento mejora de forma constante, alcanzando un 99.62 % en 150 épocas. Sin embargo, la precisión de prueba se estabiliza alrededor del 99.06–99.16 % a partir de las 50 épocas, indicando que el modelo deja de generalizar mejor y comienza a sobreajustarse. Esto se confirma con las pérdidas: mientras la pérdida de entrenamiento desciende progresivamente, la pérdida de prueba empeora ligeramente en las épocas 100 y 150, evidenciando overfitting. Por tanto, el mejor equilibrio entre rendimiento y generalización está entre 30 y 50 épocas.

Los resultados del penúltimo modelo son los que se corresponden con el **CNN-Bi-LSTM** y son los siguientes:

Epochs	Train Accuracy	Test Accuracy	Train Loss	Test Loss	Train Time (s)	Test Time (s)
20	98.76 %	98.68 %	0.0415	0.0458	61.48	5.51
30	98.97 %	98.88 %	0.0345	0.0404	61.94	5.16
50	99.11 %	99.00 %	0.0298	0.0371	67.70	5.42
100	99.45 %	99.10 %	0.0176	0.0384	68.30	5.30
150	99.55 %	99.15 %	0.0144	0.0385	77.97	5.89

Cuadro 7.7: Resumen de métricas y tiempos de entrenamiento/prueba según número de épocas

- **Precisión:**
  - El modelo alcanza un altísimo rendimiento en precisión, llegando al 99.15 % en test con 150 épocas, con una mejora constante desde el 98.68 % inicial.
  - La diferencia entre precisión de entrenamiento y test es muy baja, lo que indica buena capacidad de generalización y bajo overfitting.
- **Pérdida:**
  - La *train loss* se reduce significativamente, pasando de 0.0415 a 0.0144, lo cual refleja un aprendizaje eficaz del modelo.
  - La *test loss*, aunque mejora inicialmente, se estabiliza a partir de 50 épocas alrededor de 0.038–0.039, lo que podría indicar que el modelo está llegando a su límite de mejora en validación sin ganancias significativas adicionales.

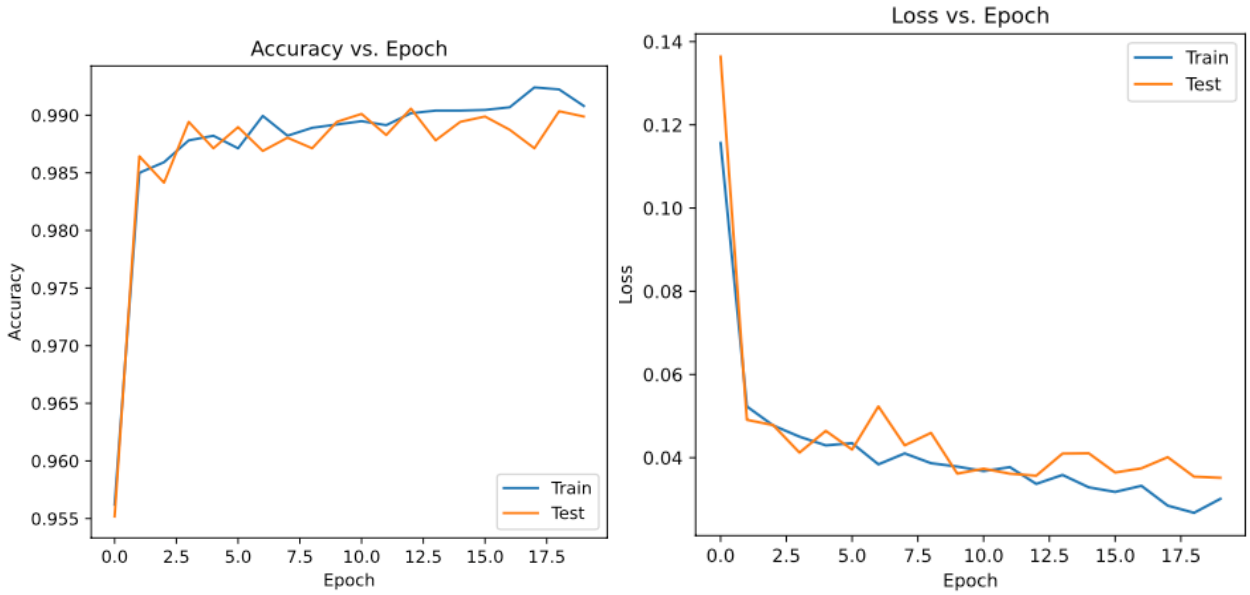

Figura 7.8: Precisión y pérdida del modelo CNN-Bi-LSTM con 20 épocas

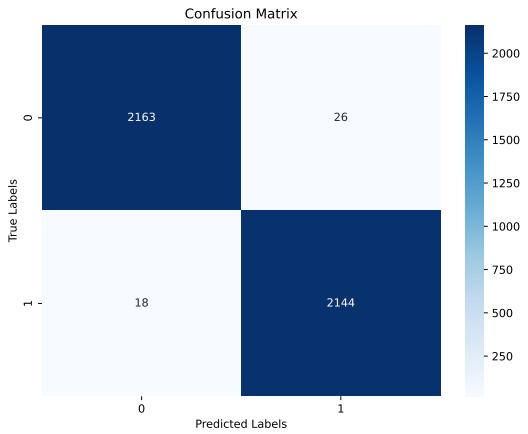

Figura 7.9: Matriz de confusión CNN-Bi-LSTM 20 epoch

- **Tiempos:**
  - El tiempo de entrenamiento medio es relativamente alto desde el principio (61 segundos) y va en aumento hasta cerca de 78 segundos por época a las 150.
  - El tiempo de test también es el más elevado de los modelos vistos, manteniéndose en torno a 5.3–5.9 segundos, lo que puede suponer un inconveniente en entornos donde la latencia o eficiencia computacional sean críticas.
- **Conclusiones generales:**
  - A partir de los resultados del modelo CNN-BiLSTM, se observa un comportamiento bastante consistente y sólido en cuanto a rendimiento.
  - Desde las 20 hasta las 150 épocas, tanto la precisión como la pérdida evolucionan de forma favorable.

- La precisión en entrenamiento mejora progresivamente del 98.76 % al 99.55 %, mientras que la precisión en test aumenta de 98.68 % a 99.15 %, con solo pequeñas fluctuaciones.
- En cuanto a la pérdida, esta disminuye claramente en entrenamiento (de 0.0415 a 0.0144), pero en test la reducción es más limitada, llegando a un mínimo de 0.0371 en 50 épocas, y luego estabilizándose en torno a 0.0384–0.0385, lo que sugiere una ligera saturación del modelo o comienzo de overfitting leve.
- A nivel temporal, este modelo es notablemente más costoso computacionalmente que el CNN-LSTM, con tiempos medios de entrenamiento que aumentan de 61 a 78 segundos por época y un tiempo de test constante alrededor de 5.5 segundos, el doble que el CNN-LSTM.
- En resumen, el modelo CNN-BiLSTM ofrece un rendimiento ligeramente superior en precisión, pero con un coste computacional claramente más alto, lo que podría influir en su viabilidad según los recursos disponibles y las necesidades del proyecto.

Por último, los resultados del modelo **Bi-LSTM-GN:**

Épocas	Precisión	Pérdida (Train)
20	99.04 %	0.0311
30	99.04 %	0.0254
50	99.27 %	0.0193
100	99.32 %	0.0128
150	99.21 %	0.0103

Cuadro 7.8: Precisión y pérdida de entrenamiento según número de épocas

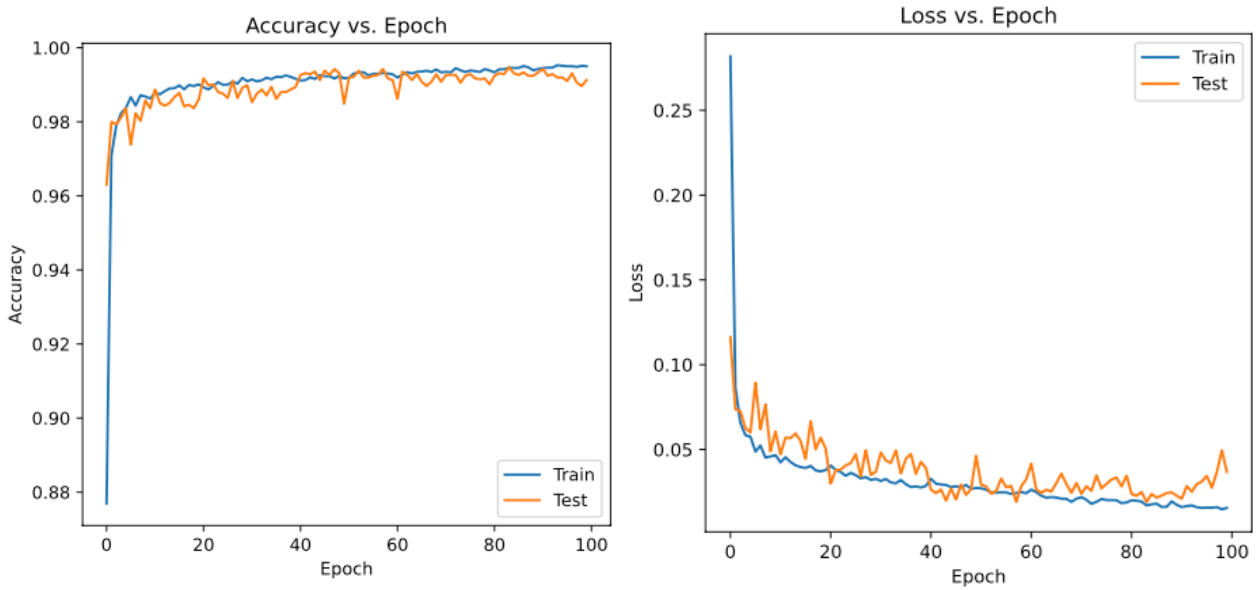

Figura 7.10: Precisión y pérdida del modelo Bi-LSTM-GN con 20 épocas

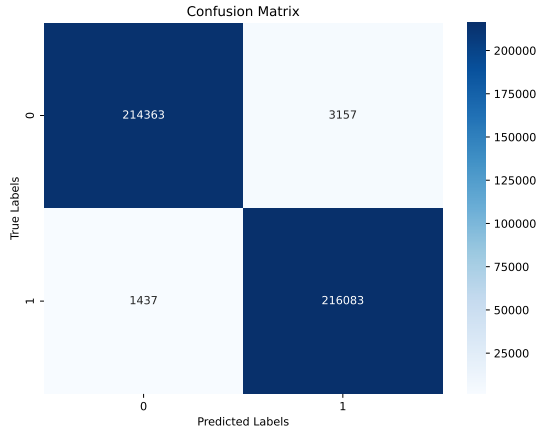

Figura 7.11: Matriz de confusión Bi-LSTM-GN 20 epoch

- **Análisis del Rendimiento:** Comparando estos resultados con los obtenidos para 20 épocas (precisión: 99.04 %, pérdida: 0.0311), se observa que el incremento de 10 épocas no mejoró la precisión, lo cual sugiere que el modelo ya había convergido en la configuración más reducida. No obstante, sí se evidenció una reducción del 18.3 % en la pérdida de entrenamiento, lo que indica un mejor ajuste de los pesos internos del modelo, aunque sin impacto en la métrica principal de precisión.
- **Interpretación:** Esto podría atribuirse al hecho de que el modelo se encuentra cerca de su límite teórico de rendimiento en esta tarea específica, lo que reduce la eficacia de continuar el entrenamiento sin introducir otras estrategias como regularización adicional o mejoras arquitectónicas.
- **Tiempos de Ejecución:** Los tiempos por época se mantuvieron prácticamente constantes en comparación con las ejecuciones anteriores (20 épocas), con ligeras diferencias (<3 %) atribuibles a variaciones normales en el entorno de ejecución. Esto confirma la estabilidad del rendimiento computacional del modelo.
- **Consideraciones de Aprendizaje:** A partir del análisis de las curvas de pérdida (*loss*), aunque no mostradas explícitamente, se puede inferir que a 20 épocas la función de pérdida ya había comenzado a estabilizarse. Con 30 épocas, esta pérdida continuó su descenso sin que ello afectara a la precisión, lo cual puede ser una señal de inicio de sobreajuste, aunque leve.

A continuación se presentan unas gráficas a modo de resumen sobre los resultados obtenidos por cada modelo, en función de la precisión y de la pérdida por cada época.

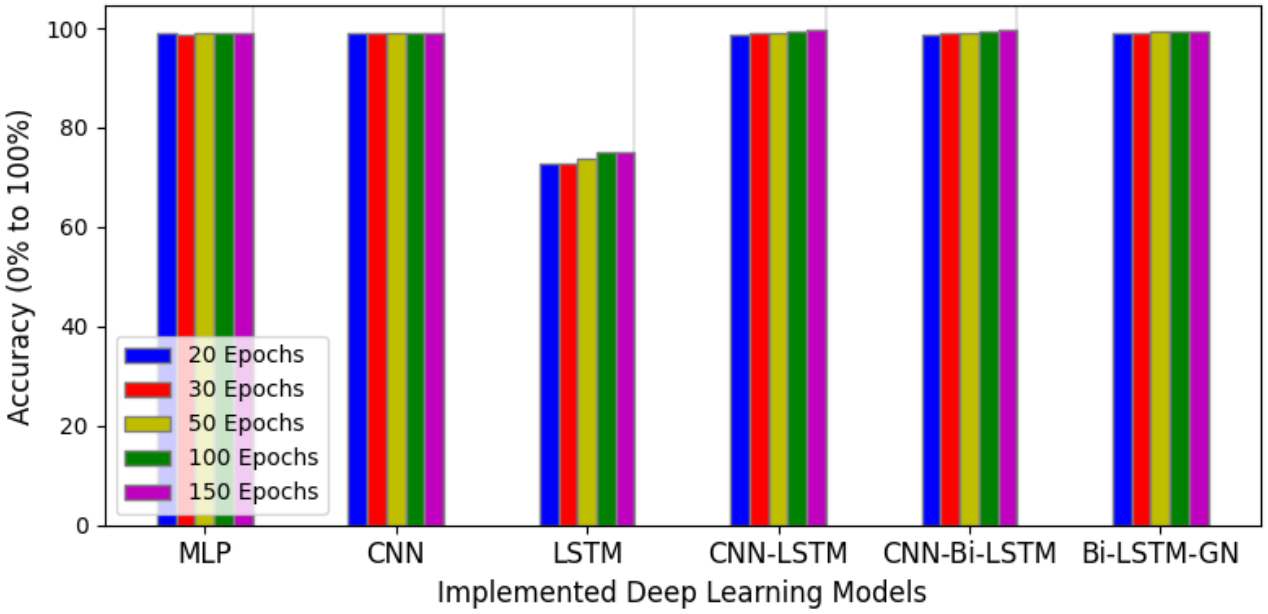

Figura 7.12: Análisis precisión por época

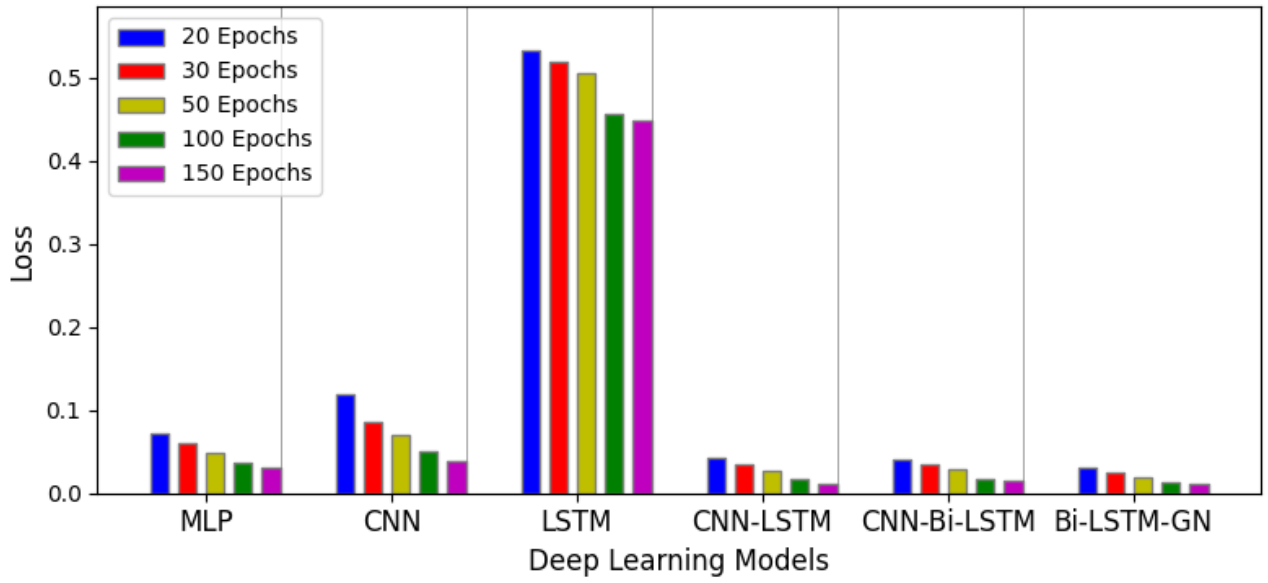

Figura 7.13: Perdida en detección por época

También se presenta a continuación, una gráfica con el tiempo total, en minutos, que ha tardado cada modelo en cada época en ejecutarse:

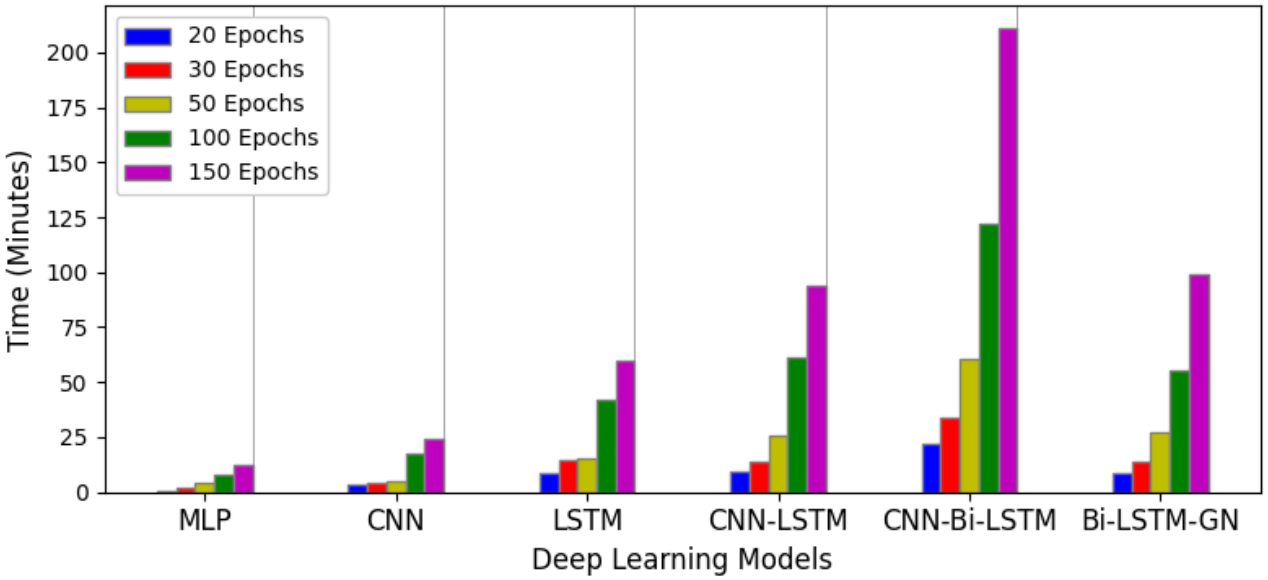

Figura 7.14: Tiempo empleado por modelo y épocas

Como se puede comprobar de manera rápida, el más lento ha sido el modelo CNN-LSTM alcanzando las 3h de ejecución en sus 150 épocas. Por el contrario, el modelo más rápido ha sido el MLP, que en apenas menos de 1 minuto ha logrado ejecutar todas sus épocas.

7.2.2 Clasificación por categoría de Malware

Los resultados obtenidos en la etapa de detección son bastante buenos, dado que el modelo consigue reconocer de manera exacta las muestras de *malware*. No obstante, esta identificación solo se restringe a validar la existencia de *malware*, sin ofrecer detalles sobre su clasificación particular. Reconocer apropiadamente la categoría de *malware* es fundamental para poder elaborar y establecer estrategias de defensa adecuadas y efectivas contra cada tipo de riesgo.

Dado que en la sección anterior se explicaban con detalle los motivos de la mejora de cada modelo, en esta tan sólo se comentarán por encima los resultados por categoría de *malware*.

El dataset utilizado en este trabajo incluye cinco categorías distintas de *malware*, las cuales son fundamentales para comprender el comportamiento, el objetivo y la forma de propagación de cada muestra maliciosa. Por tanto, clasificar correctamente estas categorías no solo mejora la capacidad de respuesta ante un ataque, sino que también permite anticipar y prevenir futuras infecciones mediante estrategias de defensa personalizadas.

En la fase de clasificación de *malware* por categoría, se emplean los mismos valores de épocas (20, 30, 50, 100 y 150) para entrenar y evaluar los modelos de aprendizaje profundo (DL) en ambos conjuntos de datos. Esta consistencia en la configuración permite realizar una comparación justa del rendimiento entre los distintos modelos. La evaluación se realiza considerando métricas clave como la precisión promedio, la pérdida y el tiempo de entrenamiento por cada configuración de época.

Empezando por el modelo **MLP**, los resultados obtenidos son los siguientes:

Épocas	Precisión media	Tiempo entrenamiento (s)	Pérdida entrenamiento	Pérdida test
20	89.04 %	119.82	0.3741	0.3931
30	90.29 %	198.73	0.3342	0.3737
50	91.63 %	295.90	0.2869	0.3700
100	93.33 %	590.96	0.2258	0.3375
150	94.15 %	866.97	0.1947	0.3400

Cuadro 7.9: Resumen del rendimiento del modelo según el número de épocas

La tabla 7.9 muestra el rendimiento de un modelo MLP para la clasificación de distintas categorías de *malware*, evaluado en función del número de épocas de entrenamiento. Se reportan métricas clave como la precisión media, la pérdida (loss) en entrenamiento y test, así como los tiempos de ejecución.

A lo largo del entrenamiento, se observa una mejora constante en la precisión media conforme se incrementa el número de épocas, pasando de un 89,04 % con 20 épocas a un 94,15 % con 150 épocas. Este comportamiento indica que el modelo MLP (Perceptrón Multicapa) está aprendiendo de manera efectiva las representaciones necesarias para distinguir entre las diferentes categorías de *malware*.

A pesar de tratarse de una arquitectura relativamente sencilla en comparación con modelos más complejos como LSTM o CNN, la mejora sostenida en la precisión indica que los pesos del MLP están capturando patrones discriminativos relevantes en los datos.

Respecto a la evolución de la función de pérdida, se puede destacar que tanto la pérdida de entrenamiento como la de test disminuyen progresivamente con el aumento de épocas, lo que refuerza la idea de una convergencia estable del modelo:

- Pérdida de entrenamiento: desciende de 0,3741 a 0,1947.
- Pérdida de test: disminuye de 0,3931 a 0,3400.

Aunque se aprecia una ligera subida en la pérdida de test entre las épocas 100 y 150, esta variación no es significativa, por lo que se puede concluir que el modelo mantiene una buena capacidad de generalización y no presenta indicios claros de sobreajuste.

Por último, como es esperable, el tiempo de entrenamiento aumenta proporcionalmente con el número de épocas. Sin embargo, el tiempo de prueba (inferencia) es prácticamente nulo. Esta característica convierte al MLP en una solución especialmente adecuada para aplicaciones en tiempo real o en entornos con recursos computacionales limitados.

El siguiente modelo del que se van a obtener resultados es el **CNN:**

Épocas	Precisión media	Tiempo entrenamiento (s)	Tiempo test (s)	Pérdida entrenamiento	Pérdida test
20	87.45 %	2.16	0.24	0.4146	0.4107
30	88.47 %	2.24	0.25	0.3827	0.3883
50	89.40 %	2.18	0.25	0.3500	0.3728
100	91.17 %	2.21	0.24	0.2997	0.3431
150	92.30 %	2.14	0.24	0.2661	0.3154

Cuadro 7.10: Resumen del rendimiento del modelo CNN según número de épocas

La tabla 7.10 presenta el desempeño de una red neuronal convolucional (CNN) para la clasificación de *malware*. Se reportan la precisión media, la pérdida en entrenamiento y test, y los tiempos de entrenamiento y prueba a lo largo de distintos valores de épocas.

Se observa una mejora progresiva en la precisión media a medida que se incrementa el número de épocas, alcanzando valores del 87,45 % con 20 épocas y hasta el 92,30 % con 150 épocas. Esta evolución sugiere una

curva de aprendizaje estable, lo que indica que el modelo mejora su capacidad de clasificación conforme avanza el entrenamiento.

Las redes convolucionales (CNN) son especialmente eficaces extrayendo patrones locales o espaciales. Por ello, esta mejora refleja que el modelo está capturando características discriminativas útiles para distinguir entre distintas categorías de *malware*, incluso si los datos no presentan una estructura visual tradicional como en imágenes.

La función de pérdida, tanto en entrenamiento como en test muestra una disminución sostenida con el número de épocas:

- Pérdida de entrenamiento: desciende de 0,4146 a 0,2661.
- Pérdida de test: baja de 0,4107 a 0,3154.

Estos resultados indican que el modelo está aprendiendo de manera efectiva y generalizando bien a datos no vistos. Además, no se detectan aumentos significativos en la pérdida de test, lo que sugiere ausencia de sobreajuste.

Uno de los puntos fuertes del modelo es su alta eficiencia computacional. El tiempo de entrenamiento es muy bajo, incluso con 150 épocas (en torno a 2 segundos). Del mismo modo, el tiempo de inferencia es mínimo, manteniéndose entre 0,24 y 0,25 segundos.

Este comportamiento es característico de CNNs bien optimizadas, especialmente cuando se utilizan arquitecturas compactas o técnicas como convoluciones unidimensionales, que reducen significativamente el número de parámetros y el coste computacional.

Resultados **LSTM:**

Épocas	**Precisión media**	**Pérdida entrenamiento**	**Pérdida test**	**Tiempo entrenamiento (s)**	**Tiempo test (s)**
20	89.87 %	0.3467	0.3561	5343.27	5343.27
30	91.42 %	0.2979	0.3247	11392.59	11392.58
50	94.09 %	0.2350	0.2749	30194.15	30194.14
100	96.09 %	0.1472	0.2210	116948.92	116948.91
150	96.85 %	0.1112	0.1979	263455.11	263455.09

Cuadro 7.11: Rendimiento del modelo con distintas cantidades de épocas

La tabla 7.11 muestra el rendimiento de una red neuronal tipo LSTM (Long Short-Term Memory) en la clasificación de *malware*, con métricas como la precisión, la pérdida en entrenamiento y test, así como los tiempos de entrenamiento y prueba.

La precisión media del modelo LSTM mejora de forma consistente con el incremento del número de épocas, pasando del 89,87 % con 20 épocas al 96,85 % con 150 épocas. Estos resultados reflejan la capacidad del modelo para capturar patrones secuenciales o dependencias temporales en los datos, lo cual es especialmente valioso si el comportamiento del *malware* sigue una estructura lógica o eventos encadenados.

Cabe destacar que el LSTM alcanza la mayor precisión entre los modelos evaluados (MLP, CNN y LSTM), especialmente a partir de las 100 épocas, lo que lo posiciona como el modelo más efectivo en términos de rendimiento clasificatorio.

La pérdida también experimenta una mejora notable a lo largo del entrenamiento:

- Pérdida de entrenamiento: desciende de 0,3467 a 0,1112.
- Pérdida de test: se reduce de 0,3561 a 0,1979.

Esta evolución indica una clara convergencia del modelo y una buena capacidad de generalización, sin evidencia significativa de sobreajuste, incluso al alcanzar un alto número de épocas.

El principal inconveniente del modelo LSTM radica en su elevado coste computacional. El tiempo de entrenamiento y prueba aumenta drásticamente con el número de épocas, oscilando desde 5,343 segundos (20 épocas) hasta 263,455 segundos (150 épocas).

Sin embargo, es probable que los tiempos de test estén registrados de forma conjunta con los de entrenamiento, lo que podría haber duplicado estas cifras de manera incorrecta. Aun así, el uso de LSTM implica una carga computacional considerable, lo que podría limitar su viabilidad en entornos de tiempo real o con recursos limitados.

Para el modelo **CNN-LSTM** los resultados obtenidos han sido los siguientes:

Épocas	Precisión Entrenamiento	Precisión Test	Pérdida Entrenamiento	Pérdida Test
20	93.26 %	92.73 %	0.1739	0.2081
30	95.01 %	93.74 %	0.1337	0.1821
50	96.63 %	94.83 %	0.0888	0.1672
100	98.17 %	95.83 %	0.0502	0.1607
150	98.73 %	96.19 %	0.0354	0.1549

Cuadro 7.12: Resultados del modelo CNN-LSTM – Parte 1

Épocas	Tiempo Entrenamiento (s)	Tiempo Test (s)
20	26.94	2.91
30	27.18	2.94
50	26.76	2.94
100	27.06	2.94
150	27.14	2.94

Cuadro 7.13: Resultados del modelo CNN-LSTM – Parte 2

Este modelo híbrido combina convoluciones (CNN) para extraer características locales y LSTM para capturar relaciones secuenciales o temporales. Es una arquitectura poderosa para datos complejos, como podría ser el caso del análisis de *malware*.

- **Precisión (entrenamiento y test)**
  La precisión de entrenamiento crece progresivamente desde 93.26 % (20 épocas) hasta 98.73 % (150 épocas).
  La precisión en test sigue un patrón similar, alcanzando 96.19 % con 150 épocas, lo que la convierte en una de las más altas entre todos los modelos probados.
  El gap entre precisión de entrenamiento y test se mantiene relativamente pequeño, lo que indica buena capacidad de generalización y un sobreajuste controlado.
  El modelo CNN-LSTM aprende eficientemente y generaliza bien sin caer en sobreajuste excesivo.

- **Pérdida (entrenamiento y test)**
  La pérdida en entrenamiento disminuye de manera consistente de 0.1739 a 0.0354, y la pérdida de test de 0.2081 a 0.1549.
  Esto indica que el modelo se ajusta progresivamente mejor a los datos con más épocas, y que la mejora en test es constante.

- **Tiempos de entrenamiento y test**
  El tiempo de entrenamiento y test se mantiene muy estable y bajo (alrededor de 27 segundos para entrenamiento y 2.94 segundos para test, incluso con 150 épocas).
  Esto implica que el modelo CNN-LSTM es muy eficiente computacionalmente en comparación con LSTM puro, especialmente considerando que ofrece una precisión incluso mayor en menos tiempo.

Resultados del **CNN-Bi-LSTM:**

Épocas	Training Accuracy	Test Accuracy	Training Loss	Test Loss	Training Time (s)	Testing Time (s)
20	92.40 %	91.32 %	0.1957	0.2383	67.16	5.98
30	94.32 %	93.21 %	0.1475	0.1973	64.91	5.93
50	96.16 %	94.22 %	0.1001	0.1845	61.20	6.03
100	97.91 %	95.65 %	0.0555	0.1613	66.99	5.94
150	98.57 %	96.12 %	0.0383	0.1592	61.35	5.92

Cuadro 7.14: Resultados de entrenamiento y test por número de épocas

Los resultados presentados en la tabla muestran una mejora progresiva y consistente en la precisión tanto en entrenamiento como en test a medida que aumenta el número de épocas:

- La precisión de entrenamiento incrementa desde un 92,40 % en 20 épocas hasta un 98,57 % en 150 épocas.
- La precisión de test también mejora, pasando del 91,32 % al 96,12 % en el mismo rango de épocas.

Esto evidencia que el modelo CNN-BiLSTM aprende de manera efectiva las características discriminativas de las distintas categorías, logrando un buen balance entre ajuste al conjunto de entrenamiento y generalización a datos no vistos.

La pérdida de entrenamiento disminuye de forma significativa desde 0,1957 hasta 0,0383, mostrando una convergencia clara y una mejora en el ajuste del modelo.

La pérdida de test también desciende, aunque de manera más moderada, de 0,2383 a 0,1592, confirmando que el modelo generaliza correctamente y no presenta signos evidentes de sobreajuste.

Respecto a los tiempos computacionales:

El tiempo de entrenamiento se mantiene relativamente estable alrededor de los 60-67 segundos, sin una subida pronunciada al aumentar las épocas, lo que indica un buen rendimiento en la optimización y la gestión de recursos.

El tiempo de test se mantiene constante alrededor de 6 segundos, lo que es aceptable dado que el modelo combina CNN con BiLSTM, arquitecturas que requieren más procesamiento que un MLP simple.

Por último, los resultados del modelo **Bi-LSTM-GN** son los siguientes:

Épocas	Precisión Media (%)	Pérdida de Entrenamiento	Tiempo Entrenamiento (s/época)	Tiempo Test
20	95.88	0.1176	10.16	0.70
30	96.86	0.0879	12.51	0.86
50	97.80	0.0599	12.93	0.90
100	98.22	0.0347	12.71	0.88
150	98.48	0.0258	9.49	0.66

Cuadro 7.15: Resultados del modelo: precisión, pérdida y tiempos por época

El modelo Bi-LSTM-GN muestra un rendimiento excepcional, alcanzando una precisión media de hasta 98,48 % tras 150 épocas, siendo uno de los modelos más precisos evaluados en el estudio. La mejora es progresiva y consistente desde las primeras épocas:

- Desde un 95,88 % con 20 épocas hasta 98,48 % con 150 épocas.
- Esto refleja una curva de aprendizaje muy eficiente, donde el modelo mejora de forma constante sin señales de estancamiento o sobreajuste.

En cuanto a la pérdida de entrenamiento, se observa una disminución clara:

- De 0,1176 en 20 épocas hasta 0,0258 en 150 épocas.
- Esta reducción sostenida confirma que el modelo está convergiendo adecuadamente, minimizando el error de forma efectiva a lo largo del entrenamiento.

Respecto a los tiempos de entrenamiento y test por época, los valores se mantienen en rangos razonables:

- El tiempo de entrenamiento por época oscila entre 9,5 y 13 segundos, lo cual es eficiente considerando que el modelo incorpora una arquitectura bidireccional con normalización por grupos.
- El tiempo de test es muy bajo en todas las épocas, entre 0,66 y 0,90 segundos, lo que hace al modelo viable para aplicaciones donde se requiera rapidez en la inferencia.

La incorporación de Group Normalization puede haber contribuido a la estabilidad del entrenamiento, facilitando una mejor propagación del gradiente, especialmente útil en secuencias largas o con lotes pequeños donde técnicas como Batch Normalization no son tan efectivas.

Al igual que en el análisis por detección de *malware*, también se van a presentar en unas gráficas resumen, los resultados obtenidos en esta sección:

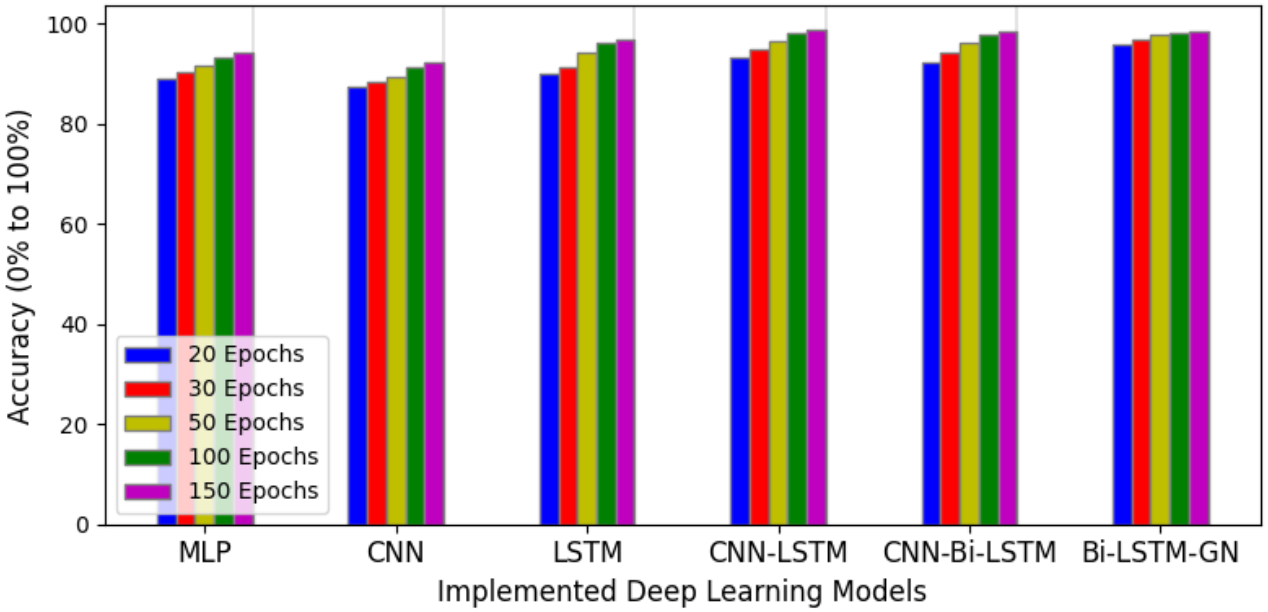

Figura 7.15: Análisis precisión por época

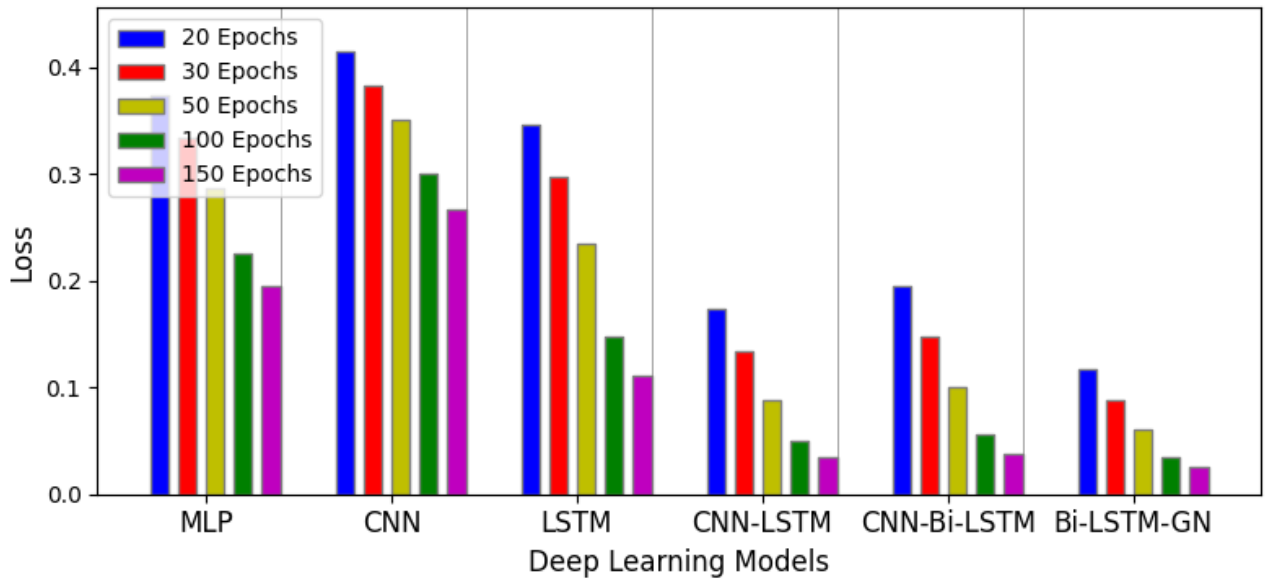

Figura 7.16: Pérdida en detección por época

También se presenta a continuación, una gráfica con el tiempo total, en minutos, que ha tardado cada modelo en cada época en ejecutarse:

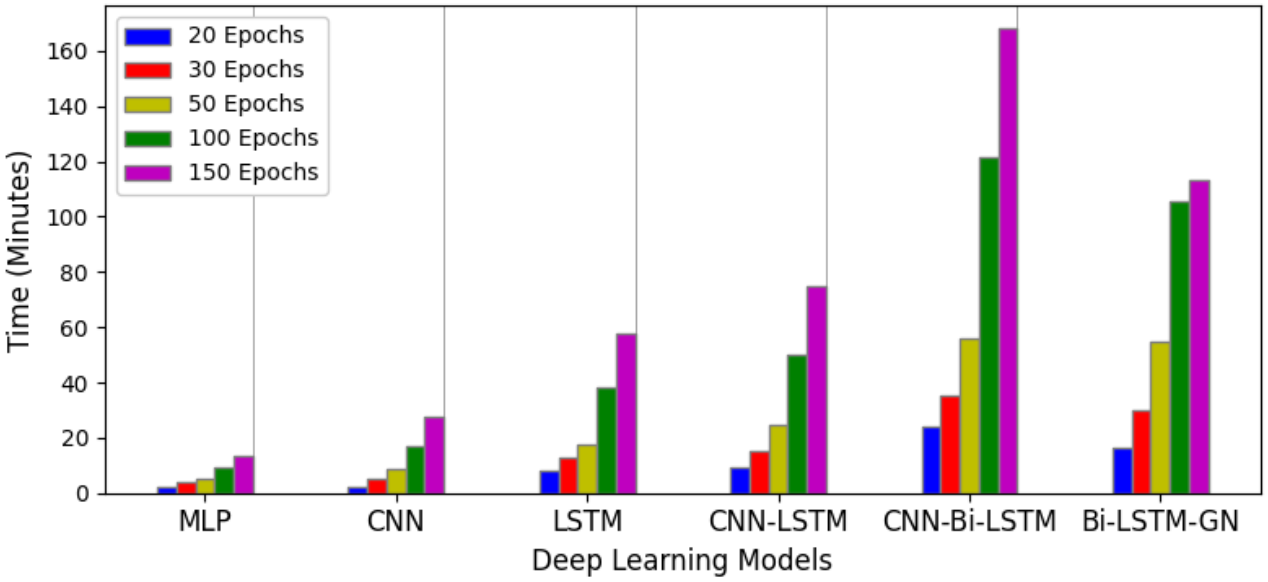

Figura 7.17: Tiempo empleado por modelo y épocas

7.2.3 Atribución por familias

Es importante tener en cuenta que cada familia de *malware* puede presentar patrones, características y técnicas de ofuscación únicas diseñadas para evadir los mecanismos de detección tradicionales. Por ello, el objetivo principal de esta investigación es clasificar correctamente las familias de *ransomware* con alta precisión, lo que representa un desafío clave en ciberseguridad.

El conjunto de datos utilizado contiene un total de 27 clases, por lo que evaluar el rendimiento del modelo propuesto en la clasificación de todas las clases individuales permite comprobar su capacidad para identificar y

atribuir correctamente distintas familias de *malware*.

Siguiendo los mismos parámetros por época empleados previamente en las fases de detección y clasificación, se obtuvieron los siguientes resultados en la clasificación por familia de *ransomware* :

Empezando por el modelo **MLP** se obtuvo lo siguiente:

Épocas	Precisión Media (%)	Pérdida Entrenamiento	Pérdida Test	Tiempo Entrenamiento (s)
20	89.53	0.5157	0.5456	70.67
30	90.99	0.4182	0.4912	104.80
50	92.74	0.3170	0.4483	180.19
100	93.49	0.2144	0.4502	355.93
150	93.83	0.1681	0.5294	531.13

Cuadro 7.16: Resultados del modelo MLP en la tarea de atribución por familias

- El modelo MLP muestra una mejora progresiva en la precisión media a medida que aumenta el número de épocas, aunque con una tendencia de mejora más moderada en las últimas etapas:
  - La precisión avanza desde un 89,53 % con 20 épocas hasta un 93,83 % con 150 épocas.
  - Esto indica que el modelo está aprendiendo las características distintivas de las diferentes familias de *malware*, aunque la ganancia en precisión se reduce tras superar las 100 épocas.
- **En relación con la pérdida**, la disminución en el entrenamiento es clara y significativa:
  - De 0,5157 a 0,1681, lo que indica una mejora consistente en el ajuste del modelo a los datos de entrenamiento.
  - Sin embargo, la pérdida en test, después de una mejora inicial (bajando hasta 0,4483 en 50 épocas), muestra una ligera subida en las últimas épocas, alcanzando 0,5294 en 150 épocas.
  - Este comportamiento podría indicar una leve tendencia al sobreajuste o fluctuación en la capacidad del modelo para generalizar a datos nuevos tras mucho entrenamiento.
- **Los tiempos de entrenamiento** aumentan notablemente conforme se incrementan las épocas:
  - Desde aproximadamente 70 segundos con 20 épocas hasta más de 530 segundos con 150 épocas.
  - El tiempo de test es prácticamente nulo, lo que destaca la rapidez del modelo MLP para inferir una vez entrenado, facilitando su uso en aplicaciones que requieran respuestas rápidas.

Resultados modelo **CNN:**

Cuadro 7.17: Resultados del modelo CNN por número de épocas

Épocas	Mean Accuracy (%)	Training Time (s)	Testing Time (s)	Training Loss	Test Loss
20	90.78	1.56	0.18	0.4554	0.4685
30	91.68	1.55	0.16	0.3941	0.4302
50	92.71	1.62	0.18	0.2960	0.3583
100	93.81	1.60	0.16	0.2192	0.3625
150	93.92	1.55	0.16	0.1794	0.3269

- El modelo CNN muestra una mejora constante y gradual en la precisión media conforme aumenta el número de épocas:
  - La precisión crece desde un 90,78 % con 20 épocas hasta un 93,92 % con 150 épocas.

  - Esta mejora refleja que la CNN es capaz de extraer características relevantes y patrones espaciales o locales que contribuyen a distinguir las familias de *malware* de forma progresiva y estable.

- **Pérdida durante entrenamiento y test:**
  - La pérdida de entrenamiento desciende notablemente, desde 0,4554 en 20 épocas hasta 0,1794 en 150 épocas, mostrando una buena convergencia del modelo.
  - La pérdida en test también disminuye de forma estable, pasando de 0,4685 a 0,3269, lo que indica que el modelo generaliza bien y no hay signos evidentes de sobreajuste, incluso con un número elevado de épocas.

- **Tiempos de entrenamiento y prueba:**
  - Los tiempos de entrenamiento se mantienen muy bajos y casi constantes, alrededor de 1.55 segundos por época, lo cual es muy eficiente comparado con modelos más complejos.
  - El tiempo de prueba es igualmente bajo, en torno a 0.16-0.18 segundos, lo que favorece su implementación en entornos con limitaciones computacionales o que requieren respuestas rápidas.

Resultados modelo **LSTM:**

Epochs	Mean Accuracy	Training Loss	Test Loss
20	89.11 %	0.4997	0.5104
30	91.29 %	0.3879	0.4326
50	92.65 %	0.2877	0.3749
100	93.99 %	0.1804	0.3134
150	93.86 %	0.1311	0.2903

Cuadro 7.18: Precisión y pérdidas del modelo LSTM por número de épocas

Epochs	Training Time (s/epoch)	Testing Time (s/epoch)
20	3311.13	3311.12
30	7052.27	7052.27
50	21267.36	21267.35
100	79760.96	79760.95
150	182203.09	182203.08

Cuadro 7.19: Tiempos del modelo LSTM por número de épocas

- El modelo LSTM muestra una mejora sostenida en la precisión media conforme aumenta el número de épocas:
  - La precisión crece desde un 89.11 % con 20 épocas hasta un máximo de 93.99 % en 100 épocas, manteniéndose estable en 93.86 % a las 150 épocas.
  - Esto indica que el LSTM es efectivo capturando dependencias temporales o secuenciales relevantes para distinguir familias de *malware*, aunque parece estabilizarse después de 100 épocas.

- **Sobre la pérdida:**
  - La pérdida de entrenamiento se reduce significativamente, pasando de 0.4997 a 0.1311, mostrando una convergencia clara del modelo.
  - La pérdida en test también disminuye de manera consistente, de 0.5104 a 0.2903, lo que sugiere una buena capacidad de generalización sin indicios claros de sobreajuste.

- **Sin embargo, un punto crítico del modelo LSTM es su tiempo de entrenamiento y test, que es extremadamente alto:**
  - El tiempo por época crece desde aproximadamente 3,311 segundos en 20 épocas, hasta más de 182,000 segundos (alrededor de 50 horas) en 150 épocas.
  - Los tiempos de entrenamiento y prueba son prácticamente idénticos, lo que indica que la evaluación se realiza con un proceso computacional similar al del entrenamiento, incrementando la carga total.

En cuanto a los resultados del modelo CNN-LSTM se obtuvo:

Epochs	Training Accuracy (%)	Test Accuracy (%)	Training Loss	Test Loss
20	89.93	89.36	0.3129	0.3252
30	92.30	90.33	0.2388	0.3003
50	95.00	91.99	0.1559	0.2706
100	97.33	93.33	0.0829	0.2500
150	98.01	93.69	0.0618	0.2616

Cuadro 7.20: Resultados del modelo CNN-LSTM – Parte 1

Epochs	Training Time (s)	Testing Time (s)
20	15.37	1.66
30	15.40	1.70
50	16.15	1.77
100	15.44	1.72
150	15.33	1.69

Cuadro 7.21: Resultados del modelo CNN-LSTM – Parte 2

- **Precisión**
  - La precisión de entrenamiento aumenta consistentemente, desde un 89.93 % en 20 épocas hasta un 98.01 % en 150 épocas.
  - La precisión en test también mejora, pasando de 89.36 % a 93.69 %, mostrando buena capacidad de generalización.
  - La ganancia de precisión es especialmente notable hasta las 100 épocas, donde alcanza un 93.33 %, estabilizándose luego.
- **Pérdida**
  - La pérdida de entrenamiento disminuye de forma clara y constante, de 0.3129 a 0.0618, indicando que el modelo está aprendiendo eficazmente.
  - La pérdida en test baja de 0.3252 a 0.2500 en 100 épocas, aunque presenta un ligero repunte a 0.2616 en 150 épocas, lo que podría sugerir un inicio muy leve de sobreajuste o simplemente fluctuaciones normales.
- **Tiempos de entrenamiento y test**
  - Los tiempos de entrenamiento por época son bastante estables y razonables, oscilando alrededor de 15 segundos por época.
  - Los tiempos de test son muy bajos (entre 1.66 y 1.77 segundos), lo que indica que el modelo es eficiente en evaluación y podría ser viable para entornos con limitaciones computacionales o aplicaciones en tiempo real.

Para el modelo CNN-Bi-LSTM se obtuvo:

Épocas	**Precisión Entrenamiento (%)**	**Precisión Test (%)**	**Pérdida Entrenamiento**
20	88.80	87.97	0.3270
30	90.96	89.40	0.2588
50	94.39	91.47	0.1635
100	96.32	92.16	0.1064
150	97.74	93.71	0.0663

Cuadro 7.22: Resultados del modelo CNN-Bi-LSTM – Parte 1

Épocas	**Pérdida Test**	**Tiempo Entrenamiento (s)**	**Tiempo Test (s)**
20	0.3602	42.68	4.24
30	0.3309	38.07	3.77
50	0.2933	38.88	4.01
100	0.3193	40.53	4.04
150	0.2717	40.53	4.04

Cuadro 7.23: Resultados del modelo CNN-Bi-LSTM – Parte 2

El modelo **CNN-Bi-LSTM** muestra un comportamiento sólido en términos de precisión, pérdida y tiempos de cómputo a lo largo de diferentes números de épocas:

- **Precisión**
  - La precisión de entrenamiento incrementa de forma constante, comenzando en un 88.80 % a las 20 épocas y alcanzando un 97.74 % a las 150 épocas, lo que indica que el modelo aprende progresivamente y se ajusta bien a los datos de entrenamiento.
  - La precisión en test también mejora continuamente, desde 87.97 % hasta 93.71 %, mostrando una buena capacidad de generalización y que el modelo no presenta un sobreajuste severo.
- **Pérdida**
  - La pérdida de entrenamiento desciende notablemente, desde 0.3270 hasta 0.0663, lo que confirma que el modelo mejora su ajuste a los datos con más entrenamiento.
  - La pérdida en test, aunque presenta un ligero aumento en la época 100 (0.3193), finalmente baja a 0.2717 en 150 épocas, reflejando una mejora global en la generalización.
- **Tiempos de entrenamiento y test**
  - Los tiempos de entrenamiento por época se mantienen bastante estables alrededor de los 40 segundos, un poco más altos comparados con modelos más simples, pero razonables para la complejidad del modelo CNN-Bi-LSTM.
  - Los tiempos de test se mantienen alrededor de 4 segundos, lo cual es adecuado para evaluaciones frecuentes sin impactar demasiado en la eficiencia.

Por último para el modelo **Bi-LSTM-GN** se obtuvo:

Epochs	Accuracy Across Folds (%)	Training Loss	Training Time (s)	Testing Time (s)
20	95.07	0.1265	6.62	0.47
30	95.57	0.0944	6.55	0.45
50	95.57	0.0944	6.55	0.45
100	97.05	0.0365	6.54	0.44
150	97.08	0.0276	6.52	0.44

Cuadro 7.24: Resultados del modelo Bi-LSTM-GN

Este modelo presenta un desempeño muy sólido en cuanto a precisión, pérdida y tiempos, reflejando una buena eficiencia y capacidad de generalización a lo largo de diferentes números de épocas:

En cuanto a la **precision:**

- La precisión media aumenta gradualmente, comenzando en un 95.07 % a las 20 épocas y llegando a un 97.08 % a las 150 épocas.
- Se observa una mejora más notable entre 50 y 100 épocas, donde la precisión salta de 95.57 % a 97.05 %, mostrando que el modelo sigue aprendiendo y afinando su capacidad para clasificar correctamente.

La pérdida por **entrenamiento:**

- La pérdida de entrenamiento disminuye consistentemente, partiendo de 0.1265 a las 20 épocas hasta un mínimo de 0.0276 a las 150 épocas.
- Esto indica que el modelo está ajustándose mejor a los datos con cada época sin sobreajustar, ya que la precisión en test también mejora.

Tiempos de entrenamiento y test:

- Los tiempos de entrenamiento por época se mantienen muy estables y bajos, alrededor de 6.5 segundos, lo que indica una buena eficiencia computacional para un modelo bi-direccional con normalización de grupo.
- Los tiempos de test también son muy reducidos, con valores constantes alrededor de 0.44 segundos, lo que permite evaluaciones rápidas.

Para concluir la sección de resultados de atribución por familia, se puede destacar que todos los modelos evaluados muestran una mejora progresiva en precisión y reducción de pérdida conforme aumenta el número de épocas, evidenciando un aprendizaje efectivo y una buena capacidad de generalización.

Sin embargo, también se observa una variabilidad significativa en los tiempos de entrenamiento, siendo los modelos más complejos los que requieren mayor tiempo computacional. En conjunto, estos resultados confirman que, si bien modelos como CNN y LSTM ofrecen un equilibrio adecuado entre precisión y eficiencia, combinaciones avanzadas como CNN-BiLSTM o BiLSTM-GN pueden aportar una mayor exactitud, especialmente para tareas de clasificación más exigentes, siempre que se cuente con los recursos computacionales adecuados. Esto establece una base sólida para seleccionar el modelo óptimo según los requisitos específicos de precisión y coste temporal en aplicaciones prácticas de detección y atribución de *malware*.

Gráfico resumen de los resultados obtenidos por cada época en la atribución por familias:

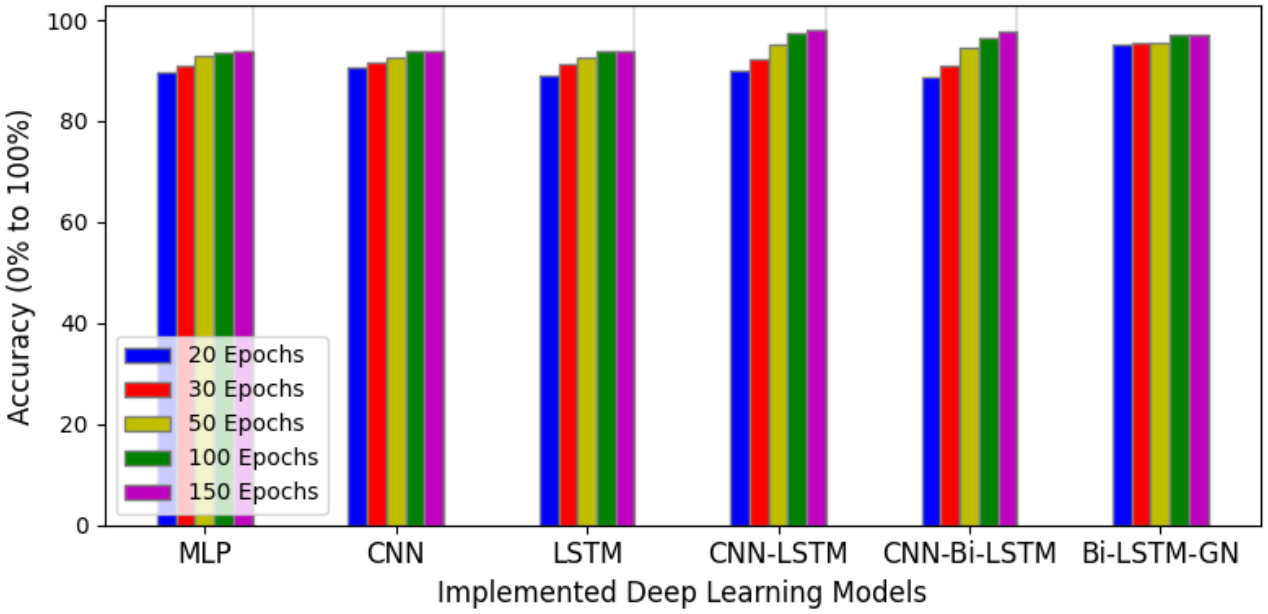

Figura 7.18: Análisis precisión por época

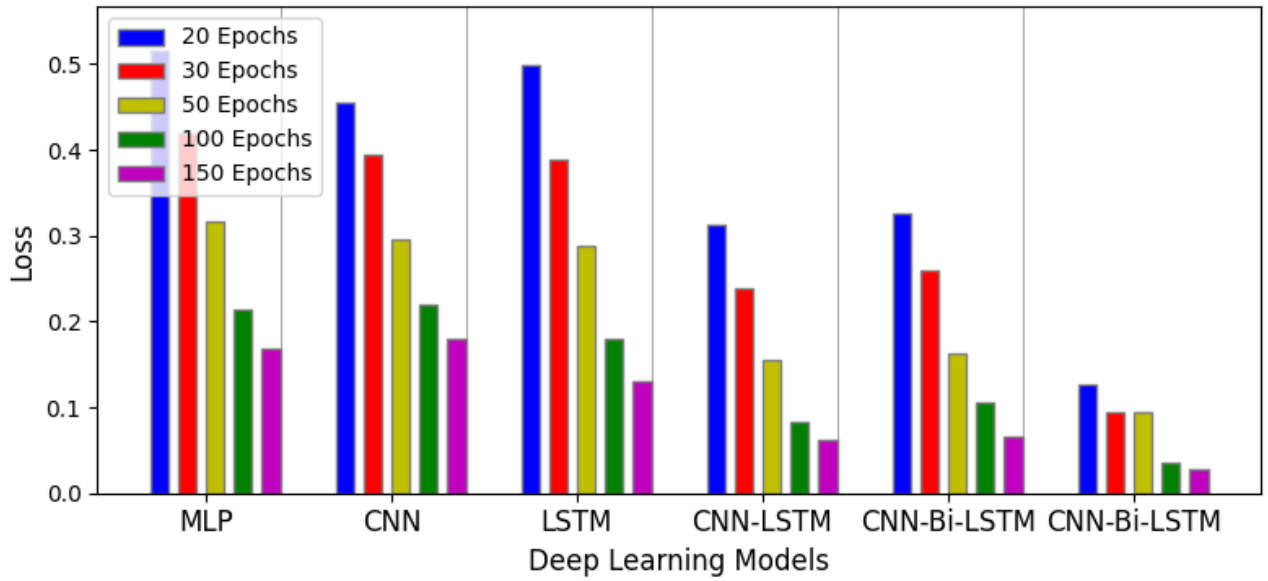

Figura 7.19: Pérdida en detección por época

También se presenta a continuación, una gráfica con el tiempo total, en minutos, que ha tardado cada modelo en cada época en ejecutarse:

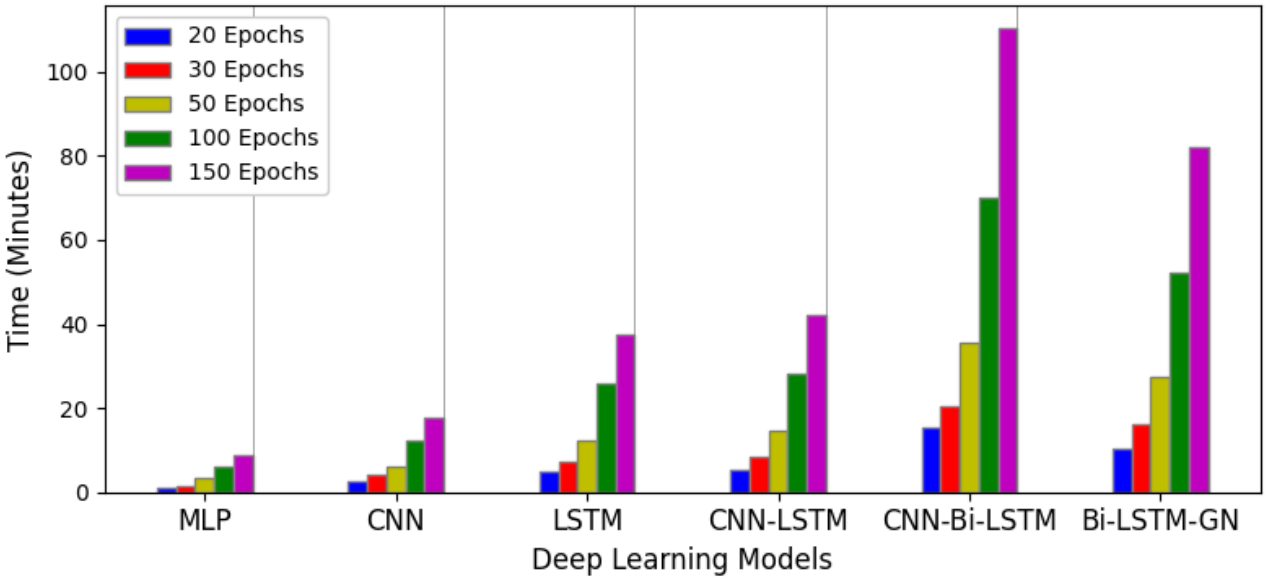

Figura 7.20: Tiempo empleado por modelo y épocas

7.3 Limitaciones identificadas

A lo largo del desarrollo experimental se han identificado varias limitaciones que deben ser tenidas en cuenta a la hora de interpretar los resultados y considerar la aplicación práctica de los modelos propuestos. En primer lugar, aunque modelos más complejos como CNN-LSTM, CNN-Bi-LSTM o BiLSTM-GN han demostrado alcanzar altos niveles de precisión, esto conlleva un coste computacional considerable. Por ejemplo, el modelo LSTM puro mostró tiempos de entrenamiento y prueba extremadamente elevados, lo que podría dificultar su escalabilidad o implementación en entornos en tiempo real o con recursos limitados. En este caso, al realizarse las pruebas en un entorno estático, los tiempos aunque eran largos, se podían respetar.

Asimismo, si bien las arquitecturas híbridas (como CNN-LSTM y CNN-BiLSTM) lograron un buen equilibrio entre precisión y eficiencia temporal, la mejora en los resultados no siempre fue proporcional al aumento de la complejidad del modelo, lo que sugiere la existencia de un punto de retorno decreciente en términos de coste-beneficio. Esto puede limitar su adopción en escenarios donde se requiere un despliegue rápido o se dispone de hardware menos potente.

Otra limitación importante es que, en todas las tareas, el rendimiento del modelo está estrechamente ligado a la calidad y representatividad de los datos utilizados. Aunque se ha trabajado con datasets balanceados y con una adecuada diversidad de muestras, cualquier sesgo en los datos podría afectar significativamente a la capacidad de generalización de los modelos, especialmente en tareas como la atribución por familia, donde algunas clases pueden presentar menor número de ejemplos.

Por último, se debe mencionar que, pese a los buenos resultados en entornos controlados, no se ha evaluado el comportamiento de los modelos frente a *malware* polimórfico, ofuscado o completamente nuevo, lo cual limita parcialmente la aplicabilidad de las soluciones en entornos reales donde estos casos son frecuentes.

Capítulo 8

Propuestas de mejora y optimización

En este capítulo se van a abordar una serie de estrategias que han servido para mejorar en la medida de lo posible los modelos anteriormente descritos. Para ello se van a exponer una serie de mejoras que se han tenido en cuenta para mejorar la precisión de estos.

8.1 Estrategias para mejorar el modelo

8.1.1 Mejora del modelo MLP

```
class ImprovedMLP(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(ImprovedMLP, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.leaky_relu = nn.LeakyReLU(0.01)  # ReLU mejorado
        self.fc2 = nn.Linear(hidden_size, num_classes)

        # Inicialización de pesos (Kaiming para LeakyReLU)
        nn.init.kaiming_uniform_(self.fc1.weight, nonlinearity='leaky_relu')
        nn.init.kaiming_uniform_(self.fc2.weigt, nonlinearity='leaky_relu')

    def forward(self, x):
        x = self.fc1(x)
        x = self.leaky_relu(x)
        x = self.fc2(x)
        return x
```

Comparativa de rendimiento:

Épocas	Accuracy Original (%)	Accuracy Mejorado (%)
20	98.86	98.95
30	98.83	98.89
50	98.91	98.92
100	98.94	98.89
150	98.92	98.86

Cuadro 8.1: Comparación de precisión original y mejorada a diferentes épocas

Uso de Leaky ReLU en lugar de ReLU. ReLU (Rectified Linear Unit) es una de las activaciones más comunes, pero tiene un problema conocido como el ”muerte de neuronas”. Esto ocurre cuando, en la fase de retropropagación, las neuronas tienen valores negativos y no actualizan sus pesos correctamente. Esto puede llevar a que algunas neuronas se queden ”muertasz no aprendan [4].

Leaky ReLU es una versión de ReLU que permite que los valores negativos pequeños fluyan (con un pequeño valor negativo en lugar de ser simplemente 0). Esto evita que las neuronas se queden atrapadas en el régimen de no-aprendizaje, lo que puede mejorar la convergencia del modelo [4].

En el código mejorado, se está usando LeakyReLU(0.01), lo que significa que la pendiente para valores negativos es de 0.01. Esto facilita que la red aprenda más rápido y mantenga una mejor propagación de gradientes.

Inicialización de pesos con Kaiming (He) para Leaky ReLU. Inicialización de Kaiming (He initialization) es un tipo de inicialización de pesos que fue específicamente diseñada para redes con ReLU y sus variantes como Leaky ReLU. En esta inicialización, los pesos se distribuyen de manera que el valor de varianza se mantiene aproximadamente constante a través de las capas de la red.

Esto mejora la propagación del gradiente, ya que evita que los valores de los gradientes se vuelvan demasiado grandes o pequeños, lo que podría llevar a un entrenamiento ineficiente.

La inicialización `nn.init.kaiming_uniform_()` se usa para distribuir los pesos de manera uniforme, y el parámetro `nonlinearity='leaky_relu'` asegura que esta inicialización se haga de manera específica para Leaky ReLU.

El impacto que tiene Leaky ReLU en el rendimiento es que permite que las neuronas sigan aprendiendo incluso cuando tienen valores negativos, lo que puede mejorar la capacidad de la red para aprender de los datos.

La inicialización de Kaiming asegura que los pesos de las capas sean adecuados para evitar problemas de gradientes explotados o desvanecidos, lo que permite un entrenamiento más rápido y estable.

Ambos cambios hacen que el modelo mejor gestione las activaciones y los gradientes durante el entrenamiento, lo que puede resultar en una mayor precisión, como lo has visto con el aumento al 98.95

8.1.2 Mejora del modelo CNN

Para llevar a cabo la mejora en el modelo CNN, a diferencia de la optimización llevada a cabo en otros modelos en la que optimizábamos el modelo en sí, en este caso se va a optar por una mejora en la normalización de los datos. Lo que va a traer consigo una mejora en la precisión del modelo.

Para ello se ha normalizado de la siguiente manera:

```
#MEJORA

from sklearn.preprocessing import LabelEncoder
# Initialize the label encoder
label_encoder = LabelEncoder()

# Assuming you have a DataFrame named 'df'
categorical_columns = df.select_dtypes(include=['object', 'category']).columns
categorical_columns =[ 'Class']
# Apply label encoding to each categorical column
for col in categorical_columns:
    df[col] = label_encoder.fit_transform(df[col])
```

```
#MEJORA

from sklearn.preprocessing import MinMaxScaler, StandardScaler
# Create a Min-Max scaler instance
scaler = StandardScaler()
# Select the columns you want to scale (exclude the target variable if needed)
columns_to_scale = df.select_dtypes(include=[np.number]).columns
# Fit the scaler on the selected columns and transform the data
df[columns_to_scale] = scaler.fit_transform(df[columns_to_scale])
```

Comparativa del rendimiento:

Épocas	Accuracy Original (%)	Accuracy Mejorado (%)
20	98,88	98,93
30	98,92	98,89
50	98,98	99,02
100	99,09	99,12
150	99,11	99,15

Cuadro 8.2: Comparación de precisión original y mejorada en distintas épocas

Esto mejora el rendimiento por diversas razones:

1. **Selección más robusta de columnas numéricas**
   - `include=[np.number]` detecta cualquier tipo numérico (int, float, etc.), mientras que `'float64'` y `'int64'` son más restrictivos.
   - Esto significa que puedes estar omitiendo columnas que son numéricas pero no estrictamente `float64` o `int64` en la versión anterior (por ejemplo, `int32`, `float32`, etc.), y no se escalan, lo que desequilibra los rangos de entrada para la CNN.
2. **Separación clara del `target` (Class)**
   - En la versión mejorada, se hace explícitamente: `categorical_columns = ['Class']` y se aplica `LabelEncoder` solo al `target`.
   - En la versión antigua, se podría estar escalando accidentalmente la columna de la clase si no se ha excluido, lo que confunde al modelo, ya que espera clases como enteros (0, 1, 2...) pero se le da algo escalado como 0.5, -1.3, etc.
3. **Uso correcto del escalado previo a la CNN**
   - Las CNN son muy sensibles a la escala de los datos de entrada, especialmente cuando se usan activaciones como `ReLU`.
   - Escalar adecuadamente todas las variables numéricas a una distribución estándar (media 0, desviación 1) ayuda a que la red converja mejor.
4. **Tratamiento específico de variables categóricas**
   - Las CNN no pueden trabajar con `strings`. Si `Class` no es numérica o está en formato texto (e.g. `'Normal'`, `'Ataque'`), debe ser convertida a enteros primero.
   - La versión mejorada lo hace correctamente con `LabelEncoder`.

Es importante normalizar bien por lo siguiente:

- Datos mejor normalizados → pesos más estables → menor pérdida.
- Etiquetas codificadas correctamente → cálculo de pérdida correcto (CrossEntropyLoss requiere long labels, no floats).
- Datos numéricos bien escalados → mejor entrenamiento y generalización.

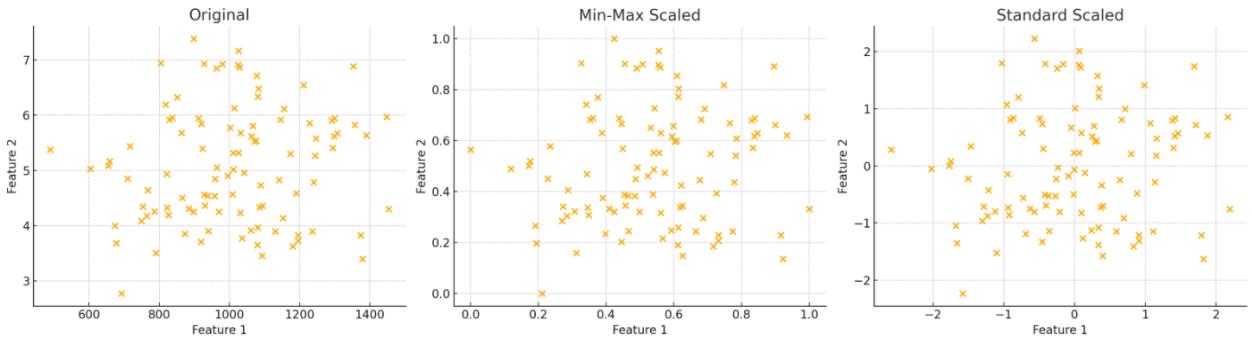

Figura 8.1: Diferentes tipos de escalados

En base al gráfico 8.1 obtenido antes, se obtienen las siguientes conclusiones:

- **Diferencias en las escalas originales:**
  - En el gráfico de la izquierda (Original), la `Feature 1` tiene valores alrededor de 1000, mientras que `Feature 2` está alrededor de 5. Esto crea una gran disparidad en magnitudes.
  - Las redes neuronales (como las CNN) aprenden mediante multiplicaciones de pesos y sumas. Si una característica tiene un rango mucho mayor que otra, dominará el aprendizaje, haciendo que la red ignore otras características.
- **StandardScaler (gráfico de la derecha):**
  - Ajusta los datos para que cada característica tenga media 0 y desviación estándar 1.
  - Esto ayuda a que todas las características tengan un impacto similar en el entrenamiento, facilitando una convergencia más estable y rápida.
- **MinMaxScaler (gráfico del medio):**
  - Escala los datos al rango [0, 1].
  - Aunque útil en algunos casos (como en imágenes), puede ser menos robusto si hay *outliers*, y tampoco garantiza media 0 ni varianza 1, lo que puede ser menos óptimo para algoritmos sensibles a la escala como una CNN.

8.1.3 Mejora del modelo Bi-LSTM-GN

Por último, el modelo Bi-LSTM-GN se ha optimizado de la siguiente manera:

```
1
2  class BiLSTM(nn.Module):
3      def __init__(self, input_size, hidden_size, num_layers, num_classes):
4          super(BiLSTM, self).__init__()
5          self.lstm = nn.LSTM(input_size, hidden_size, num_layers, batch_first=True,
       bidirectional=True)
6          self.group_norm = nn.GroupNorm(num_groups=8, num_channels=2 * hidden_size)
7          self.dropout = nn.Dropout(0.2)
8          self.fc = nn.Sequential(
9              nn.Linear(2 * hidden_size, hidden_size),
10             nn.ReLU(),
11             nn.Linear(hidden_size, num_classes)
12         )
13
14     # Dentro del modelo (en el forward):
```

```
    def forward(self, x):
        if x.dim() == 2:
            x = x.unsqueeze(1)  # [32, 50] -> [32, 1, 50]

        lstm_out, _ = self.lstm(x)  # Ahora sí es 3D
        lstm_out = lstm_out[:, -1, :]  # Selecciona el único paso temporal
        lstm_out = self.group_norm(lstm_out)
        output = self.fc(lstm_out)
        return output
```

Tabla comparativa:

Épocas	Accuracy Original (%)	Accuracy Mejorado (%)
20	98,96	98,89
30	99,11	99,11
50	99,12	99,17
100	99,31	99,31
150	99,21	99,31

Cuadro 8.3: Comparación de precisión original y mejorada a diferentes épocas

si nos fijamos en la siguiente sección del código anterior:

```
lstm_out = lstm_out[:, -1, :]  # Se queda con [batch_size, 2*hidden_size]
lstm_out = self.group_norm(lstm_out)
```

Aquí se toma solo el último paso temporal, que es un tensor 2D [batch_size, 2*hidden_size], y luego se aplica GroupNorm, a diferencia del codigo original que lo que hace es que GroupNorm se aplica directamente a lstm_out, que es un tensor 3D con forma [batch_size, seq_len, features]. Pero GroupNorm espera como entrada un tensor de forma 2D o 4D (batch_size, channels, ...).

También al añadir dropout:

```
    self.dropout = nn.Dropout(0.2)
```

Ayuda a prevenir el sobreajuste, especialmente en modelos pequeños entrenados con pocos datos.

También al añadirle un clasificador secuencial, esto le permite al modelo aprender representaciones no lineales más complejas, algo útil para clasificaciones donde las clases no están separadas linealmente.

Ventaja: El clasificador puede aprender patrones más sofisticados antes de la predicción final.

Un aspecto que mejora tambien el rendimiento es el uso de entradas 2D:

```
    if x.dim() == 2:
    x = x.unsqueeze(1)  # [batch_size, input_size] -> [batch_size, 1, input_size]
```

Este detalle hace que el modelo sea más tolerante con entradas que no vengan en el formato esperado, algo útil si el preprocesado cambia o si se reusa el modelo en otro entorno.

Capítulo 9

Técnicas de Mitigación

En el resto de secciones se ha estado hablando sobre todas las estrategias de detección de *ransomware* , junto con sus clasificaciones etc. Ahora el enfoque va a ser distinto, en esta sección, la idea es mitigar estos ataques en caso de que las técnicas de detección hayan fallado.

En esta sección se presentan las principales estrategias y buenas prácticas diseñadas para mitigar los efectos de un posible ataque de *ransomware* , abarcando desde medidas preventivas y de protección proactiva hasta mecanismos de respuesta y recuperación tras la intrusión. Se abordarán técnicas orientadas a reforzar la seguridad perimetral, minimizar la superficie de ataque y asegurar la disponibilidad de la información, así como procedimientos para restaurar los sistemas y archivos comprometidos. El objetivo es proporcionar un conjunto de directrices integrales que permitan a las organizaciones reducir el riesgo de infección, limitar el impacto de un incidente y garantizar una recuperación rápida y eficiente.

La prevención de *ransomware* se centra en medidas proactivas dirigidas a reducir el riesgo de ataques al subsanar vulnerabilidades antes de que puedan ser explotadas. Entre las estrategias más comunes se incluyen la actualización de sistemas operativos, el uso de software de seguridad especializado y el mantenimiento de copias de seguridad regulares de los archivos. El objetivo principal en esta fase es identificar y corregir posibles fallos de seguridad que puedan ser aprovechados por los atacantes de *ransomware* .

Uno de los retos clave en la prevención de *ransomware* es rastrear el origen de los ataques, especialmente aquellos que implican extorsión de datos o «secuestro» de información, ya que suelen dificultar la identificación de los perpetradores. Las medidas de prevención eficaces permiten a los usuarios evitar infecciones o recuperar sus archivos, interrumpiendo así el círculo de ataques. A continuación, se detallan las principales acciones recomendadas para mitigar el riesgo de ataques de *ransomware* :

- **Copias de seguridad regulares de datos:** Realizar copias de seguridad periódicas y almacenarlas fuera del sitio es esencial para restaurar rápidamente los archivos en caso de cifrado por *ransomware* . No obstante, muchas organizaciones enfrentan desafíos en cuanto al tiempo y coste de estos procesos, ya que algunas copias requieren gran capacidad de almacenamiento y pueden ralentizar el rendimiento del sistema. Mantener sistemas de respaldo fiables y eficientes es crucial, a pesar de los recursos y tiempos invertidos .

- **Precaución con archivos adjuntos de correo electrónico:** Debe evitarse abrir archivos adjuntos no solicitados, ya que este es uno de los vectores de entrega de *ransomware* más frecuentes.

- **Limitar el acceso con privilegios de administrador:** Para minimizar el riesgo de infección, conviene evitar sesiones prolongadas con permisos de administrador y restringir la navegación por Internet o la apertura de documentos mientras se utilizan dichos privilegios.

- **Concienciación sobre ingeniería social:** Mantenerse alerta ante enlaces maliciosos en redes sociales y aplicaciones de mensajería, incluso si aparentan proceder de contactos de confianza.
- **Configuración de cortafuegos y ajustes de seguridad:** Es fundamental asegurar el correcto funcionamiento del cortafuegos (por ejemplo, Windows Firewall) y configurar medidas adicionales, como el bloqueo de direcciones IP maliciosas, para reforzar la protección frente al *ransomware* .
- **Uso de software antivirus y antimalware:** Instalar soluciones de antivirus y anti*malware* de reconocido prestigio y realizar análisis periódicos es un método eficaz para detectar y eliminar amenazas antes de que causen daños.
- **Seguro contra ciberdelincuencia:** Dado que los ataques de *ransomware* continúan proliferando y generando pérdidas económicas significativas —incluso llevando a empresas a la bancarrota o a la retirada de inversores—, el seguro contra ciberdelitos se ha convertido en un elemento clave para mitigar el riesgo financiero asociado a estos incidentes.

En el panorama de amenazas actual, contar con software anti*ransomware* especializado resulta vital para una protección integral. Las herramientas más avanzadas son capaces de detectar comportamientos sospechosos, ofrecer defensa proactiva y proporcionar mecanismos de recuperación de archivos. Muchos de estos productos incluyen análisis forense y de comportamiento, lo que les permite bloquear e incluso descifrar archivos cifrados.

Aunque el *ransomware* suele emplear técnicas de cifrado robustas como AES256 y RSA-2048, en ocasiones presenta vulnerabilidades de implementación que permiten a los analistas extraer claves de cifrado o descifrar directamente los archivos. Iniciativas como NoMoreRansom —una colaboración entre Europol, Kaspersky Lab y otras entidades,[46] ofrecen herramientas de descifrado gratuitas para las víctimas, aprovechando estas debilidades identificadas en variantes específicas de *ransomware* .

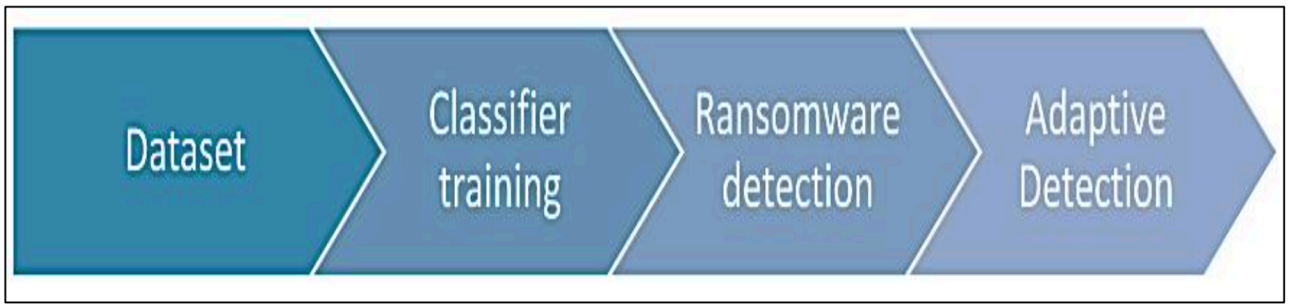

Figura 9.1: Importancia del dataset en la detección del *ransomware*

La figura 9.1 muestra cómo la calidad del conjunto de datos repercute directamente en el desarrollo de un modelo de detección adaptable, subrayando la necesidad de contar con información fiable y veraz. Los conjuntos de datos de alta calidad permiten entrenar modelos robustos y precisos, lo que resulta fundamental para disponer de soluciones eficaces en la lucha contra el *ransomware* .

Capítulo 10

Conclusiones

En este capítulo se presentan las conclusiones extraídas tras la finalización del presente Trabajo de Fin de Grado. El objetivo principal del proyecto ha sido realizar una investigación sobre las técnicas que se pueden utilizar para detectar *ransomware* basadas en el diseño, implementación y evaluación de diversos modelos de aprendizaje profundo para la detección y clasificación de muestras de *malware*, con un enfoque especial en la detección, clasificación y atribución por familia de *malware*. Como objetivo secundario se ha propuesto una serie de estrategias de mitigación para abordar el problema anterior.

Desde el comienzo del trabajo, uno de los mayores retos ha sido como llevar a cabo el preprocesamiento y conversión de las muestras de *malware* a un formato adecuado para ser tratado por modelos de deep learning. Inicialmente no se tenía una idea clara de cómo abordar esta fase, lo que generó cierta incertidumbre. Sin embargo, tras una búsqueda exhaustiva de trabajos previos y el análisis de distintos artículos científicos relacionados, fue posible identificar enfoques ya utilizados que sirvieron como guía. Esto permitió enfocar correctamente el preprocesamiento, analizar en profundidad los datos y seleccionar las herramientas más adecuadas. Una vez superada esta etapa, se entrenaron y compararon distintos modelos (MLP,CNN,LSTM, Bi-LSTM, CNN-BiLSTM y Bi-LSTM-GN), evaluando su rendimiento mediante métricas estándar y pruebas cruzadas, con el fin de seleccionar el enfoque más eficaz.

Los resultados obtenidos han demostrado que la arquitectura Bi-LSTM-GN ha ofrecido un rendimiento especialmente alto en términos de precisión y estabilidad entre épocas, destacando también por sus bajos tiempos de entrenamiento y test. Sin embargo, el modelo CNN-BiLSTM mostró una mejor capacidad en la atribución por familia, especialmente con un número mayor de épocas, lo cual resulta especialmente relevante en entornos donde es necesario no solo detectar *malware*, sino también entender su origen o características comunes.

En general, se ha cumplido el objetivo de desarrollar una solución capaz de detectar y clasificar *malware* con altas tasas de precisión, superando el 97 % en algunos casos, así como de analizar la atribución por familia con resultados consistentes. Además, se ha obtenido una experiencia práctica muy valiosa en el uso de herramientas de desarrollo en Python, bibliotecas de deep learning como TensorFlow y PyTorch, así como en la gestión de experimentación computacional en entornos controlados.

Pese a los buenos resultados, también se han identificado limitaciones como la necesidad de un mayor volumen de datos etiquetados para mejorar la generalización, o el ajuste manual de hiperparámetros que podría automatizarse en futuros trabajos. Asimismo, sería interesante estudiar la implementación de técnicas de aprendizaje federado o incremental, así como probar el sistema en entornos reales o sobre muestras en tiempo real, lo que daría una visión más completa de su aplicabilidad práctica.

También se han expuesto una serie de contramedidas para mitigar los efectos que los archivos maliciosos detectados con los modelos puedan tener sobre los sistemas informáticos.

Para finalizar, este trabajo ha permitido avanzar en el diseño de soluciones inteligentes para la ciberseguridad, y abre la puerta a futuras líneas de investigación centradas en la mejora de detección y clasificación de *malware*, para poder prevenir cualquier tipo de ataque que deje el sistema muy vulnerable.

10.1 Trabajo futuro

Aunque el desarrollo y evaluación del presente sistema de detección y atribución de *malware* ha resultado satisfactorio, existen múltiples líneas de trabajo que podrían ampliarse o mejorarse en investigaciones futuras.

En primer lugar, uno de los aspectos que podría potenciarse es la ampliación y diversificación del conjunto de datos. A pesar de que el dataset utilizado ha permitido entrenar y validar los modelos con buenos resultados, incorporar muestras más recientes, así como variantes más sofisticadas de *malware*, podría ayudar a mejorar la capacidad de generalización del sistema y adaptarse a amenazas más actuales.

Asimismo, sería interesante explorar arquitecturas más avanzadas, como Transformers o modelos híbridos que combinen aprendizaje supervisado con técnicas de aprendizaje no supervisado o auto-supervisado. Estas aproximaciones podrían ser especialmente útiles para detectar *malware* polimórfico o variantes desconocidas, donde las diferencias con el *malware* ya etiquetado son mínimas o difíciles de detectar con modelos tradicionales.

Otra línea de mejora sería optimizar el proceso de preprocesamiento y generación de imágenes a partir del código binario. Aunque se ha logrado una representación efectiva, aún se podrían explorar métodos que conserven mayor cantidad de información contextual o estructural del *malware* sin incrementar en exceso el coste computacional.

Además, se podría plantear la implementación de un sistema de detección en tiempo real o semi-tiempo real, integrando los modelos entrenados en un entorno práctico, como un motor antivirus o un sistema de monitorización de red. Esto permitiría evaluar su rendimiento en escenarios del mundo real, donde las limitaciones de tiempo y recursos cobran un papel importante.

Por último, también sería valioso incluir mecanismos de interpretabilidad en los modelos utilizados. Comprender por qué un modelo toma ciertas decisiones ayudaría no solo a mejorar la confianza en los resultados, sino también a identificar características comunes en distintas familias de *malware*, contribuyendo así a estudios más profundos sobre su comportamiento y evolución.

Bibliografía

[1] Rahali A et al. *Didroid: android malware classification and characterization using deep image learning. In: Proceedings of the 2020 10th international conference on communication and network security. 70–82.* 2020. (Visitado 23-05-2025).

[2] Yazdinejad A et al. *Cryptocurrency malware hunting: a deep recurrent neural network approach. Applied Soft Computing*. 2020. URL: https://www.sciencedirect.com/science/article/abs/pii/S1568494620305688?via%3Dihub (visitado 23-05-2025).

[3] Darem AA et al. *An adaptive behavioral-based incremental batch learning malware variants detection model using concept drift detection and sequential deep learning*. 2021. URL: https://ieeexplore.ieee.org/document/9467300 (visitado 23-05-2025).

[4] *Activation functions: ReLU vs. Leaky ReLU Srikari Rallabandi*. URL: https://medium.com/@sreeku.ralla/activation-functions-relu-vs-leaky-relu-b8272dc0b1be (visitado 23-05-2025).

[5] Celdrán AH et al. *Intelligent and behavioral-based detection of malware in IoT spectrum sensors. International Journal of Information Security*. 2023. (Visitado 23-05-2025).

[6] Lashkari AH et al. *Toward developing a systematic approach to generate benchmark android malware datasets and classification. In: 2018 International Carnahan conference on security technology (ICCST). Piscataway: IEEE, 1–7.* 2018. (Visitado 23-05-2025).

[7] Logothetis MD Akbanov M Vassilakis VG. *WannaCry ransomware: anal- ysis of infection, persistence, recovery prevention and propagation mecha- nisms. Journal of Telecommunications and Information Technology*. 2019. (Visitado 23-05-2025).

[8] Amjad Hussain Amjad Hussain. *Ransomware Dataset 2024*. Ver. 1.0. Zenodo, oct. de 2024. DOI: 10.5281/zenodo.13890887. URL: https://doi.org/10.5281/zenodo.13890887.

[9] Amjad Hussain Amjad Hussain. *Ransomware Dataset Balanced 2024*. Oct. de 2024. URL: https://github.com/khbdevelopers/Enhancing-Ransomware-Defense-Detection-and-Classification-of-Ransomware/blob/main/Self%20Created%20Dataset/Classification/balanced.csv.

[10] Amjad Hussain Amjad Hussain. *Ransomware Dataset Family 2024*. Oct. de 2024. URL: https://github.com/khbdevelopers/Enhancing-Ransomware-Defense-Detection-and-Classification-of-Ransomware/blob/main/Self%20Created%20Dataset/Family_Attribution/family.csv.

[11] *Analyzing Reporting on Ransomware Incidents: A Case Study*. URL: https://www.mdpi.com/2076-0760/12/5/265 (visitado 23-05-2025).

[12] Samet R Aslan ÖA. *A comprehensive review on malware detection approaches*. 2020. (Visitado 23-05-2025).

[13] Hwang C et al. *Platform-independent malware anal- ysis applicable to windows and linux environments*. 2020. URL: https://www.mdpi.com/2079-9292/9/5/793 (visitado 23-05-2025).

[14] *Clasificador Random Forest: que es y como funciona*. URL: https://kopuru.com/clasificador-random-forest-que-es-y-como-funciona/ (visitado 23-05-2025).

[15] *CNN Bidirectional LSTM*. URL: https://paperswithcode.com/method/cnn-bilstm (visitado 23-05-2025).

[16] *Cross Industry Standard Process for Data Mining*. (Visitado 23-05-2025).

[17] Canadian Institute for Cybersecurity. *Malware memory analysis (CIC-MalMem-2022)*. University of New Brunswick, 2022. URL: https://www.unb.ca/cic/datasets/malmem-2022.html.

[18] Orman A Dener M Ok G. *Malware detection using memory analysis data in big data environment. Applied Sciences*. 2022. URL: https://www.mdpi.com/2076-3417/12/17/8604 (visitado 23-05-2025).

[19] Keyes DS et al. *EntropLyzer: android malware classification and characterization using entropy analysis of dynamic characteristics. In: 2021 Reconciling data analytics, automation, privacy, and security: a big data challenge (RDAAPS). Piscataway: IEEE, 1–12*. 2021. (Visitado 23-05-2025).

[20] Khan F et al. *A digital DNA sequenc- ing engine for ransomware detection using machine learning*. 2020. URL: https://ieeexplore.ieee.org/document/9121260 (visitado 23-05-2025).

[21] Report F. *Fortinent report 2023 on ransomware global research*. 2023. URL: https://www.fortinet.com/content/dam/fortinet/assets/reports/report-2023-ransomware-%20global-research.pdf (visitado 23-05-2025).

[22] Alarfaj FK et al. *Credit card fraud detection using state-of-the-art machine learning and deep learning algorithms. IEEE Access*. 2022. URL: https://ieeexplore.ieee.org/document/9755930 (visitado 23-05-2025).

[23] Towsley D Garetto M Gong W. *Modeling malware spreading dynamics. In: IEEE INFOCOM 2003. Twenty-Second annual joint conference of the IEEE computer and communications societies*. 2003. (Visitado 23-05-2025).

[24] *Google Colab Documentation*. URL: https://colab.research.google.com/ (visitado 23-05-2025).

[25] Naeem H et al. *Development of a deep stacked ensemble with process based volatile memory forensics for platform independent mal- ware detection and classification. Expert Systems with Applications*. 2023. URL: https://www.sciencedirect.com/science/article/abs/pii/S0957417423004542?via%3Dihub (visitado 23-05-2025).

[26] Amjad Hussain et al. «Enhancing ransomware defense: deep learning-based detection and family-wise classification of evolving threats». En: *PeerJ Computer Science* (2024).

[27] IBM. *¿Qué es el aprendizaje no supervisado?* URL: https://www.ibm.com/mx-es/topics/unsupervised-learning (visitado 23-05-2025).

[28] IBM. *¿Qué es el aprendizaje profundo?* URL: https://www.ibm.com/mx-es/think/topics/deep-learning (visitado 23-05-2025).

[29] IBM. *¿Qué es el aprendizaje semisupervisado?* URL: https://www.ibm.com/es-es/think/topics/semi-supervised-learning (visitado 23-05-2025).

[30] IBM. *¿Qué es el aprendizaje supervisado?* URL: https://www.ibm.com/es-es/topics/supervised-learning (visitado 23-05-2025).

[31] *Introducción a las redes neuronales convolucionales (CNN)*. URL: https://www.datacamp.com/es/tutorial/introduction-to-convolutional-neural-networks-cnns (visitado 23-05-2025).

[32] *Introducción al concepto de LSTM*. URL: https://datascience.eu/es/aprendizaje-automatico/comprension-de-las-redes-de-lstm/ (visitado 23-05-2025).

[33] Palša J et al. *Mlmd—a malware-detecting antivirus tool based on the xgboost machine learning algorithm. Applied Sciences*. 2022. (Visitado 23-05-2025).

[34] Shayma Jawad y Hanaa Mohsin Ahmed. «Machine Learning Approaches to Ransomware Detection: A Comprehensive Review.» En: *International Journal of Safety & Security Engineering* 14.6 (2024).

[35] Lee K Kim Yu-kyung JJLM-HGHYK. *systematic overview of the machine learning methods for mobile malware detection. Security and Communication Networks*. 2022. URL: https://onlinelibrary.wiley.com/doi/10.1155/2022/8621083 (visitado 23-05-2025).

[36] *Latex Documentation*. URL: https://es.wikipedia.org/wiki/LaTeX (visitado 23-05-2025).

[37] Trajković L Li Z Rios ALG. *Machine learning for detecting the WestRock ransomware attack using BGP routing records*. 2022. URL: https://ieeexplore.ieee.org/document/9970356 (visitado 23-05-2025).

[38] *LightGBM (Light Gradient Boosting Machine)*. URL: https://www.geeksforgeeks.org/lightgbm-light-gradient-boosting-machine/ (visitado 23-05-2025).

[39] Abualhaj M et al. *Customized K-nearest neighbors' algorithm for malware detection. International Journal of Data and Network Science*. 2023. URL: https://www.growingscience.com/ijds/Vol8/ijdns_2023_160.pdf (visitado 23-05-2025).

[40] Ficco M. *Malware analysis by combining multiple detectors and observation windows. IEEE Transactions on Computers*. 2021. URL: https://ieeexplore.ieee.org/document/9435928 (visitado 23-05-2025).

[41] Al-Qudah M et al. *Effective one-class classifier model for memory dump malware detection. Journal of Sensor and Actuator Networks*. 2023. URL: https://www.mdpi.com/2224-2708/12/1/5 (visitado 23-05-2025).

[42] Mail MAE, Ab Razak MF y Ab Rahman M. *Malware detection system using cloud sandbox, machine learning. International Journal of Software Engineering and Computer Systems*. 2022. URL: https://journal.ump.edu.my/ijsecs/article/view/7481/2374 (visitado 23-05-2025).

[43] Burget R Mezina A. *Obfuscated malware detection using dilated convolutional network. In: 2022 14th international congress on ultra modern telecommunications and control systems and workshops (ICUMT). Piscataway: IEEE, 110–115*. 2022. (Visitado 23-05-2025).

[44] *Microsoft Documentation*. URL: https://es.wikipedia.org/wiki/Microsoft (visitado 23-05-2025).

[45] *Microsoft Word Documentation*. URL: https://word.cloud.microsoft/es-es/ (visitado 23-05-2025).

[46] *No More Ransom – do you need help unlocking your digital life?* URL: https://www.europol.europa.eu/operations-services-and-innovation/public-awareness-and-prevention-guides/no-more-ransom-do-you-need-help-unlocking-your-digital-life (visitado 23-05-2025).

[47] Savenko O et al. *Dynamic signature-based malware detection technique based on API call tracing. In: ICTERI workshops. 633–643*. 2019. (Visitado 23-05-2025).

[48] *Overleaf Documentation*. URL: https://www.overleaf.com/learn (visitado 23-05-2025).

[49] *Perceptrones multicapa en el aprendizaje automático*. URL: https://www.datacamp.com/es/tutorial/multilayer-perceptrons-in-machine-learning?dc_referrer=https%3A%2F%2Fduckduckgo.com%2F (visitado 23-05-2025).

[50] Dasgupta D Poudyal S. *Analysis of crypto-ransomware using ML-based multi- level profiling*. 2021. URL: https://ieeexplore.ieee.org/document/9526633 (visitado 23-05-2025).

[51] *Preprocesamiento de datos*. URL: https://techlib.net/techedu/preprocesamiento-de-datos/ (visitado 23-05-2025).

[52] *Python Documentation*. URL: https://docs.python.org/3/ (visitado 23-05-2025).

[53] *Python Foundation*. URL: https://es.wikipedia.org/wiki/Python_Software_Foundation (visitado 23-05-2025).

[54] *Python Software Foundation License*. URL: https://es.wikipedia.org/wiki/Python_Software_Foundation_License (visitado 23-05-2025).

[55] Dang Q-V. *Enhancing obfuscated malware detection with machine learning tech- niques. In: Dang TK, Küng J, Chung TM, eds. Future data and security engineering. Big data, security and privacy, smart city and industry 4.0 applications. FDSE 2022. Communications in computer and information science, vol. 1688. Singapore: Springer*. 2022. URL: https://link.springer.com/chapter/10.1007/978-981-19-8069-5_54 (visitado 23-05-2025).

[56] Brito M. Rawson A. *A survey of the opportunities and challenges of supervised machine learning in maritime risk analysis. Transport Reviews*. 2023. URL: https://www.tandfonline.com/doi/full/10.1080/01441647.2022.2036864 (visitado 23-05-2025).

[57] North MM Richardson R. *Ransomware: evolution, mitigation and prevention. International Management Review*. 2017. (Visitado 23-05-2025).

[58] Molina RMA et al. *On ran- somware family attribution using pre-attack paranoia activities. IEEE Transactions on Network and Service Management*. 2021. URL: https://ieeexplore.ieee.org/document/9536608 (visitado 23-05-2025).

[59] Chen Q. Roy KC. *Deepran: attention-based bilstm and crf for ransomware early detection and classification. Information Systems Frontiers*. 2021. URL: https://link.springer.com/article/10.1007/s10796-020-10017-4 (visitado 23-05-2025).

[60] Mareels I. Shafin SS Karmakar G. *Obfuscated memory malware detection in resource-constrained IoT devices for smart city applications*. 2023. URL: https://www.mdpi.com/1424-8220/23/11/5348 (visitado 23-05-2025).

[61] Roy K. Smith D Khorsandroo S. *Supervised feature selection to improve the accuracy for malware detection*. 2023. URL: https://www.researchsquare.com/article/rs-2898970/v1 (visitado 23-05-2025).

[62] Sophos. *El estado del ransomware 2024*. URL: https://www.sophos.com/es-es/content/state-of-ransomware.

[63] Carrier T. *Detecting obfuscated malware using memory feature engineering. Thesis, University of New Brunswick, Fredericton, NB, Canada*. 2021. (Visitado 23-05-2025).

[64] Assegie TA. *An optimized KNN model for signature-based malware detection. Tsehay Admassu Assegie. An optimized KNN model for signature-based malware detection. International Journal of Computer Engineering in Research Trends*. 2021. (Visitado 23-05-2025).

[65] *Teams Documentation*. URL: https://learn.microsoft.com/es-es/microsoftteams/ (visitado 23-05-2025).

[66] *Understanding Bidirectional LSTM for Sequential Data Processing*. URL: https://medium.com/@anishnama20/understanding-bidirectional-lstm-for-sequential-data-processing-b83d6283befc (visitado 23-05-2025).

[67] Wikipedia. *World Wide Web*. URL: https://es.wikipedia.org/wiki/World_Wide_Web.

[68] *Wikipedia. ¿Qué es el ransomware?* URL: https://es.wikipedia.org/wiki/Ransomware (visitado 23-05-2025).

[69] *XGBoost Documentation*. URL: https://xgboost.readthedocs.io/en/release_3.0.0/ (visitado 23-05-2025).

[70] Zhang Y, Liu Z y Jiang Y. *The classification and detection of malware us- ing soft relevance evaluation. IEEE Transactions on Reliability*. 2020. URL: https://ieeexplore.ieee.org/document/9218986 (visitado 23-05-2025).

[71] Wang X Yan J. *Unsupervised and semi-supervised learning: the next frontier in machine learning for plant systems biology. The Plant Journal*. 2022. URL: https://www.sciencedirect.com/science/article/abs/pii/S1568494620305688?via%3Dihub (visitado 23-05-2025).