

UNIVERSIDAD COMPLUTENSE DE MADRID
FACULTAD DE CIENCIAS MATEMÁTICAS

DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN



TRABAJO DE FIN DE GRADO

**Algoritmos de Aprendizaje Automático
aplicados a problemas de Ciberseguridad**

Presentado por: Pablo Jiménez Poyatos

Dirigido por: Luis Fernando Llana Diaz

Grado en Matemáticas

Curso académico 2023-24

Agradecimientos

Quiero expresar mi profunda gratitud hacia todas las personas que han sido indispensables en esta etapa. Sin su apoyo, guía y compañía, este camino no habría sido posible.

En primer lugar, dedico este trabajo a mis amigos de la carrera: María, Miguel, Nerea, Alex, Meléndez, Claudia, Tofi, Vicky, Luna, Celia, Paula, Pablo y Carlos. Gracias por compartir conmigo este viaje lleno de desafíos y logros. A pesar de las dificultades y las largas horas de estudio, vuestra presencia ha hecho que esta carrera sea mucho más llevadera y divertida. Sois parte esencial de esta etapa inolvidable.

También quiero reconocer a todos mis profesores, especialmente a Rafa, Óscar Martín, Natalia, María Isabel y Óscar Domínguez. Cada uno de vosotros ha dejado una marca especial en mí, convirtiendo este difícil camino en una aventura especial y enriquecedora. Agradezco sinceramente vuestro apoyo constante, pasión y ayuda durante estos años.

A Luis Llana, quien me ha guiado a lo largo de este año. Muchas gracias por la dedicación que has tenido conmigo durante todo el curso, con reuniones semanales, aunque alguna semana yo no tuviera mucho que decirte, han sido un impulso constante para mí. Muchas gracias también por tu apoyo, orientación y consejos han sido cruciales para la finalización de este trabajo.

No quiero olvidar a las dos personas que despertaron mi interés y pasión por las matemáticas, Carmen Zamarro y Eulalia. Vuestra energía, pasión e inspiración han sido fundamentales en mi trayectoria académica. Gracias por guiarme hacia este fascinante mundo.

A mis padres, gracias por vuestro amor y apoyo incondicional. Habéis estado ahí en todo momento, dándome la fuerza para seguir adelante incluso en los momentos más difíciles. Vuestra confianza en mí ha sido fundamental para alcanzar este logro. No hay palabras para expresar lo agradecido que estoy por todo lo que habéis hecho por mí.

Por último y más importante, a mi novia. Noe, eres mi motor y mi fuente de inspiración. Gracias por estar siempre a mi lado, por tu paciencia y por creer en mí. Tu apoyo ha sido esencial para superar todos los obstáculos y lograr este objetivo. Gracias por apoyarme y escucharme en todo momento, aunque a veces me ponga un poco pesado. Te agradezco de corazón por ser parte de mi vida y por todo lo que has hecho para ayudarme a llegar hasta aquí. Te quiero.

Gracias a todos por hacer posible este momento tan importante.

Resumen

Vivimos en un mundo cada vez más digital, donde la seguridad de la información se ha convertido en una prioridad. Este trabajo se enfoca en aplicar métodos de aprendizaje automático para intentar encontrar soluciones eficaces a problemas de ciberseguridad. Partiendo de un análisis exhaustivo sobre los fundamentos del aprendizaje profundo y sus diferentes arquitecturas, se estudia su uso en la clasificación y detección de intrusiones. Para el problema de clasificación de malware, se utiliza el *Microsoft Malware Classification Challenge*, aplicando redes neuronales convolucionales y autoencoders para evaluar su rendimiento. En la detección de intrusiones, se emplean los datos del *KDD Cup 1999*, implementando modelos de redes neuronales profundas, recurrentes, convolucionales y *autoencoders*. Finalmente, se discuten los resultados obtenidos en función de su precisión y capacidad de generalización de los datos, destacando el gran potencial de estas técnicas de aprendizaje automático para fortalecer la seguridad informática.

Palabras clave: Aprendizaje Automático, Ciberseguridad, Detección de Intrusiones, Clasificación de Malware, Redes Neuronales.

Abstract

We live in an increasingly digital world where information security has become a priority. This work focuses on applying machine learning methods to attempt to find effective solutions to cybersecurity problems. Starting from a comprehensive analysis of the fundamentals of deep learning and its different architectures, its use in classification and intrusion detection is studied. For the malware classification problem, the *Microsoft Malware Classification Challenge* is used, applying convolutional neural networks and autoencoders to evaluate their performance. For intrusion detection, data from the *KDD Cup 1999* is used, implementing deep, recurrent, convolutional neural network models and autoencoders. Finally, the results obtained are discussed in terms of their accuracy and data generalization capability, highlighting the great potential of these machine learning techniques to strengthen information security.

Keywords: Machine learning, Ciberseguridad, Intrusion Detection, Malware Classification, Neural Networks

Índice general

1. Introducción	1
1.1. Motivación y objetivos del trabajo	1
1.2. Contexto y antecedentes del trabajo	2
1.2.1. Definiciones y conceptos sobre ciberseguridad	3
1.2.2. Tecnología de redes y transmisión de datos	3
1.3. Metodología	5
1.4. Estructura de la memoria	5
1.5. Contribuciones	6
2. Fundamentos de las redes neuronales	7
2.1. Aprendizaje automático	7
2.2. Aprendizaje profundo	9
2.3. Optimización de los modelos de machine learning	9
2.3.1. Descenso de gradiente	10
2.3.2. Descenso de gradiente estocástico	11
2.3.3. Descenso de gradiente por mini-lotes	12
2.3.4. Propagación de la raíz media cuadrática	12
2.3.5. Estimación del Momento Adaptativo	13
2.4. Arquitecturas relevantes	13
2.4.1. Perceptrón	14
2.4.2. Perceptrón Multicapa	15
2.4.3. Autoencoder	16

ÍNDICE GENERAL

2.4.4. Red Neuronal Convolucional	17
2.4.5. Red Neuronal Recurrente	22
2.5. Función de perdida	26
2.5.1. Clasificación	26
2.5.2. Regresión	27
2.6. Función de activación.	28
2.7. Sobreajuste del modelo.	31
2.7.1. Regularización l1 y l2	32
2.7.2. Dropout	33
2.7.3. Parada temprana	34
2.8. Evaluación del modelo.	35
2.8.1. Matriz de Confusión	36
2.8.2. Métricas	36
2.8.3. Curva ROC y AUC	37
2.8.4. Curva de Precisión-Recall	38
2.9. Bibliotecas utilizadas en Python	39
2.9.1. Principales frameworks.	39
2.9.2. Librerías y herramientas esenciales.	40
3. Clasificación de Malware	43
3.1. Microsoft Malware Classification Challenge	43
3.1.1. Distribución del dataset	45
3.2. Red Neuronal Convolutinal	47
3.2.1. Visualizar el malware como imagen	47
3.2.2. Visualización del modelo	48
3.2.3. Aportaciones al modelo	49
3.3. Autoencoder	53
3.3.1. Autoencoder simple	54
3.3.2. Autoencoder profundo	55

ÍNDICE GENERAL

3.3.3. Autoencoder convolucional	57
3.4. Resultados	59
4. Detección de intrusiones	63
4.1. KDD Cup 1999 Data	63
4.2. Preparación datos	66
4.3. Red Neuronal Profunda	67
4.3.1. Arquitectura del Modelo	67
4.3.2. Evaluación del modelo	69
4.4. Red Neuronal Recurrente	70
4.4.1. Arquitectura del Modelo	70
4.4.2. Evaluación del modelo	71
4.5. Autoencoder	72
4.5.1. Arquitectura del modelo	72
4.5.2. Evaluación del modelo	73
4.6. Red Neuronal Convolutacional	74
4.6.1. Visualización de los datos	74
4.6.2. Arquitectura del Modelo	75
4.6.3. Evaluación del modelo	77
4.7. Resultados	78
5. Conclusiones y Trabajo Futuro	79
5.1. Conclusiones	79
5.2. Trabajo Futuro	80
Bibliografía	81

Capítulo 1

Introducción

Las redes neuronales representan uno de los métodos más prometedores en la resolución de problemas complejos en diversos campos. En el ámbito de la ciberseguridad, su aplicación ofrece nuevas vías para abordar desafíos como la detección de amenazas y la mitigación de ataques. En este primer capítulo, definiremos los conceptos clave de la ciberseguridad, explorando los diferentes tipos de malware y los principales desafíos a los que se enfrenta hoy en día. A continuación, describiremos los objetivos (tanto principales como secundarios) de esta investigación, incluyendo la motivación para la elección de este tema. Finalmente, presentaremos la estructura de esta memoria, destacando las contribuciones más importantes de nuestro trabajo. Las principales fuentes de consulta para la elaboración de este capítulo son: [77, 28, 74].

1.1. Motivación y objetivos del trabajo

La creciente digitalización de nuestro día a día y la expansión masiva de Internet han traído consigo numerosos beneficios, pero también han incrementado significativamente el riesgo de ciberataques. Empresas, gobiernos y particulares son cada vez más dependientes de la tecnología, lo que ha llevado a un aumento en la exposición a amenazas cibernéticas.

La sofisticación de los atacantes y la variedad de métodos utilizados hacen que la detección y prevención de intrusiones sea un desafío constante. En este contexto, la ciberseguridad no es solo una preocupación personal, sino también una cuestión de seguridad nacional y estabilidad económica. La capacidad para anticipar, identificar y mitigar estos ataques se ha convertido en un aspecto crucial para proteger tanto los datos sensibles como las infraestructuras críticas.

Desde un enfoque más personal, mi interés en la inteligencia artificial (IA) ha sido un factor determinante en la elección de este tema para mi Trabajo Fin de Grado. Aunque no tuve la oportunidad de cursar asignaturas específicas sobre IA durante mi formación, siempre he estado fascinado por su potencial. Una vez decidido ese tema, quería aplicarlo a un problema en particular. A raíz del ataque cibernético sufrido por Telepizza [15], encontré una gran oportunidad para explorar este campo, ya que mi cercanía a una persona que estuvo directamente involucrado en la respuesta a este incidente me permitió conocer de primera mano los desafíos y la importancia de una protección adecuada contra los ciberataques.

El presente trabajo tiene como objetivo principal explorar y desarrollar técnicas avanzadas basadas en redes neuronales artificiales en el ámbito de la seguridad digital. Para alcanzar estos objetivos generales, se han definido los siguientes objetivos específicos.

- Estudio teórico de las redes neuronales con sus diferentes arquitecturas e hiperparámetros.
- Presentación de dos problemas actuales de ciberseguridad, la clasificación de malware y la detección de intrusiones.
- Replicación de diferentes modelos de redes neuronales para resolver estos problemas.
- Implementación en Python de estos modelos para su entrenamiento y análisis de resultados.

1.2. Contexto y antecedentes del trabajo

En la era digital actual, la ciberseguridad se ha convertido en una preocupación esencial para empresas, ciudadanos y sociedades. Los ciberataques, que van desde virus y troyanos hasta el espionaje y la filtración de datos, son cada vez más comunes y sofisticados. Estos ataques no solo amenazan nuestra información personal y financiera, sino que también ponen en peligro grandes infraestructuras, lo que puede tener efectos en cadena a gran escala.

Debido al creciente número de sistemas conectados a Internet, el riesgo de ataques también a aumentado. Recientes ciberataques como el sufrido por el Banco Santander accediendo a su base de datos [104], o el ataque a la aplicación de gestión de prácticas de la Universidad Complutense de Madrid [42], resaltan cómo las amenazas están en constante evolución y la necesidad urgente de fortalecer nuestras defensas cibernéticas.

Para enfrentar este problema, la investigación en detección de intrusiones ha cobrado mayor importancia. Los sistemas de detección de intrusiones (IDS) son herramientas clave en la defensa cibernética, y se dividen principalmente en dos tipos: basados en firmas y basados en anomalías. Los IDS basados en firmas detectan ataques comparando el comportamiento observado, con una base de datos de amenazas conocidas, pero este método puede ser esquivado por ataques nuevos y desconocidos. Por otro lado, los IDS basados en anomalías identifican comportamientos inusuales que no se ajustan a los patrones normales, ofreciendo una protección más amplia contra ataques novedosos [75].

La inteligencia artificial (IA), y en particular las redes neuronales artificiales (ANN), se presentan como una solución prometedora para mejorar la detección de intrusiones. Las ANN pueden analizar grandes volúmenes de datos y detectar patrones complejos de ataque que los métodos tradicionales podrían pasar por alto.

Uno de los principales desafíos es la gran cantidad de datos en los conjuntos de datos de ciberseguridad actuales, lo que requiere algoritmos inteligentes, como los de aprendizaje automático, para extraer información valiosa. En el contexto de los IDS, esto implica manejar una gran cantidad de características para seleccionar el mejor enfoque y detectar posibles ataques. Además, hay otros desafíos como la optimización de hiperparámetros, el balanceo de datos y la necesidad de explicabilidad y equidad en los modelos. Este es un problema importante porque un alto número de características en un conjunto de datos puede llevar al sobreajuste del modelo, lo que resulta en un desempeño deficiente en los conjuntos de datos de validación.

Entre todas las técnicas de aprendizaje automático disponibles, esta investigación se centra en el estudio de modelos basados en redes neuronales.

1.2.1. Definiciones y conceptos sobre ciberseguridad

Veamos ahora algunos conceptos sobre ciberseguridad que necesitaremos más adelante:

La ciberseguridad se refiere a las prácticas y tecnologías diseñadas para proteger máquinas, servidores, dispositivos móviles, sistemas electrónicos, redes y datos contra accesos no autorizados, ataques, daños o cualquier tipo de amenaza maliciosa.

El malware, abreviatura de *malicious software* (software malicioso), es un software diseñado para dañar, infiltrarse o interrumpir el funcionamiento de sistemas informáticos y redes. El propósito del malware puede variar desde el robo de información confidencial hasta el control remoto de dispositivos infectados.

Veamos cuales son los principales tipos de malware y en que consiste cada uno según [33, 58]

- **Virus:** Programas dañinos que se adhieren a archivos legítimos y se propagan a través de redes y correos electrónicos, causando problemas operativos, pérdida de datos, y ralentizaciones del sistema.
- **Spyware (programa espía):** Software que recopila información sobre el usuario y sus actividades en línea sin su consentimiento, comprometiendo la seguridad y la privacidad.
- **Worm (Gusano):** Malware que se replica y se propaga a través de redes sin intervención humana o de un programa huésped, consumiendo ancho de banda y recursos del sistema.
- **Adware (programa publicitario):** Variedad de malware que muestra anuncios no deseados a los usuarios, típicamente como ventanas emergentes, degradando el rendimiento del sistema y recopilando datos sobre las actividades del usuario.
- **Backdoor (Puerta trasera):** Herramienta que permite que una entidad no autorizada tome el control completo del sistema de una víctima sin su consentimiento. Un troyano de puerta trasera siempre se presenta como una herramienta de software legítima esencialmente requerida por el usuario. Herramienta que permite el control remoto no autorizado de un sistema, facilitando actividades maliciosas como el robo de datos y la instalación de otros malware.
- **Trojan (Troyano):** Programa que se disfraza de software legítimo para realizar acciones maliciosas como robar datos o abrir puertas traseras para otros malware.
- **Trojan downloader (Descargador troyano):** Malware que descarga e instala otros programas maliciosos en el sistema infectado, deshabilitando herramientas de seguridad y transfiriendo información sin permiso.
- **Obfuscated malware (Malware ofuscado):** Malware cuyo código se ha modificado para dificultar su detección y análisis por parte de software de seguridad.

1.2.2. Tecnología de redes y transmisión de datos

Por otro lado, entender la tecnología de redes y la transmisión de datos es crucial en este mundo cada vez más digital, ya que es la base de la comunicación entre dispositivos. Veremos algunos conceptos relacionados.

El **HTTP** (*HyperText Transfer Protocol*) es el protocolo estándar utilizado para la transmisión de información en la World Wide Web. Facilita la comunicación entre los navegadores web y los servidores, permitiendo que los navegadores soliciten datos y reciban contenido como páginas web. Cada solicitud HTTP utiliza un método específico (como GET, POST, PUT, DELETE,

entre otros) que define la operación que el cliente desea realizar. Por otro lado, el **HTTPS** (*HyperText Transfer Protocol Secure*) es la versión segura de HTTP. Utiliza el cifrado SSL/TLS para proteger la integridad y la confidencialidad de los datos transmitidos entre el navegador y el servidor.

Una **LAN** (*Local Area Network*) es una red de área local que conecta dispositivos dentro de un área limitada, como una oficina o un edificio. Las LANs son rápidas y permiten la comunicación eficiente y la compartición de recursos como archivos e impresoras.

El **IP** (*Internet Protocol*) es el conjunto de reglas que define cómo se envían y reciben los datos a través de la red. Existen dos versiones principales de este protocolo: *IPv4* e *IPv6*. *IPv4* es la versión más utilizada históricamente y tiene un rango de direcciones de 4 bytes (32 bits), permitiendo un total de 2^{32} direcciones IP posibles. Debido al limitado número de direcciones, se han implementado soluciones como la creación de subredes y el uso de tecnologías como **NAT** (Traducción de Direcciones de Red), donde una única dirección IP puede ser compartida por múltiples dispositivos. Por otro lado, *IPv6*, con direcciones de 128 bits, ofrece un rango de direcciones significativamente más amplio, permitiendo asignar una dirección IP única a cada dispositivo, solucionando las limitaciones de escalabilidad presentes en *IPv4*.

El **TCP** (*Transmission Control Protocol*) es un protocolo de control de transmisión que garantiza la entrega fiable de datos entre dispositivos en una red. TCP segmenta los datos en paquetes, los envía y asegura que lleguen al destino en el orden correcto y sin errores. Por otro lado, se encuentra el protocolo **UDP** (*User Datagram Protocol*), que no da garantías de que el mensaje ha llegado. Por último, vamos a ver el protocolo **ICMP** (*Internet Control Message Protocol*), que complementa a TCP y UDP proporcionando mecanismos de diagnóstico y control en redes IP, tales como la comunicación de errores y la verificación de conectividad a través de mensajes informativos, siendo fundamental para la gestión y el mantenimiento de la comunicación en Internet.

El **DNS** (*Domain Name System*) traduce nombres de dominio legibles para humanos en direcciones IP que las computadoras utilizan para identificar servidores en la red. Es una parte clave de la navegación en Internet.

Un **router** es un dispositivo de red que dirige el tráfico de datos entre diferentes redes. Determina la mejor ruta para cada paquete de datos y lo envía al siguiente nodo en esa ruta.

Un **firewall** es una barrera de seguridad que controla el tráfico de red permitido o denegado basado en reglas de seguridad predefinidas. Protege las redes contra accesos no autorizados y ataques cibernéticos.

Finalmente, un **proxy** actúa como intermediario entre un usuario y la Internet. Puede filtrar solicitudes, mejorar la seguridad y almacenar datos en caché para mejorar la eficiencia de la red.

La comprensión de estos elementos no solo es vital para la configuración y mantenimiento de redes seguras, sino que también sienta las bases para el estudio de mecanismos de detección y prevención de intrusiones.

1.3. Metodología

Para lograr los objetivos establecidos en este estudio, se comenzó con un profundo análisis teórico y práctico del funcionamiento de las redes neuronales siguiendo los libros [28, 74] e incluyendo los apuntes de la asignatura de Geometría Computacional del profesor Robert Monjo [66]. Paralelamente, se implementaron varios modelos utilizando tutoriales de Kaggle¹ como guía, tales como la detección de señales de tráfico y la clasificación de dígitos en el conjunto de datos MNIST mediante redes neuronales convolucionales (CNN) [2, 29]. Estas implementaciones proporcionaron una base sólida para comprender las arquitecturas y técnicas de entrenamiento de redes neuronales aplicadas a los problemas específicos.

Posteriormente, se llevó a cabo un estudio sobre el uso de redes neuronales en ciberseguridad. Se consultó el *review* [77] que ofreció una visión general de cómo estas redes se emplean para abordar diversos problemas en este campo. A partir de esta revisión, se identificaron dos problemas principales en ciberseguridad: la detección de intrusiones y la clasificación de ataques.

Con base en los problemas identificados, se procedió a seleccionar las bases de datos más apropiadas que proporcionaran datos relevantes y variados para cada uno de estos problemas. Esta selección se basó principalmente en escoger las bases de datos con más métodos de aprendizaje profundo documentados en artículos.

Para el desarrollo de los modelos de redes neuronales en Python, se implementaron diversas mejoras y adaptaciones específicas a cada problema. Esto incluyó ajustes en las arquitecturas de red, la optimización de hiperparámetros y la selección de técnicas de procesamiento de datos adecuadas. Cada modelo fue sometido a pruebas para medir su rendimiento y asegurar su eficacia en la detección y clasificación de intrusiones y ataques.

Finalmente, se procedió a la evaluación comparativa de los modelos desarrollados utilizando diferentes métricas de rendimiento. El modelo que demostró ser más efectivo en términos de precisión y capacidad para generalizar fue seleccionado como la solución óptima para cada uno de los problemas estudiados.

1.4. Estructura de la memoria

La presente memoria se estructura en cinco capítulos, cada uno de los cuales aborda diferentes aspectos del estudio y aplicación de algoritmos de aprendizaje automático en problemas de ciberseguridad.

El **Capítulo 2**, denominado *Fundamentos de las redes neuronales*, ofrece una introducción detallada a los conceptos fundamentales del aprendizaje automático y el aprendizaje profundo. Se explican diversas técnicas de optimización de modelos de *machine learning*, incluyendo el descenso de gradiente y sus variantes, así como diferentes arquitecturas de redes neuronales, tales como el perceptrón, los *autoencoders*, las redes neuronales convolucionales y las redes neuronales recurrentes. También se discuten las funciones de pérdida y activación, el sobreajuste del modelo y algunas métricas de evaluación para los diferentes modelos.

En los **capítulos 3 y 4**, titulados *Clasificación de Malware* y *Detección de intrusiones* respectivamente, se describen dos de los principales desafíos de la ciberseguridad. Para ello, primero se va a explicar la base de datos utilizada, para después aplicar diferentes modelos de redes neuro-

¹<https://www.kaggle.com/>

nales y estudiar cual de ellos se ajusta mejor al problema. Por último, se presentan y analizan los resultados obtenidos, destacando los modelos más efectivos y sus aportaciones al problema.

Finalmente, el **Capítulo 5**, titulado *Conclusiones y Trabajo Futuro*, resume las conclusiones obtenidas a partir de los experimentos y análisis realizados en los capítulos anteriores. Se destacan las principales contribuciones del trabajo y se sugieren posibles direcciones para investigaciones futuras, con el objetivo de mejorar y ampliar los enfoques presentados en esta memoria.

Todos los experimentos y códigos de Python utilizados se encuentran en [78].

El entorno de hardware en el que he realizado todos los experimentos es un servidor proporcionado por la Facultad de Informática de la Universidad Complutense de Madrid llamado Simba. Tiene un sistema operativo Debian 12.2 con Linux version 6.1.0-17-*amd64* con memoria RAM disponible de 128 GB. La CPU utilizada es un Intel(R) Xeon(R) W-2235 CPU con 3.8 GHz con 6 núcleos.

1.5. Contribuciones

Aunque mi objetivo inicial era replicar modelos existentes de redes neuronales, a lo largo del trabajo he decidido hacer ciertas modificaciones y aportaciones a diferentes modelos para lograr mejores resultados. En otros casos, he decidido realizar diferentes experimentos para comprobar la efectividad de diferentes hiperparámetros.

En particular, en el Capítulo 3, he realizado algunas contribuciones en el campo de la clasificación de malware. Para los modelos de CNN, llevé a cabo diferentes experimentos con el fin de verificar que los hiperparámetros indicados en el artículo eran efectivamente los mejores. Además, implementé mejoras adicionales para optimizar la generalización del modelo y evitar el sobreajuste.

En el capítulo 3 pero en los modelos basados en *autoencoders*, mis principales contribuciones consistieron en que no utilicé ningún artículo específico que hubiera resuelto este problema con anterioridad. En su lugar, investigué diversas arquitecturas y realicé diferentes experimentos para determinar cuál de todas ellas ofrecía el mejor rendimiento para la clasificación de malware.

Por último, en el Capítulo 4 decidí añadir métodos específicos para evitar el sobreajuste en los modelos desarrollados. Esta decisión fue crucial para mejorar la robustez y la capacidad de generalización de los modelos de redes neuronales aplicados a la detección de intrusiones.

Capítulo 2

Fundamentos de las redes neuronales

La Inteligencia Artificial (IA) es un campo que busca desarrollar sistemas capaces de realizar tareas que normalmente requieren inteligencia humana. Desde su conceptualización en la conferencia de Dartmouth en 1956, donde se propuso la idea de que “cada aspecto del aprendizaje o cualquier otra característica de la inteligencia podría, en principio, ser descrito tan precisamente que una máquina podría simularlo” [67], la IA ha evolucionado significativamente y ha encontrado aplicaciones en una amplia gama de sectores. La inteligencia artificial permite que los sistemas informáticos extraigan conclusiones a partir de datos, creando cada vez algoritmos más complejos para obtener mejores resultados. Aunque la cultura popular a menudo representa la IA de manera exagerada, como en la película de *Terminator*, estamos lejos de alcanzar ese nivel de sofisticación. Sin embargo, con los avances en *chatbots* que pueden mantener conversaciones más naturales, estamos superando algunas limitaciones. A pesar de esto, estos sistemas todavía dependen de datos preexistentes y no pueden pensar de manera independiente.

2.1. Aprendizaje automático

Dentro del ámbito de la IA, el aprendizaje automático (*Machine Learning*, ML) se destaca como una de sus técnicas fundamentales. A diferencia de los enfoques tradicionales de programación, donde las instrucciones son codificadas por los programadores, el ML es la ciencia o el arte de programar ordenadores para que puedan aprender a partir de datos [52].

Arthur Samuel lo definió en 1959 como “el campo de estudio que otorga a las computadoras la capacidad de aprender sin ser explícitamente programadas”[28]. Más formalmente, según Tom Mitchell (1997), “se dice que un programa de computadora aprende de la experiencia E con respecto a alguna tarea T y alguna medida de rendimiento P , si su rendimiento en T , medido por P , mejora con la experiencia E ” [28]. El aprendizaje automático ha revolucionado numerosos campos, permitiendo a las máquinas realizar tareas que antes requerían intervención humana directa. Desde la conducción autónoma hasta el diagnóstico médico, las aplicaciones del aprendizaje automático son diversas. A diferencia de los métodos tradicionales de programación,

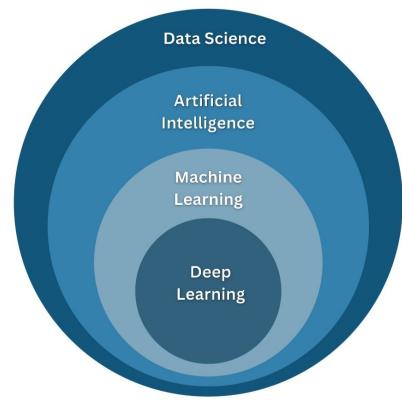


Figura 2.1: Relación entre Ciencia de Datos, Inteligencia Artificial, *Machine Learning* y *Deep Learning*. Fuente [57]

el aprendizaje automático permite que los sistemas descubran patrones y relaciones directamente a partir de los datos, adaptándose y mejorando con el tiempo. Los modelos se entrenan utilizando conjuntos de datos y, a partir de este entrenamiento, pueden predecir las salidas para nuevas entradas.

Un **ejemplo** de aprendizaje automático es un filtro de *spam* que, dándole ejemplos de correos *spam* y ejemplos de correos electrónicos normales (llamados *ham*), puede aprender a marcar el *spam* [28]. Los ejemplos que el sistema utiliza para aprender se llaman el conjunto de entrenamiento. Cada ejemplo de entrenamiento se llama una instancia de entrenamiento (o muestra). En este caso, la tarea T es marcar el *spam* en los nuevos correos electrónicos, la experiencia E son los datos de entrenamiento, y la medida de rendimiento P podría ser la precisión del filtro. Un filtro de *spam* utilizando técnicas tradicionales de programación, en primer lugar, consideraría cómo se ve normalmente el *spam*, detectando palabras comunes u otros patrones como el nombre del remitente y escribiendo reglas para cada una de estas. Pero si los encargados de mandar el *spam* detectan que todos los correos que incluyen la palabra “para usted” o “cuenta bancaria” son rechazados, pueden modificar estas palabras por otras y así ser aceptados por el filtro. Luego un filtro de *spam* que utiliza técnicas tradicionales de programación necesitaría ser actualizado continuamente para detectar correos electrónicos *spam*. Por otro lado, un filtro de *spam* basado en técnicas de aprendizaje automático nota automáticamente que “para ti”, se ha vuelto inusualmente frecuente en el *spam* marcado por los usuarios, y comienza a marcarlos sin intervención humana [28].

El **esquema global de aprendizaje** consta de tres módulos principales: el generador, el entrenamiento y la decisión. El generador proporciona entradas estructuradas, principalmente vectores con los datos para su procesamiento. El entrenamiento ajusta los parámetros del modelo basándose en las salidas deseadas, y por último, la decisión asigna categorías a nuevas muestras de entrada utilizando los parámetros aprendidos durante el entrenamiento [74].

En cuanto a la **clasificación de los sistemas de aprendizaje automático**, se distinguen cuatro tipos principales: el aprendizaje supervisado, aprendizaje no supervisado, aprendizaje semisupervisado y aprendizaje por refuerzo. Toda esta información proporcionada a continuación ha sido extraída de [28].

1. En el aprendizaje **supervisado**, todos los datos de entrenamiento tienen asociados las salidas deseadas (también llamadas etiquetas). Todo dato de entrada x , conoce su salida y , de tal forma que existe una función f desconocida que relaciona ambas, $y = f(x)$. El objetivo del aprendizaje supervisado consiste en estimar la función \hat{f} de tal forma que \hat{f} sea una buena aproximación de la función f que relaciona las entradas x con las salidas Y . Un ejemplo puede ser el del filtro de *spam* ya que se entrena el modelo con los correos y con la etiqueta de si son *spam* o no. Otros ejemplos incluyen regresión lineal, regresión logística, árboles de decisión y redes neuronales.
2. En el aprendizaje **no supervisado**, al contrario que en el supervisado, los datos de entrenamiento no están etiquetados. De esta forma, el modelo tiene que aprender sin supervisión (también llamado sin “profesor”). El objetivo principal de este tipo de algoritmos se basa en el agrupamiento, detección de anomalías o reducción de dimensionalidad. Un ejemplo podría ser el clustering, que se usa principalmente cuando queremos clasificar una muestra de datos según su similitud.
3. En el caso de encontrarnos ante un problema en el que tengamos datos tanto etiquetados como sin etiquetar, nos encontramos ante un problema de **aprendizaje semisupervisado**. Este tipo de aprendizaje suele darse en situaciones en las que obtener etiquetas de los datos puede ser muy costoso pero sin embargo obtener datos sin etiquetar no tanto. Un

ejemplo podría ser la agrupación de fotos donde sale la misma persona en Google Photos. La parte no supervisada sería la de agrupación y la supervisada la de dar una etiqueta a cada grupo.

4. Por último, está el **aprendizaje por refuerzo**, un tipo de aprendizaje un poco diferente a los otros tres. Es un método en el cual un agente (como un robot o un programa) interactúa con su entorno y toma decisiones para recibir recompensas o evitar penalizaciones. El agente aprende a través de la experiencia que obtiene al observar qué acciones le proporcionan las mejores recompensas a lo largo del tiempo. Este proceso implica desarrollar una estrategia, llamada política, que le indica qué acción tomar en cada situación para maximizar sus recompensas futuras. Por ejemplo, un robot que usa aprendizaje por refuerzo podría aprender a caminar ajustando sus movimientos según las recompensas y penalizaciones que recibe por sus acciones, mejorando gradualmente su habilidad para andar.

2.2. Aprendizaje profundo

Dentro del *Machine Learning*, se encuentra el *Deep Learning*, cuya base son las redes neuronales artificiales (ANN). Las ANN son unas estructuras versátiles, potentes y escalables, lo que las hace ideales para abordar tareas grandes y complicadas para el aprendizaje profundo.

Según [74], se definen como un modelo matemático inspirado en la estructura de una red neuronal biológica. Consiste en una red de neuronas interconectadas organizadas por capas, con una capa de entrada, una o más capas ocultas y una capa de salida [22]. En cada neurona se aplica una suma ponderada de las señales recibidas a las que se le aplica una función de activación o conexión no lineal.

La capa de entrada recibe la información del exterior y la agrupa en la capa de entrada, mandando una salida a la siguiente capa a través de sus neuronas con **pesos** asociados en cada conexión. Las capas ocultas reciben información de otras neuronas artificiales y cuyas señales de entrada y salida permanecen dentro de la red. Por último, la capa de salida recibe la información procesada y la devuelve al exterior con la salida predicha por nuestro modelo. Además de los pesos que se van ajustando durante el entrenamiento, también está el **sesgo o bias**, que es un valor que se asigna a cada neurona de cada capa para añadir características adicionales a la red neuronal que antes no tenía.

Para optimizar el rendimiento de estas redes, es esencial entender y aplicar conceptos de optimización.

2.3. Optimización de los modelos de machine learning

Sea $f : S \subseteq \mathbb{R}^n \rightarrow \mathbb{R}^m$, una función donde n y m son números enteros y sea $w \in \mathbb{R}^n$. La optimización de $f(w)$ consiste en minimizar o maximizar¹ su valor variando w . La función f que se pretende optimizar se conoce como función objetivo, función de pérdida o función de error.

En Cálculo Diferencial, aprendemos que la derivada de una función en un punto w nos da la inclinación o pendiente de la función en ese punto. Si aplicamos un pequeño desplazamiento

¹En general, optimización hace referencia a una minimización, ya que la maximizar $f(w)$ es equivalente a minimizar $-f(w)$.

$\delta \in \mathbb{R}^n$ en la entrada w , podemos aproximar el valor de la función en este nuevo punto $w + \delta$ mediante la siguiente relación:

$$f(w + \delta) \approx f(w) + f'(w) \cdot \delta \quad (2.1)$$

Para poder utilizar esta aproximación, podemos observar que es necesario que la función sea diferenciable. Si se utilizan funciones que no son diferenciables en todos los puntos, entonces se debe aplicar una aproximación de la derivada para poder utilizar la función en el método de descenso de gradiente.

En el caso particular de las redes neuronales, la función de pérdida calcula la discrepancia entre la salida de nuestra red neuronal (predicción) y la etiqueta verdadera. Se va a presentar ahora un método de optimización conocido como método de descenso de gradiente (*gradient descent*). Durante este método, se propaga el error hacia atrás a través de la estructura del modelo, de forma que los pesos en las redes neuronales se vayan ajustando a lo largo del entrenamiento. Este proceso cierra el ciclo de aprendizaje entre el envío de los datos hacia adelante, la generación de predicciones y la mejora durante la propagación hacia atrás. Al adaptar los pesos, el modelo probablemente mejore (a veces mucho, a veces ligeramente) y, por lo tanto, se dice que se ha realizado el aprendizaje [74].

2.3.1. Descenso de gradiente

Sea $f : S \subseteq \mathbb{R}^n \rightarrow \mathbb{R}^m$, donde n y m son enteros y sea $w \in \mathbb{R}^n$. Cuando $f'(w) = 0$, la derivada no proporciona información sobre la dirección en la que hay que moverse. Estos puntos se conocen como puntos críticos o puntos estacionarios.

Definición 2.1. Un punto $w_0 \in \mathbb{R}^n$ es un **mínimo local** si $\exists \epsilon \in \mathbb{R}$ tal que $f(w_0) \leq f(w)$ para todo $w \in B(w_0, \epsilon)$ ².

Definición 2.2. Un punto $w_0 \in \mathbb{R}^n$ es un **punto de inflexión** de f si es continua en w_0 y existe un entorno $B(w_0, \epsilon)$ tal que f'' cambia de signo.

Definición 2.3. Un punto w_0 es un **mínimo global** de f si para todo $w \in S$, $f(w_0) \leq f(w)$.

Una función puede tener solo un mínimo global o múltiples mínimos globales de la función. También es posible que haya mínimos locales que no sean globalmente óptimos.

Si queremos minimizar una función que tiene varias entradas pero solo una salida, ($f : \mathbb{R}^n \rightarrow \mathbb{R}$), usamos el concepto de derivada parcial. Recordemos que si $w = (w_1, w_2, \dots, w_n) \in \mathbb{R}^n$, entonces $\frac{\partial f}{\partial w_i}(w)$ nos dice cómo cambia el valor de f cuando w_i varía.

Definición 2.4. La derivada direccional nos dice cómo cambia la función f en una dirección específica, indicada por un vector unitario $u \in \mathbb{R}^n$. Para encontrar la derivada direccional en la dirección u , consideraremos cómo cambia la función f en el punto $w + \alpha u$ cuando α varía. Usando la regla de la cadena, podemos ver que esta derivada direccional es igual a:

$$f'(w + \alpha u) = u^t \cdot \nabla f(w) \quad (2.2)$$

Esto nos proporciona una forma de evaluar cómo cambiará la función en cualquier dirección específica desde el punto w [74].

En el contexto del aprendizaje automático, se optimizan funciones que pueden tener muchos mínimos locales que no son óptimos y muchos puntos de inflexión rodeados de regiones muy

² $B(a, b)$ hace referencia a la bola o disco de centro a y radio b

planas. Todo esto hace que la optimización sea difícil, especialmente cuando la entrada a la función es multidimensional. Por lo tanto, se suele conformar con encontrar un valor que sea muy bajo pero no necesariamente mínimo en ningún sentido formal [74].

Sea $u \in \mathbb{R}^n$ una dirección y sean $w \in \mathbb{R}^n$ los pesos de la red neuronal. Para minimizar f , es deseable encontrar la dirección $u \in \mathbb{R}^n$ en la que f disminuye de forma más rápida. Esto se puede conseguir minimizando la derivada direccional:

$$\min_{u \in \mathbb{R}^n} f'(w + \alpha u) = \min_{u \in \mathbb{R}^n} u^t \cdot \nabla f(w) = \min_{u \in \mathbb{R}^n} \|u\| \cdot \|\nabla f(w)\| \cos(\theta) \quad (2.3)$$

donde θ es el ángulo entre u y el gradiente del vector w . Asumiendo la normalidad en el vector u e ignorando factores que no dependen de u , se concluye que hay que minimizar:

$$\min_{u \in \mathbb{R}^n} \cos(\theta) \quad (2.4)$$

Esto nos sugiere que para minimizar el valor de $f(w)$, hay que moverse en dirección opuesta al gradiente. Este método se conoce como método de descenso de gradiente (*steepest descent* o *gradient descent*). El nuevo valor que propone este método para obtener un valor menor en la función de coste es:

$$w' = w - \eta \nabla f(w), \quad (2.5)$$

donde $\eta \in \mathbb{R}$ es la razón de aprendizaje (o *learning rate*), que determina el tamaño del paso. En caso de aplicar esta ecuación a cada uno de los datos de entrada, se llamará **descenso de gradiente por lotes**.

Según [74], el *learning rate* se puede elegir de varias formas. Una de ellas es fijarlo a un valor constante pequeño. Otra forma es determinando la dimensión del paso que hace desaparecer la derivada direccional. Por último, otra aproximación consiste en evaluar $f(\mathbf{w} - \eta \nabla f(\mathbf{w}))$ para varios valores de η y escoger el que proporciona el valor de la función objetivo menor. Escoger valores de η muy pequeños hace que la convergencia sea muy lenta. Sin embargo, valores muy grandes pueden hacer que el algoritmo diverja y se aleje en cada paso más del valor óptimo.

Este método converge cuando cada elemento del gradiente es cero o muy próximo a él.

2.3.2. Descenso de gradiente estocástico

El Descenso de Gradiente Estocástico (SGD, por sus siglas en inglés) es otro método utilizado en aprendizaje automático para minimizar una función objetivo. El funcionamiento del SGD consiste en seleccionar aleatoriamente un elemento del conjunto de entrenamiento en cada iteración y calcular los gradientes basándose únicamente en esa instancia individual [74]. Esto hace que se pueda entrenar una muestra de datos más grandes y que el algoritmo sea mucho más eficiente que calculándolo para todo el conjunto de datos.

Sin embargo, debido a su naturaleza aleatoria, el SGD es mucho menos regular que el Descenso de Gradiente por Lote. En lugar de disminuir suavemente hasta alcanzar el mínimo, la función de perdida oscilará, disminuyendo solo en promedio. Con el tiempo, el algoritmo se acercará mucho al mínimo, pero continuará oscilando alrededor de él sin asentarse completamente [28]. Una solución a este problema es reducir gradualmente la tasa de aprendizaje. Al principio, los pasos son grandes, lo que ayuda a hacer un progreso rápido y a escapar de mínimos locales. Luego, los pasos se hacen más pequeños, permitiendo que el algoritmo se asiente en el mínimo global [28].

Para aplicar este algoritmo, primero se elige un vector inicial de parámetros $w \in \mathbb{R}^n$ (que puede ser seleccionado aleatoriamente) y una razón de aprendizaje $\eta \in \mathbb{R}$. A continuación, se selecciona aleatoriamente una instancia del conjunto de entrenamiento y se actualizan los pesos según la fórmula 2.5. Después, con estos nuevos pesos, se vuelve a seleccionar otra nueva instancia y se vuelven a actualizar los pesos. Estos pasos se repiten iterativamente según el número de veces que se haya fijado [74].

También está la posibilidad de añadirle *momentum* al SGD, que es un hiperparámetro positivo que acelera el descenso del gradiente en la dirección relevante.

2.3.3. Descenso de gradiente por mini-lotes

Por último, se verá el método de descenso de gradiente por mini-lote (*Mini-batch Gradient Descent*). Es un algoritmo que combina aspectos del Descenso de Gradiente por Lote y el SGD. El descenso de gradiente por mini-lote calcula los gradientes basándose en pequeños subconjuntos aleatorios de instancias llamados mini-lotes (*mini-batch*). Estos subconjuntos definen el número de muestras utilizadas antes de actualizar los pesos de la red en el modelo.

Este enfoque logra un progreso menos errático que el SGD, especialmente con mini-lotes relativamente grandes, y se acerca más al mínimo, aunque puede ser más difícil escapar de mínimos locales. No obstante, con un buen plan de aprendizaje, tanto el SGD como el Descenso de Gradiente por Mini-lote pueden llegar al mínimo [28].

Otro parámetro relevante es la **época** (*epoch*), que define cuántas veces el conjunto total de muestras es procesado por el algoritmo. Cada muestra del conjunto de entrenamiento tiene una oportunidad de actualizar los pesos en cada época. El número de épocas varía según el tipo de datos y la arquitectura de red neuronal que se esté usando [74].

Resumimos los tres tipos de descenso de gradiente usando la definición de mini-lotes:

1. Si se utilizan todas las muestras de entrenamiento para crear un lote, se denomina **descenso del gradiente por lotes**.
2. Si el lote tiene el tamaño de una muestra, se llama **descenso de gradiente estocástico**.
3. Cuando el lote tiene más de una muestra pero es menor que el tamaño del conjunto de datos de entrenamiento, se denomina **descenso de gradiente de mini-lote**.

2.3.4. Propagación de la raíz media cuadrática

Además de estos métodos, existen optimizadores avanzados que mejoran la eficiencia del entrenamiento. Entre ellos se encuentra el algoritmo RMSProp (*Root Mean Square Propagation*). Es un método de optimización que ajusta la tasa de aprendizaje para cada parámetro manteniendo una media móvil de los cuadrados de los gradientes. Este método se basa en la observación de que la magnitud de los gradientes puede variar significativamente, lo que puede dificultar la convergencia del algoritmo de optimización. RMSProp aborda este problema adaptando dinámicamente la tasa de aprendizaje [74].

Sea v_t la media móvil de los cuadrados del gradiente, $\beta \in \mathbb{R}^+$ el factor de decaimiento de la media, $\nabla f(w_t) \in \mathbb{R}^n$ el gradiente de los pesos en la iteración t , $\eta \in \mathbb{R}^n$ la tasa de aprendizaje,

$\epsilon > 0$ la constante de ajuste que evita divisiones por cero y w_t el valor de los pesos en la iteración t . Entonces, las ecuaciones del algoritmo se definen como:

$$v_t = \beta \cdot v_{t-1} + (1 - \beta)(\nabla f(w_t))^2 \quad (2.6)$$

$$w_t = w_{t-1} - \frac{\eta}{\sqrt{v_t + \epsilon}}(\nabla f(w_t)) \quad (2.7)$$

donde w_t es la actualización de los pesos en la iteración t . Se inicializa este método con el valor de las constantes $v_0 = 0$ y $w_0 = 0$. Además, $\sqrt{v_t + \epsilon}$ es siempre distinto de cero ya que v_t es siempre mayor o igual que cero y ϵ es mayor que cero.

2.3.5. Estimación del Momento Adaptativo

Por último se verá el algoritmo Adam (*Adaptive Moment Estimation*), que combina las ideas de RMSProp y momentum, manteniendo medias móviles de los gradientes y de los cuadrados de los gradientes. Esto permite una optimización más estable y eficiente [74].

Sea $m_t \in \mathbb{R}^n$ la media móvil de los gradientes en la iteración t , $v_t \in \mathbb{R}^n$ la media móvil de los cuadrados de los gradientes en la iteración t , $\beta_1^t \in [0, 1]$ el factor de decaimiento de la media para los gradientes en el instante t , $\beta_2^t \in [0, 1]$ el factor de decaimiento de la media para los cuadrados de los gradientes en el instante t , $\hat{m}_t \in \mathbb{R}^n$ la media móvil de los gradientes corregida por el sesgo en la iteración t , $\hat{v}_t \in \mathbb{R}^n$ la media móvil de los cuadrados de los gradientes corregida por el sesgo en la iteración t , $\eta \in \mathbb{R}$ la tasa de aprendizaje, $\epsilon \in \mathbb{R}$ la constante de ajuste para evitar divisiones por cero y $w_t \in \mathbb{R}^n$ el valor de los pesos en la iteración t .

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot \nabla f(w_t), \quad (2.8)$$

$$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2)(\nabla f(w_t))^2, \quad (2.9)$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad (2.10)$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}, \quad (2.11)$$

$$w_t = w_{t-1} - \frac{\eta \cdot \hat{m}_t}{\sqrt{\hat{v}_t + \epsilon}}, \quad (2.12)$$

donde w_t es la actualización de los pesos en la iteración t . Los denominadores no se anulan ya que $\beta_1^t, \beta_2^t \in [0, 1]$ y v_t es siempre mayor o igual que cero y ϵ es mayor que cero.

2.4. Arquitecturas relevantes

En esta sección vamos a hacer un estudio de las arquitecturas de redes neuronales más utilizadas en ciberseguridad. Para ello, primero vamos a estudiar la arquitectura más básica, el perceptrón simple.

2.4.1. Perceptrón

Antes de profundizar en los modelos de redes neuronales más complejos y profundos, veamos el funcionamiento del modelo de ANN más simple, el perceptrón. Este modelo es la base del resto de modelos de aprendizaje automático. Consiste en una capa de entrada y en una de salida en la que hay que aplicar dos etapas. En la figura 2.2 podemos ver el esquema para dos posibles salidas.

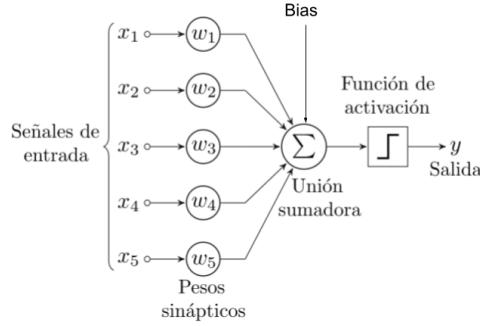


Figura 2.2: Modelo del perceptrón simple. Adaptación de [92]

La primera etapa del proceso consiste en calcular la suma promediada de sus entrada mediante una función lineal

$$f(x) = \sum_{i=1}^n w_i \cdot x_i + b \quad (2.13)$$

Los coeficientes $w_i \in \mathbb{R}$, $i = 1, 2, \dots, n$ se llaman pesos y dan un valor determinado a cada una de las entradas en función de la importancia para obtener la salida. Además, el coeficiente b es el sesgo o bias que se añade a la función. Otra forma práctica de escribir esta ecuación sería:

$$f(x) = \sum_{i=1}^n (w_i \cdot x_i) + b = w^t \cdot x + b \quad (2.14)$$

donde $w^t = (w_1, \dots, w_n)$ y $x^t = (x_1, \dots, x_n)$

La segunda capa consiste en transformar la salida de la primera etapa mediante una función de activación $\phi : \mathbb{R}^n \rightarrow \mathbb{R}$. Se podría generalizar de la siguiente manera:

$$y = \phi(w^t \cdot x + b) \quad (2.15)$$

Pero, ¿cómo ajusta los parámetros el perceptrón? El algoritmo de entrenamiento del perceptrón se basa en la regla de aprendizaje de Hebb[14] y tiene en cuenta el error cometido por la red al hacer una predicción. El perceptrón se alimenta con una entrada de entrenamiento a la vez, y para cada una, realiza sus predicciones. Para cada neurona de salida que produce una predicción errónea, se refuerzan los pesos de conexión que habrían contribuido a la predicción fallida [28]. La regla se muestra en la Ecuación 2.16.

$$w_{i,j}^{(t+1)} = w_{i,j}^{(t)} + \eta(y_j - \hat{y}_j)x_i \quad (2.16)$$

En esta ecuación, $w_{i,j}^{(t+1)}$ es el peso de conexión actualizado entre la i -ésima neurona de entrada y la j -ésima neurona de salida, $w_{i,j}^{(t)}$ es el peso de conexión actual entre la i -ésima neurona de entrada y la j -ésima neurona de salida, x_i es el valor de la i -ésima entrada de la instancia de

entrenamiento actual, \hat{y}_j es la salida de la j -ésima neurona de salida, y_j es la salida objetivo de la j -ésima neurona de salida y η es la tasa de aprendizaje [28].

El límite de decisión de cada neurona de salida es lineal, por lo que los perceptrones son incapaces de aprender patrones complejos. Esto significa que solo pueden separar los datos si hay una línea recta (o un hiperplano en dimensiones superiores) que divida las distintas clases de datos. Sin embargo, si las instancias de entrenamiento son linealmente separables, Rosenblatt demostró que este algoritmo convergería a una solución, es decir, encontraría un conjunto de pesos que clasifique correctamente todas las instancias de entrenamiento [28]. Esto se conoce como el Teorema de Convergencia del Perceptrón.

2.4.2. Perceptrón Multicapa

Un Perceptrón Multicapa (MLP) está compuesto por una capa de entrada, una o más capas ocultas, y una capa final llamada capa de salida. Todas las capas, excepto la capa de salida, incluyen una neurona de sesgo y están completamente conectadas a la siguiente capa [28].

Cuando una red neuronal artificial (ANN) contiene un conjunto profundo de capas ocultas, se llama red neuronal profunda (DNN). El campo del Aprendizaje Profundo estudia las DNNs, y más generalmente modelos que contienen conjuntos profundos de capas. Sin embargo, muchas personas hablan de Aprendizaje Profundo siempre que están involucradas redes neuronales.

David Rumelhart, Geoffrey Hinton y Ronald Williams publicaron un artículo revolucionario que introdujo el algoritmo de entrenamiento de retropropagación [79], que todavía se usa hoy en día. Consiste en un descenso de gradiente que utiliza una técnica eficiente para calcular los gradientes automáticamente. En solo dos etapas (una hacia adelante, una hacia atrás), el algoritmo puede determinar cómo se deben ajustar cada peso de conexión y cada término de sesgo para reducir el error.

El algoritmo maneja un mini-lote a la vez (*batch size*), y pasa por el conjunto completo de entrenamiento múltiples veces. Cada pasada se llama época. Cada mini-lote se pasa a la capa de entrada de la red, que lo envía a la primera capa oculta. El algoritmo luego calcula la salida de todas las neuronas en esta capa (para cada instancia en el mini-lote). El resultado se pasa a la siguiente capa, su salida se calcula y se pasa a la siguiente capa, y así sucesivamente hasta obtener la salida de la última capa, la capa de salida. Esta es la pasada hacia adelante, que consiste en hacer predicciones y los resultados intermedios se preservan ya que son necesarios para la pasada hacia atrás. A continuación, el algoritmo mide el error de salida de la red (es decir, utiliza una función de pérdida que compara la salida deseada y la salida real de la red, y devuelve alguna medida del error). Luego, calcula cuánto contribuyó cada conexión de salida al error. Esto se hace analíticamente aplicando la regla de la cadena, lo que hace que este paso sea rápido y preciso. El algoritmo luego mide cuánto de estas contribuciones de error provienen de cada conexión en la capa anterior, nuevamente utilizando la regla de la cadena, trabajando hacia atrás hasta que el algoritmo llega a la capa de entrada. Como se explicó anteriormente,

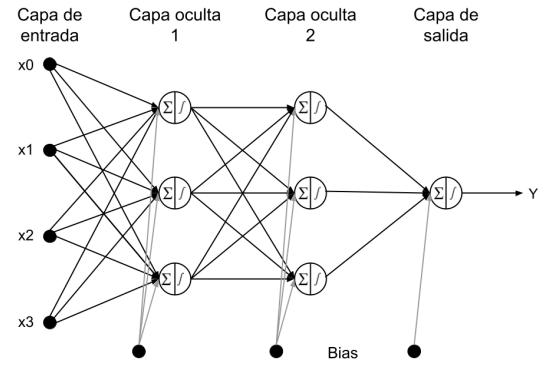


Figura 2.3: Arquitectura de un Perceptrón Multicapa con tres entradas, dos capas ocultas de tres neuronas cada una y una neurona de salida (las neuronas de sesgo se muestran aquí, pero normalmente son implícitas).

esta pasada inversa mide eficientemente el gradiente de error en todos los pesos de conexión en la red propagando el gradiente de error hacia atrás a través de la red (de ahí el nombre del algoritmo). Finalmente, el algoritmo realiza un paso de descenso de gradiente para ajustar todos los pesos de conexión en la red [28].

Este algoritmo es tan importante que vale la pena resumirlo nuevamente: para cada instancia de entrenamiento, el algoritmo de retropropagación primero hace una predicción (pasada hacia adelante) y mide el error, luego pasa por cada capa en reversa para medir la contribución del error de cada conexión (pasada hacia atrás), y finalmente ajusta los pesos de conexión para reducir el error (paso de Descenso de Gradiente).

Es importante inicializar aleatoriamente todos los pesos de conexión de las capas ocultas, de lo contrario, el entrenamiento fallará. Por ejemplo, si inicializas todos los pesos y sesgos a cero, todas las neuronas en una capa dada serán perfectamente idénticas, y por lo tanto la retropropagación las afectará de la misma manera, por lo que permanecerán idénticas. En otras palabras, a pesar de tener cientos de neuronas por capa, tu modelo actuará como si tuviera solo una neurona por capa: no será muy inteligente. Si en cambio inicializas aleatoriamente los pesos, rompes la simetría y permites que la retropropagación entrene un equipo diverso de neuronas [28].

Las capas básicas de esta arquitectura son las capas totalmente conectadas o *fully connected*.

La capa densa o capa totalmente conectada (*fully connected layer*) es la unidad básica del perceptrón multicapa. En una capa densa, cada neurona de la capa está conectada a todas las neuronas de la capa anterior. Esto permite una integración completa de la información entre las capas y es crucial para la capacidad de aprendizaje del modelo.

Las capas densas son utilizadas en muchos tipos de arquitecturas de redes neuronales, desde las redes *feedforward* básicas hasta redes más complejas como las convolucionales (CNN) y las recurrentes (RNN), sirviendo como el componente principal para la combinación de características aprendidas.

2.4.3. Autoencoder

Los *autoencoders* son una clase de redes neuronales artificiales utilizadas en aprendizaje no supervisado para aprender representaciones eficientes de los datos. Su funcionamiento consiste en codificar la entrada en una representación comprimida y significativa, y luego decodificarla de manera que la reconstrucción sea lo más similar posible a la entrada original [56]. La arquitectura básica de un *autoencoder* consta de tres partes: el *encoder*, el cuello de botella y el *decoder* (Figura 2.4).

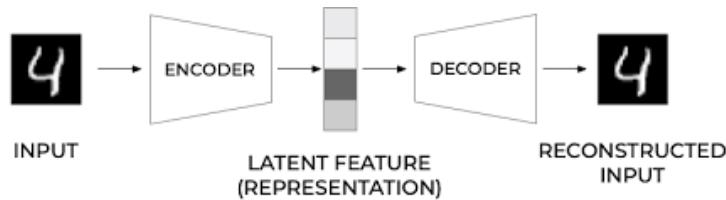


Figura 2.4: Arquitectura de un *autoencoder*. Fuente:[44].

Sean $z = f(x) : \mathbb{R}^n \rightarrow \mathbb{R}^m$ y $x' = g(z) : \mathbb{R}^m \rightarrow \mathbb{R}^n$ dos funciones que representan al codificador y decodificador respectivamente. Además, $n, m \in \mathbb{N}$ donde normalmente $n \leq m$. El *autoencoder* ilustrado en la figura 2.4 puede modelarse mediante la ecuación (2.17). Sea e y d dos conjuntos de índices pertenecientes al codificador y decodificador respectivamente. Entonces,

$$\begin{cases} z = f(x; w_e, b_e) \\ x' = g(z; w_d, b_d) \end{cases} \quad (2.17)$$

donde x' es la reconstrucción del *autoencoder*. Si $f(\cdot)$ y $g(\cdot)$ son redes neuronales, entonces w_i y b_i son las matrices de pesos y los vectores de sesgo con respecto a la red neuronal del codificador y decodificador respectivamente ($\forall i \in e, d$). El codificador mapea el conjunto de datos x al espacio de características z que normalmente suelen tener una menor dimensión. Por otra parte, el decodificador reconstruye los datos iniciales x a través de la representación comprimida oculta z [101]. Durante el entrenamiento, los parámetros del *autoencoder* se optimizan para minimizar la diferencia entre la entrada y la salida reconstruida, utilizando una función de pérdida que mide esta discrepancia, como por ejemplo la `binary_crossentropy` o el `mse`. Esto puede expresarse matemáticamente como:

$$\min_{f: \mathbb{R}^n \rightarrow \mathbb{R}^m; g: \mathbb{R}^m \rightarrow \mathbb{R}^n} \Delta(x, g \circ f(x)) \quad (2.18)$$

donde $\Delta : \mathbb{R} \rightarrow \mathbb{R}$ es la función de pérdida de reconstrucción, que mide la distancia entre la salida del decodificador y la entrada [8]. Esto concluye el proceso de entrenamiento de un *autoencoder*.

Tradicionalmente, los *autoencoders* se empleaban principalmente para reducir la dimensionalidad o para extraer características. No obstante, con el auge de diversos modelos de aprendizaje profundo, especialmente los modelos de redes generativas adversarias, los *autoencoders* han cobrado relevancia en la modelización generativa. Esto ha llevado a numerosos investigadores a proponer una variedad de modelos extendidos de *autoencoders*, como los *autoencoders* convolucionales (que veremos en más profundidad en la sección 2.4.4) o los recurrentes [101].

Las principales capas que se utilizan en esta red neuronal son las capas densas y las de aplanoamiento, aunque también se pueden utilizar capas convolucionales traspuestas y capas *unpooling* (sección 2.4.4) para autocodificadores convolucionales³ o LSTM en el caso de autocodificadores recurrentes⁴[28].

2.4.4. Red Neuronal Convolucional

Las redes neuronales convolucionales son una de las métodos de *machine learning* más importantes y utilizados en el campo de la ciberseguridad. Estas redes neuronales están diseñadas para procesar entradas almacenadas en matrices, como las imágenes [77]. La arquitectura de una CNN (Figura 2.5) consta de tres tipos de capas: capas de convolución, capas de *pooling* y la capa de clasificación.

Capa convolucional

La capa convolucional es la capa más importante de una CNN. En ella se extraen las características más significativas de la imagen de entrada, como los bordes, el color o la forma. Para ello, se aplica una convolución a la imagen con un filtro. Esta operación matemática se representa como $f(x * w)(t)$ donde x y w son dos funciones que se denominan entrada y núcleo de convolución respectivamente [74].

³Autocodificador para imágenes de gran tamaño

⁴Autocodificador específico para series temporales o secuencias.

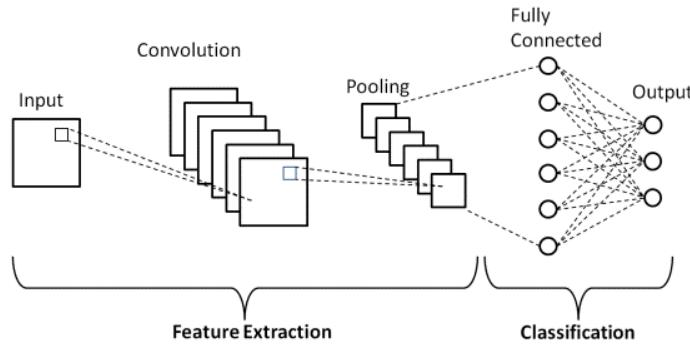


Figura 2.5: Arquitectura de una CNN con capas de convolución, *pooling* y clasificación. Fuente:[76].

En una CNN, la entrada de la convolución es una matriz multidimensional $I \in \mathcal{M}_{m \times n}(\mathbb{R})$ con $n, m \in \mathbb{N}$ y el núcleo de convolución es otra matriz $K \in \mathcal{M}_{i \times j}$ con $i, j \in \mathbb{N}$ de parámetros que se van ajustando durante el entrenamiento. A la matriz K también se le puede llamar filtro.

Cada píxel de la capa de convolución tiene una neurona. Estas se conectan con su campo receptivo⁵ de la capa anterior y les aplica la convolución [28]. Esta convolución se realiza con un solapamiento total del filtro, lo que resulta en una imagen de menor dimensión (Figura 2.6a). Si se desea mantener la misma dimensión, se puede aplicar *zero-padding*, que consiste en llenar con ceros la matriz para obtener las dimensiones deseadas (Figura 2.6b). En la figura 2.6 se muestran los campos receptivos de la neurona P y Q con un recuadro amarillo y verde respectivamente sobre la imagen I . La matriz de respuesta al aplicarle el kernel se llama mapa de características (O).

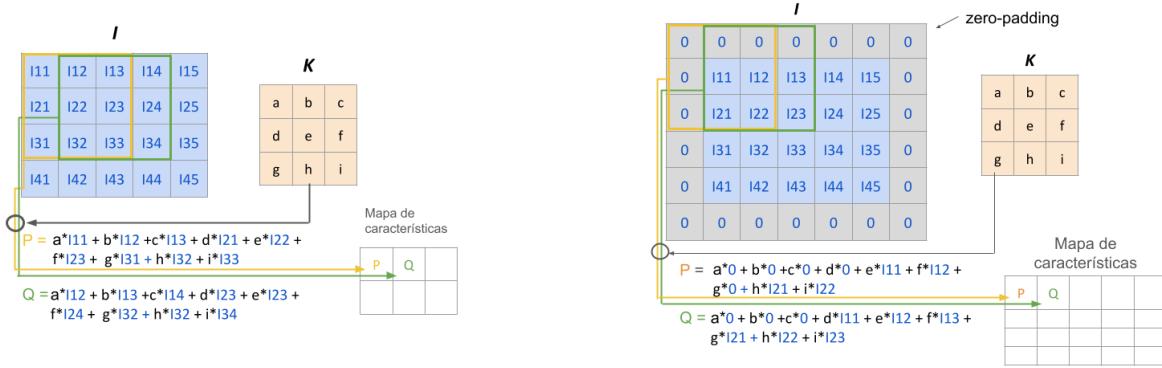


Figura 2.6: Convolución en 2D.

Se observa que el filtro se desplaza por la matriz I con paso unitario en vertical y horizontal. Este parámetro se llama *stride* y su valor depende de el objetivo que se quiera lograr con esta capa convolucional (Figura 2.7).

Una longitud de paso de 1 se utiliza normalmente para extraer el máximo número de características, ya que proporciona el máximo solapamiento entre el núcleo y la entrada. Por otro lado, cuando la longitud de paso es mayor que 1, los campos receptivos se solapan menos y producen una salida más pequeña. Si en la figura 2.7, la longitud de paso fuera 3, habría problemas con el espaciado, ya que el campo receptivo no encajaría alrededor de la entrada como un número entero [98].

⁵Región de entrada que contribuye a la salida generada por el filtro

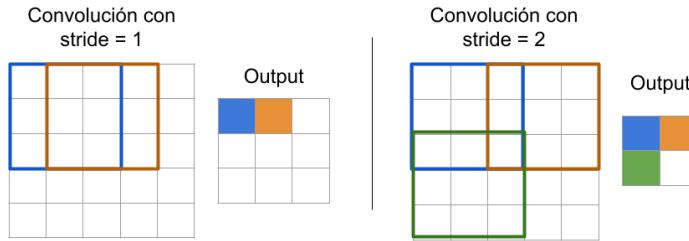


Figura 2.7: Campos receptivos en una convolución. Adaptación de imagen de [98]

Por simplicidad, se ha usado siempre un único kernel, pero se puede generalizar a varios filtros, creando un mapa de características por cada uno. En cada uno de estos mapas hay una neurona por píxel y todas ellas comparten los mismos parámetros, lo que reduce considerablemente el número de parámetros del modelo. El campo receptivo de una neurona ahora se extiende por los mapas de características de todas las capas anteriores [28].

Toda la información anterior se resume en la siguiente ecuación [74]:

$$z_{i,j,k} = b_k + \sum_{u=0}^{f_h-1} \sum_{v=0}^{f_w-1} \sum_{k'=0}^{f'_n-1} x_{i',j',k'} \cdot w_{u,v,k',k} \quad \text{con } \begin{cases} i' = i \cdot s_h + u \\ j' = j \cdot s_w + v \end{cases} \quad (2.19)$$

donde:

- $z_{i,j,k}$ es la salida de la neurona ubicada en la fila i , columna j en el mapa de características k de la capa convolucional (capa l).
- s_h y s_w son los pasos de avance vertical y horizontal.
- f_h y f_w son la altura y la anchura del campo receptivo y f'_n es el número de mapas de características de la capa anterior (capa $l - 1$).
- $x_{i',j',k'}$ es la salida de la neurona situada en la fila i' , columna j' , mapa de características k' .
- b_k es el sesgo para el mapa de características k (en la capa l).
- $w_{u,v,k',k}$ es el peso de conexión entre cualquier neurona del mapa de características k de la capa l y su entrada situada en la fila u , columna v (relativa al campo receptivo de la neurona) y el mapa de características k' .

Capa de pooling

El siguiente tipo de capa de las CNN son las *pooling*, cuyo objetivo es reducir la imagen de entrada para disminuir la carga computacional, el uso de memoria y el número de parámetros, limitando así el riesgo de sobreajuste y proporcionando robustez contra el ruido y las distorsiones. Esta capa se suele colocar entre las capas de convolución, permitiendo reducir el tamaño de las imágenes mientras se preservan las características más importantes [77]. Al igual que en las capas convolucionales, sus neuronas están conectadas a un pequeño grupo de neuronas de la capa anterior a las que se le aplica una función de agregación⁶. Las tres funciones más comunes son el promedio, la suma y el máximo. La Figura 2.8 muestra una capa de *max pooling*, que es el tipo más común [28].

⁶Las funciones de agregación devuelven un valor único de un conjunto de registros.

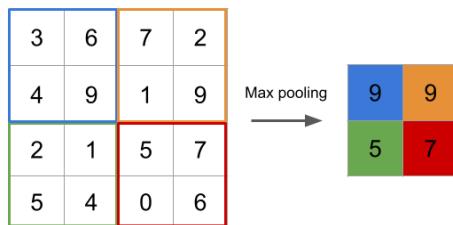


Figura 2.8: Capa de *max pooling* con un kernel de 2×2 , *stride* 2 y sin *padding*.

Además de reducir el número de operaciones, el número de parámetros y ayudar con el *overfitting*, la capa de *max pooling* introduce cierto nivel de invarianza a pequeñas translaciones, ya que si un píxel se traslada hacia la derecha, la salida también debería trasladarse un píxel hacia la derecha, como se ilustra en la Figura 2.9. Esto significa que pequeñas variaciones en la posición de las características dentro de la imagen no afectan significativamente la salida.

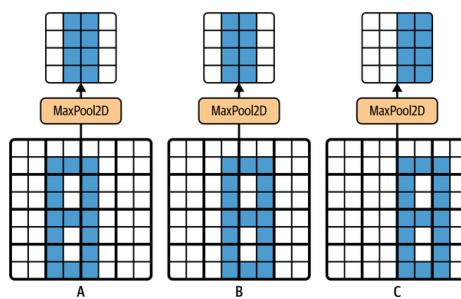


Figura 2.9: Invarianza a translaciones pequeñas mediante una capa de *max pooling*. Fuente [28]

Una capa típica de convolución se compone de tres etapas. En la primera etapa, se realizan múltiples convoluciones para producir un conjunto de activaciones lineales. En la segunda etapa, cada activación lineal se procesa a través de una función de activación no lineal, como la función de activación rectificada lineal (ReLU). Esta etapa a menudo se denomina etapa de detección. En la tercera etapa, se aplica una función de *pooling* para modificar la salida de la capa. Este tipo de capa de convolución compuesta será de utilidad en los experimentos realizados en capítulos posteriores, como por ejemplo en la creación de una M-CNN para el problema de clasificación de malware del Capítulo 3 [74].

Capa de Unpooling

En la figura 2.8 se presenta un ejemplo gráfico relacionado con la operación de *max pooling*. Durante este proceso, es posible mantener un mapa de localizaciones, que señala la posición de origen del valor máximo que ha generado la capa. Este mapa permite ejecutar la operación inversa conocida como reconstrucción (*Unpooling*), colocando los valores obtenidos en las posiciones indicadas por el mapa de localizaciones. Una vez completada la reconstrucción, se puede realizar una interpolación lineal usando el método del vecino más próximo para obtener el mapa de características reconstruido [74]. Este tipo de capa hace el proceso inverso de la capa de *pooling*. En vez de reducir la dimensión de entrada, la aumenta utilizando o bien interpolación lineal o bilineal o bien rellenando con ceros el resto de casillas [74]. La operación de *Unpooling* se suele emplear junto con la operación de convolución transpuesta, como se describe más adelante, y se utiliza, por ejemplo, en el modelo de *autoencoder* utilizado en la sección 3.3, en un *autoencoder* convolucional.

En Keras, esta capa se denomina `UpSampling2D`.

Convolución Transpuesta

La necesidad de realizar convoluciones transpuestas surge generalmente de la necesidad de aplicar una transformación en la dirección inversa a una convolución directa o normal, como la proyección de mapas de características a un espacio de mayor dimensión. Aunque a veces se denomina deconvolución, esta es en realidad una operación distinta. En las redes neuronales convolucionales (CNN), esta operación es más compleja que en las redes totalmente conectadas, donde solo se requiere el uso de una matriz de pesos transpuesta [74].

Considere la convolución mostrada en la figura 2.6a. Teníamos que $I \in \mathcal{M}_{m \times n}(\mathbb{R})$ con $n, m \in \mathbb{N}$ y $O \in \mathcal{M}_{i \times j}(\mathbb{R})$ con $i, j \in \mathbb{N}$ la entrada y la salida respectivamente. Expandamos ahora estas matrices a un vector $I' \in \mathcal{M}_{m \cdot n}(\mathbb{R})$ y $O' \in \mathcal{M}_{i \cdot j}(\mathbb{R})$ de izquierda a derecha y de arriba a abajo. Con esta nueva notación, la operación de convolución 2.4.4 también puede expresarse como $C \cdot I' = O'$ como podemos observar en la figura 2.10:

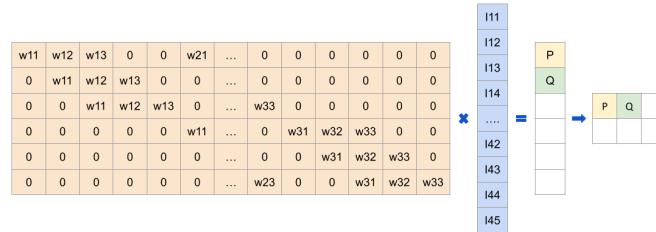


Figura 2.10: Convolución y representación matricial de la operación $C \cdot I' = O'$

Una vez tenemos estos dos vectores transformados, hay que modificar la matriz C original. Para ello, se añaden múltiples ceros de forma que al multiplicarla por O' , nos de resultado el vector I' . Esta nueva matriz la denominamos como $C' \in \mathcal{M}_{(i \cdot j) \times (n \cdot m)}$. Los elementos no nulos de C' se corresponden a los elementos w_{ij} del núcleo K . Aquí, i y j denotan la fila y la columna, respectivamente.

Usando esta formulación, para poder hacer el paso inverso y obtener la matriz original I a partir de O' y C' , tenemos que aplicar una trasposición de la matriz C' como se muestra en la figura 2.11 [74].

Con la operación mencionada, a partir del vector C'^t y O' , obtenemos la matriz I' como un vector que hay que redimensionar para obtener la entrada original I [74].

Al igual que en las capas convolucionales originales, también puede añadirse *zero-padding* a la matriz y con desplazamientos (strides) superiores a la unidad [74].

Estas dos últimas capas son especialmente utilizadas en los *autoencoders* convolucionales, donde hay que construir y reconstruir una versión reducida de los datos de entrada.

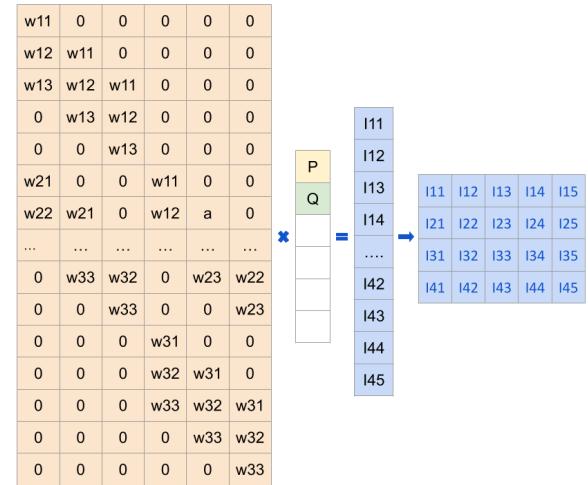


Figura 2.11: Convolución traspuesta. $C'^t \cdot O' = I'$.

Capas Totalmente Conectadas (Fully Connected)

Por último están las capas totalmente conectadas (*fully connected*), que realizan la clasificación sobre la salida generada por las capas de convolución y *pooling*. Como el input de una capa densa debe ser un vector, primero se debe aplanar la salida de la última capa para poder utilizar después esta capa. Cada una de sus neuronas está conectada a todas las de la capa anterior, estableciendo una red densa de conexiones. Este tipo de neuronas suele ir seguido de una capa Dropout para mejorar la generalización del modelo y evitar un problema muy común en el aprendizaje automático como es el sobreajuste [39].

2.4.5. Red Neuronal Recurrente

Las Redes Neuronales Recurrentes (RNN) son un tipo de red neuronal diseñada para procesar secuencias de datos. A diferencia de las redes neuronales *feedforward* tradicionales, que asumen que las entradas y salidas son independientes, las RNN tienen conexiones recurrentes que permiten la propagación de información temporal, haciendo posible el modelado de dependencias a lo largo del tiempo. Esta característica es clave en tareas que involucran datos secuenciales, como series temporales, texto o audio.

Neuronas y Capas Recurrentes

En una neurona recurrente, la estructura se extiende para incorporar no solo la entrada en el tiempo t , $x(t)$, sino también su propio **estado oculto** en el tiempo anterior $h(t - 1)$. Este mecanismo dota a la neurona de una forma de memoria temporal, permitiendo que el modelo mantenga información sobre secuencias pasadas. La Figura 2.12 muestra cómo una neurona recurrente se despliega en el tiempo.

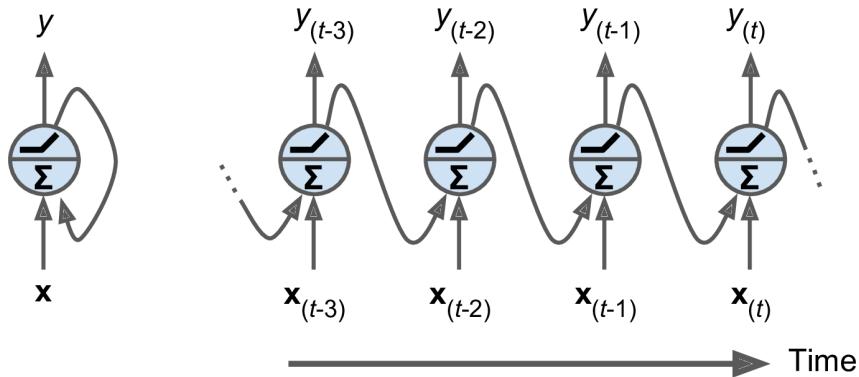


Figura 2.12: Una neurona recurrente desplegada en el tiempo. Fuente [28]

Matemáticamente, el **estado oculto** de una neurona recurrente en el tiempo t se define como:

$$h(t) = \phi(W_x \cdot x(t) + W_y \cdot h(t - 1) + b) \quad (2.20)$$

donde W_x y W_y son las matrices de pesos asociadas con la entrada y el estado oculto, respectivamente, b es el vector de sesgo, y ϕ es la función de activación.

Para una capa completa de neuronas recurrentes, la **salida** para un lote de instancias en el tiempo t se calcula como:

$$y(t) = \phi(h(t) \cdot W_y) \quad (2.21)$$

donde $h(t)$ es el estado oculto en el tiempo t , W_y es la matriz de pesos asociada a la capa de salida e $y(t)$ es la predicción final en el tiempo t .

Puede resultar un poco confuso estos dos conceptos, por ello vamos a resumirlos para ver sus diferencias. El estado oculto en una RNN hace referencia a la representación interna (memoria) de los datos anteriores de la red hasta ese momento dado. Este estado oculto se actualiza en cada paso de tiempo y sirve como entrada para calcular la siguiente salida. Por otro lado, la predicción es el resultado final que la red genera después de procesar toda la secuencia de datos anterior. Se basa en la información contenida en el último estado oculto.

Entrenamiento de RNNs

El entrenamiento de una RNN se lleva a cabo utilizando una técnica conocida como *backpropagation through time* (BPTT). Este método implica desplegar la red a lo largo del tiempo y luego aplicar retropropagación a la red desplegada. Inicialmente, se realiza una pasada hacia adelante a través de la red desplegada, calculando la secuencia de **salidas** en cada paso temporal (representado en la figura 2.13 con las líneas discontinuas). Posteriormente, se evalúa la salida con una función de costo $C(y(0), y(1), \dots, y(T))$, donde T es el tiempo máximo considerado. Es importante señalar que la función de costo puede ignorar ciertas salidas, dependiendo de la tarea específica (por ejemplo, en la Figura 2.13, la función de costo se calcula usando las tres últimas salidas de la red, $y(2)$, $y(3)$ e $y(4)$, por lo que los gradientes se propagan a través de estas tres salidas, pero no a través de $y(0)$ y $y(1)$).

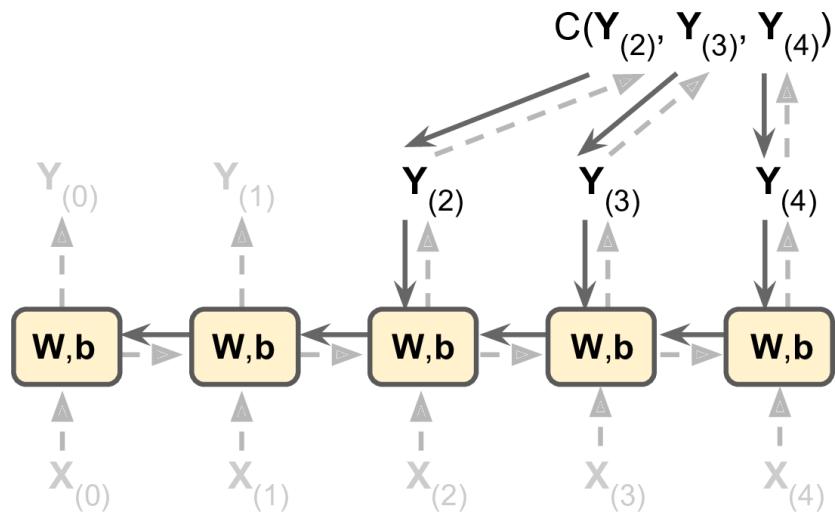


Figura 2.13: *Backpropagation through time* (BPTT) en una RNN. Fuente [28]

Una vez calculado el costo, los gradientes de la función de costo se propagan hacia atrás a través de la red desplegada. Este proceso de retropropagación ajusta los parámetros del modelo utilizando los gradientes calculados. Un aspecto crítico en BPTT es que los mismos parámetros de la red (como los pesos W y los sesgos b) se utilizan en cada paso temporal, lo que implica que la retropropagación debe sumar correctamente los gradientes a lo largo de todos los pasos temporales.

Aunque el BPTT permite el entrenamiento de RNNs, existen desafíos con este método. Un ejemplo es la desaparición de gradientes, que ocurre cuando los gradientes se vuelven muy pequeños, impidiendo la actualización efectiva de los parámetros. Otro problema consiste en la explosión de gradientes, que se refiere a la situación en la que los gradientes crecen exponencialmente, causando inestabilidad en el entrenamiento.

Para mitigar estos problemas, se pueden usar técnicas como *gradient clipping*, que limita el tamaño de los gradientes para evitar la explosión, el uso de activaciones saturadas que estabilizan el flujo de gradientes o el uso de las celdas *Long Short-Term Memory*.

Long Short-Term Memory

Para superar las limitaciones de las RNN básicas, se han desarrollado celdas de *Long Short-Term Memory* (LSTM). Introducidas por [38], las LSTM están diseñadas para almacenar y gestionar información durante largos períodos de tiempo, abordando los problemas de gradientes mediante el uso de mecanismos de puertas. Las celdas de este tipo de redes son una extensión de las redes neuronales recurrentes (RNN), diseñadas para mejorar su capacidad de recordar información relevante durante largos períodos. Esta mejora permite a las RNN retener sus entradas por mucho más tiempo, ya que las LSTM almacenan la información en una estructura de memoria que es análoga a la memoria de un ordenador. En este sentido, una célula de LSTM puede realizar operaciones de lectura, escritura y eliminación de información en su memoria.

La estructura de memoria en una LSTM puede ser conceptualizada como una celda “cerrada”, donde “cerrada” implica que la celda decide si debe almacenar o descartar la información en función de su relevancia. Esta decisión se realiza a través de las puertas de la LSTM, cuya apertura o cierre depende de los pesos asignados, los cuales se ajustan durante el proceso de aprendizaje. En esencia, la red aprende a través del tiempo qué información debe ser retenida y cuál puede ser descartada.

Cada célula de una red LSTM incluye tres tipos de puertas que regulan el flujo de información: la puerta de entrada, la puerta de olvido y la puerta de salida. La puerta de entrada controla la incorporación de nueva información a la memoria, la puerta de olvido decide si se debe eliminar información previa y la puerta de salida determina si la información almacenada debe influir en la salida en el tiempo presente.

Estas puertas utilizan funciones sigmoides, lo que les permite operar en un rango continuo entre 0 y 1. Las LSTM resuelven el problema del desvanecimiento del gradiente (*vanishing gradients*) al mantener los gradientes suficientemente pronunciados, resultando en un entrenamiento eficiente y una alta precisión.

La arquitectura de una celda LSTM se representa en la Figura 2.14, donde FC hace referencia a una capa *fully connected* que se utilizan para procesar los datos de entrada antes de interactuar con las puertas. Además, a continuación se describen las operaciones de cada puerta.

En una red LSTM, la **puerta de olvido** se encarga de determinar qué información del estado de la celda debe ser eliminada en cada instante de tiempo. Sea $f_t \in \mathbb{R}^n$ el vector de activaciones de la puerta de olvido en el tiempo t . Sea $W_f \in \mathbb{R}^{n \times m}$ la matriz de pesos asociada a la puerta de olvido que incluye los pesos del estado oculto y de los datos de entrada, y $b_f \in \mathbb{R}^n$ el vector de sesgos para la puerta del olvido. Consideremos $h(t-1) \in \mathbb{R}^m$ el estado oculto en el tiempo $t-1$ y $x(t) \in \mathbb{R}^p$ la entrada actual, cuya concatenación está dada por $[h(t-1), x(t)] \in \mathbb{R}^{m+p}$. La activación a través de la función σ de la puerta de olvido se define por la ecuación:

$$f_t = \sigma(W_f \cdot [h(t-1), x(t)] + b_f) \quad (2.22)$$

La **puerta de entrada** determina qué información nueva debe ser añadida al estado de la celda. Sea $i_t \in \mathbb{R}^n$ el vector de activaciones de la puerta de entrada en el tiempo t . Definimos

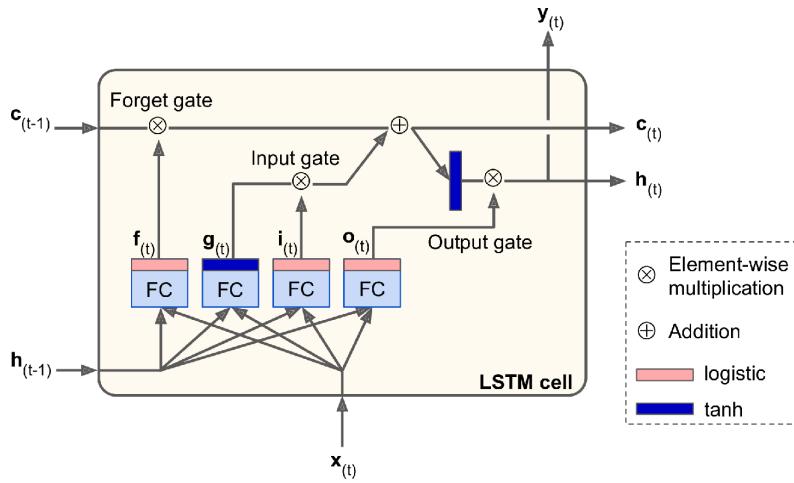


Figura 2.14: Arquitectura de una celda LSTM. Fuente [28]

$W_i \in \mathbb{R}^{n \times m}$ como la matriz de pesos y $b_i \in \mathbb{R}^n$ como el vector de sesgos para la puerta de entrada. La activación de la puerta de entrada está dada por:

$$i_t = \sigma(W_i \cdot [h(t-1), x(t)] + b_i) \quad (2.23)$$

Para **actualizar el estado** de la celda con las nuevas posibles informaciones, consideremos $\tilde{C}_t \in \mathbb{R}^n$ como el vector de nuevas posibles actualizaciones en el tiempo t . Sea $W_C \in \mathbb{R}^{n \times m}$ la matriz de pesos y $b_C \in \mathbb{R}^n$ el vector de sesgos para actualizar el estado de la celda. La actualización se describe mediante:

$$\tilde{C}_t = \tanh(W_C \cdot [h(t-1), x(t)] + b_C) \quad (2.24)$$

El **estado de la celda** $C_t \in \mathbb{R}^n$ en el tiempo t se obtiene combinando la información retenida y la nueva entrada. Sea $C_{t-1} \in \mathbb{R}^n$ el estado de la celda anterior. La actualización del estado de la celda se calcula como:

$$C_t = f_t \cdot C_{t-1} + i_t \cdot \tilde{C}_t \quad (2.25)$$

La **puerta de salida** decide qué información del estado de la celda debe ser utilizada en la salida. Sea $o_t \in \mathbb{R}^n$ el vector de activaciones de la puerta de salida en el tiempo t . Consideraremos $W_o \in \mathbb{R}^{n \times m}$ la matriz de pesos y $b_o \in \mathbb{R}^n$ el vector de sesgos para la puerta de salida. Esta activación está dada por:

$$o_t = \sigma(W_o \cdot [h(t-1), x(t)] + b_o) \quad (2.26)$$

Finalmente, el **estado oculto** $h(t) \in \mathbb{R}^m$ en el tiempo t se obtiene combinando la activación de la puerta de salida con el estado de la celda actual. Esto se formaliza como:

$$h(t) = o_t \cdot \tanh(C_t) \quad (2.27)$$

Keras también incluye otras variantes de RNN, como las Unidades Recurrentes Gated (GRU). Introducidas en 2014, las GRU simplifican los principios de las LSTM, ofreciendo un rendimiento comparable pero con una mayor eficiencia computacional.

2.5. Función de perdida

En el aprendizaje supervisado, las funciones de pérdida son herramientas muy importantes para evaluar la precisión con la que un modelo predice los resultados esperados. Su objetivo es cuantificar el grado de error en las predicciones realizadas por el modelo, lo cual es fundamental para el ajuste de sus parámetros durante el entrenamiento. En términos generales, estas funciones se dividen en dos grandes categorías: las utilizadas para problemas de clasificación y las utilizadas para problemas de regresión [74]. Ambas buscan minimizar el error, pero lo hacen con diferentes enfoques dependiendo de la naturaleza de la variable de salida.

2.5.1. Clasificación

En los problemas de clasificación, la tarea del modelo es asignar una etiqueta a cada observación basada en las características de entrada. La etiqueta es una variable categórica que indica a cuál de varias categorías pertenece cada observación. Las funciones de pérdida en clasificación miden la discrepancia entre las etiquetas reales y las etiquetas predichas, o las probabilidades de las categorías predichas.

Entropía Cruzada

La entropía cruzada mide la diferencia entre la distribución de probabilidad verdadera de las etiquetas y la distribución de probabilidad predicha por el modelo. La variante general de esta función se denomina **entropía cruzada categórica** que se utiliza cuando se predicen múltiples clases. Se calcula como:

$$H(p, q) = - \sum_x p(x) \log q(x) \quad (2.28)$$

donde $H(p, q) \in \mathbb{R}$ devuelve el valor que mide la diferencia entre las distribuciones de probabilidad del conjunto de predicciones del modelo ($q(x)$) y las etiquetas reales ($p(x)$) [74].

Un caso particular de esta función se presenta cuando tenemos un problema de clasificación binaria. Su ecuación se reduce a:

$$H(p, q) = -[p \log q + (1 - p) \log(1 - q)] \quad (2.29)$$

donde p es la probabilidad de la etiqueta verdadera (1 para la clase positiva, 0 para la clase negativa), y q es la probabilidad predicha para la clase positiva. Esta variante es más sencilla y específica para problemas donde las etiquetas son binarias [74].

Coeficiente de Dice

El coeficiente de Dice es una métrica especialmente útil en problemas de segmentación de imágenes y clasificación. Esta métrica calcula la similitud entre el conjunto de píxeles predichos y el

conjunto de píxeles verdaderos, y se define como:

$$D = \frac{2|P \cap G|}{|P| + |G|} \quad (2.30)$$

donde P representa el conjunto de píxeles predichos y G representa el conjunto de píxeles verdaderos. La interpretación de esta función es que un valor más alto indica una mayor superposición entre las predicciones y la verdad del terreno, siendo 1 el valor ideal cuando hay coincidencia perfecta [74].

2.5.2. Regresión

En los problemas de regresión, la tarea del modelo es predecir una variable de salida continua basada en las características de entrada. Las funciones de pérdida en regresión cuantifican la discrepancia entre los valores continuos predichos y los valores reales.

Error Cuadrático Medio (ECM)

El error cuadrático medio (MSE, *mean square error*) es una función de pérdida que calcula la media de los cuadrados de los errores entre las predicciones y los valores reales. Se define como:

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2 \quad (2.31)$$

donde n es el número de muestras, y_i es el valor real y \hat{y}_i es el valor predicho. Esta función penaliza los errores grandes más severamente que los pequeños, debido al cuadrado del término de error. Un valor de MSE más bajo indica predicciones más cercanas a los valores reales [74].

Error Absoluto Medio (EAM)

El error absoluto medio (MAE, *mean absolute error*) mide la media de los valores absolutos de las diferencias entre las predicciones y los valores reales. Está dado por:

$$\text{MAE} = \frac{1}{n} \sum_{i=1}^n |y_i - \hat{y}_i| \quad (2.32)$$

A diferencia del MSE, el MAE no penaliza tanto los errores grandes, ya que no los eleva al cuadrado. Esto hace que sea más robusto frente a los valores atípicos. Un MAE más bajo indica un modelo más preciso [74].

Pérdida Basada en Cuantiles

La función de pérdida basada en cuantiles (*quantile loss*) se utiliza en regresión cuando se quiere predecir un intervalo en lugar de un punto concreto. Sea $\tau \in [0, 1]$ un cuantil, entonces la función

de pérdida se define como:

$$L_\tau = \sum_{i=1}^n (\tau \max(y_i - \hat{y}_i, 0) + (1 - \tau) \max(\hat{y}_i - y_i, 0)) \quad (2.33)$$

2.6. Función de activación.

Entre capa y capa de las redes neuronales, cada neurona suma ponderadamente las entradas provenientes de las neuronas de la capa anterior. Este resultado, se lo pasa a una función no lineal, llamada función de activación, que permiten que la red neuronal pueda aprender, representar patrones complejos y crear relaciones no lineales entre las características de entrada y la salida. El valor resultante de aplicarle la función es el valor que se le queda asociado a esta neurona, para que futuras capas puedan usarlo. En esta sección, describimos varias funciones de activación populares, sus propiedades y aplicaciones [74].

Softmax

La función **Softmax** se utiliza comúnmente en la capa de salida de redes neuronales para problemas de clasificación multiclase. Convierte un vector de valores arbitrarios en un vector de probabilidades, cuya suma es 1. La función Softmax se define como:

$$\text{Softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^K e^{z_j}} \quad (2.34)$$

donde z_i es el valor de la i -ésima neurona de salida, y K es el número total de neuronas en la capa de salida. La interpretación de esta función es que transforma las salidas en probabilidades, facilitando la toma de decisiones sobre la clase a la que pertenece una muestra en función de las probabilidades más altas [74].

Sigmoide

La función sigmoide, también conocida como la función logística, convierte entradas en el rango $(0, 1)$. Está definida como:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \quad (2.35)$$

Sin embargo, tiene problemas como la saturación del gradiente. Esto ocurre cuando los valores de entrada se aproximan a 0 o 1. En estas regiones, el gradiente de la función sigmoide tiende a 0. Esto se puede ver derivando la función sigmoide:

$$\sigma'(x) = \frac{e^{-x}}{(1 + e^{-x})^2} \quad (2.36)$$

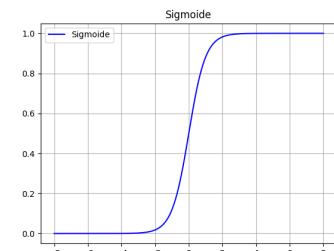


Figura 2.15: Gráfica de la función sigmoide.

Cuando x tiende a infinito, $\sigma(x)$ tiende a 1, luego $\sigma'(x)$ tiende a 0. Cuando x tiende a menos infinito, $\sigma(x)$ tiende a 0, y nuevamente $\sigma'(x)$ tiende a 0. En ambos extremos, el gradiente tiende a 0, lo que afecta la actualización de los pesos durante el entrenamiento mediante descenso de gradiente. En otras palabras, los pesos se actualizan muy lentamente en estas regiones, lo que puede llevar a una convergencia lenta.

Función Tangente hiperbólica

La función **tangente hiperbólica (tanh)** es similar a la sigmoide pero escala las salidas al rango $[-1, 1]$:

$$\tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^x + e^{-x}}{e^x - e^{-x}} \quad (2.37)$$

Aunque también sufre de saturación del gradiente, *tanh* centra las salidas alrededor de cero, lo que puede acelerar el entrenamiento [74].

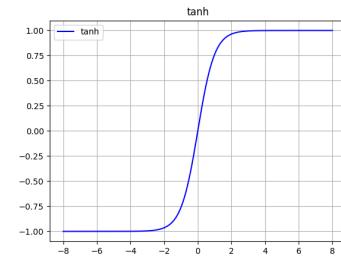


Figura 2.16: Gráfica de la función tangente hiperbólica.

Función ReLU

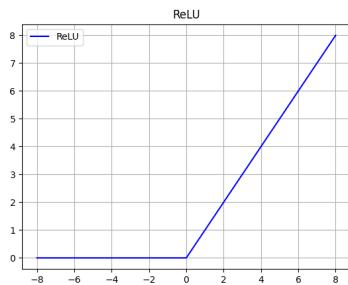


Figura 2.17: Gráfica de la función ReLU.

La **función ReLU** es popular debido a su simplicidad y efectividad para solventar el problema de la saturación del gradiente.

$$f(x) = \max(0, x) \quad (2.38)$$

Sin embargo, la función ReLU tiene algunos problemas, como el problema de la “ReLU muerta” [59] y la falta de diferenciabilidad en cero. Para este problema último, se suele asignar un valor arbitrario para este valor como 0, 0.5 o 1 [74]. Esto ocurre cuando se aprende un sesgo negativo grande, haciendo que la salida de la neurona sea siempre cero, sin importar la entrada [5]. Por eso, a continuación, discutimos algunas variantes de ReLU.

Función Leaky ReLU (LReLU)

Fue una de las primeras funciones de activación basadas en ReLU [59].

$$f(x) = \max(x \cdot 10^{-3}, x) \quad (2.39)$$

Esta función de activación permite que haya un pequeño gradiente cuando la salida es menor que cero. Sin embargo, se ha demostrado que Leaky ReLU funciona de manera casi idéntica a ReLU, lo que no mejora significativamente su rendimiento [94]. Además, se propuso una versión aleatoria de ReLU, donde $x \in U(l, u)$, $0 \leq l < u < 1$ [5].

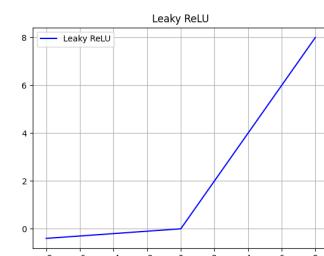


Figura 2.18: Gráfica Leaky ReLU.

Función paramétrica ReLU(PReLU)

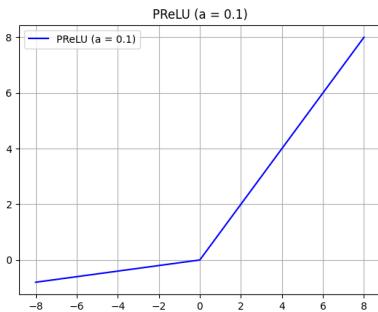


Figura 2.19: Gráfica de la función Paramétrica ReLU con $\alpha = 10^{-1}$.

Una forma de generalizar la función anterior es introduciendo un parámetro α para ajustar la pendiente para entradas negativas. Esta función se llama PReLU y se define:

$$f_\alpha(x) = \begin{cases} \alpha \cdot x & \text{si } x < 0 \\ x & \text{si } x \geq 0 \end{cases} \quad (2.40)$$

Cuando $\alpha = 0$, la función se corresponde con ReLU y cuando $\alpha > 0$, con la LReLU. La función se adapta durante el entrenamiento, permitiendo más flexibilidad que ReLU o LReLU.

Función Softplus

La función softplus puede considerarse una aproximación suave de la función ReLU. Se define como:

$$f(x) = \log(1 + \exp(x)) \quad (2.41)$$

La forma más suave de la función y la ausencia de puntos no diferenciables podrían sugerir un mejor comportamiento y un entrenamiento más fácil como función de activación. Sin embargo, los resultados experimentales obtenidos tienden a contradecir esta hipótesis, sugiriendo que las propiedades de ReLU pueden facilitar el entrenamiento supervisado mejor que las funciones softplus [5].

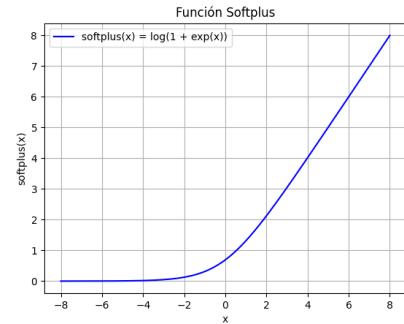


Figura 2.20: Gráfica de la función Softplus.

Función ELU

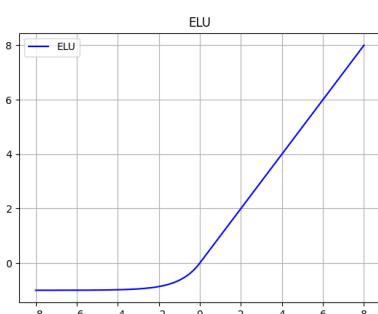


Figura 2.21: Gráfica de la función ELU.

La **función ELU** es una función de activación que mantiene la identidad para argumentos positivos pero con valores no nulos para los negativos. Se define como:

$$f_\alpha(x) = \begin{cases} x & \text{si } x > 0 \\ \alpha \cdot (\exp(x) - 1) & \text{de lo contrario} \end{cases}$$

donde α controla el valor para entradas negativas. Los valores dados por las unidades ELU empujan la media de las activaciones más cerca de cero, permitiendo una fase de aprendizaje más rápida, a costa de un hiperparámetro extra (α) que necesita ser establecido durante el entrenamiento [5].

Las funciones de activación juegan un papel esencial en la formación de redes neuronales al permitir que las redes aprendan y generen relaciones complejas. Desde funciones clásicas como la *sigmoide* y *tanh*, hasta variantes más recientes como *ReLU*, *LReLU* y *ELU*, cada función tiene sus propios beneficios y limitaciones. La elección de la función de activación depende del tipo de problema al que nos enfrentemos y de lo que se busque con el modelo, pero tiene que ser una elección acertada, ya que puede influir significativamente en el rendimiento del modelo [74]. Además, permiten resolver el desvanecimiento del gradiente durante el proceso de aprendizaje, evitando que el gradiente se acerque a cero, cumpliendo la hipótesis de derivabilidad [74].

2.7. Sobreajuste del modelo.

El sobreajuste es uno de los principales desafíos a los que se enfrentan los expertos en *machine learning* a día de hoy. Para poder explicar y presentar diferentes soluciones a este problema, vamos a ver primero algunos conceptos.

El **sobreajuste** (también conocido como *overfitting*) se produce cuando un modelo se ajusta demasiado bien a los datos de entrenamiento, capturando incluso el ruido y las fluctuaciones aleatorias. Este ajuste excesivo provoca que el modelo tenga un rendimiento excelente en los datos de entrenamiento pero un rendimiento deficiente en datos nuevos. El *overfitting* suele ocurrir cuando el modelo es demasiado complejo en comparación con la cantidad de datos disponibles.

El **subajuste** (también conocido como *underfitting*) ocurre cuando el modelo es demasiado simple para capturar la estructura subyacente en los datos. Esto se manifiesta con un bajo ajuste del polinomio tanto a los datos de entrenamiento como a los de validación, indicando que el modelo no ha capturado patrones importantes en los datos.

En la primera gráfica de la Figura 2.22, se puede observar como el modelo no es capaz de captar la complejidad del problema, fallando en las predicciones tanto en los datos de entrenamiento como en los datos nuevos. En la segunda gráfica se puede ver que el modelo captura perfectamente todos los puntos de entrenamiento, pero fallará al predecir datos nuevos debido a que ha aprendido el ruido presente en los datos de entrenamiento.

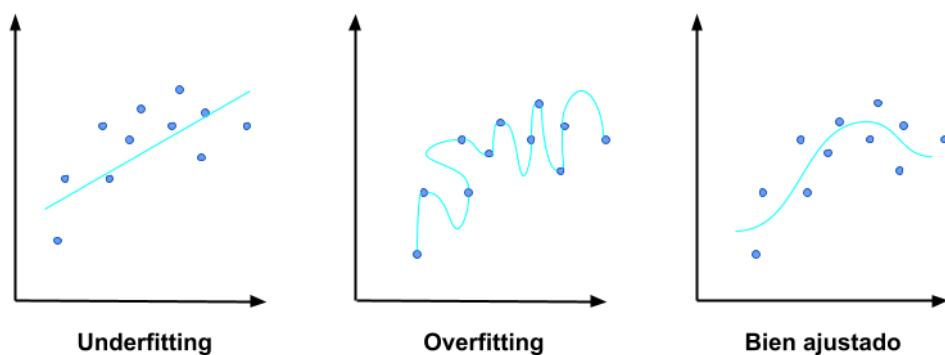


Figura 2.22: Ejemplo de sobreajuste, subajuste y buen ajuste.

Por último, la gráfica de más a la derecha presenta un ajuste idóneo para este conjunto de datos, ya que se puede observar como se ajusta a todos los puntos pero sin capturar ruido. Para llegar a este tipo de ajuste y evitar los dos primeros casos, vamos a explorar diferentes métodos para mitigar el sobreajuste, como la regularización l_1 y l_2 , las capas *dropout* o la parada temprana.

2.7.1. Regularización l1 y l2

La regularización ayuda a evitar el sobreajuste, lo cual ocurre cuando un modelo es demasiado complejo y comienza a captar el ruido o los patrones irrelevantes en los datos de entrenamiento. En lugar de eso, la regularización favorece la creación de modelos más simples que pueden identificar los patrones importantes y generalizar mejor a datos nuevos. Esta técnica es especialmente útil cuando se tiene una cantidad limitada de datos, conjuntos de datos con muchas características o modelos con muchos parámetros [74].

Para implementarla, se añade un término de regularización a la función de pérdida durante el entrenamiento. Este término penaliza ciertos parámetros del modelo, ajustando el total de la pérdida. La intensidad de la regularización se controla mediante un parámetro que determina el equilibrio entre ajustar los datos y reducir el impacto de coeficientes muy grandes [28].

$$\text{loss}_r(w) = \text{loss}(w) + \beta \cdot f(w) \quad (2.42)$$

Aquí, $\text{loss}_r(w)$ es la función de pérdida final después de aplicarle una regularización. $\text{loss}(w)$ es la función de pérdida original que mide el ajuste del modelo a los datos, y $w \in \mathbb{R}^n$ representa los parámetros del modelo (pesos y sesgos). El parámetro β controla la intensidad de la regularización, equilibrando entre ajustar el modelo a los datos y mantener los valores de los parámetros bajo control. Por último, el término de regularización $f(w)$ penaliza los parámetros del modelo, siendo comúnmente la regularización $l2$ o la $l1$.

La **regularización $l2$** (o regresión Ridge) añade una penalización proporcional a la suma de los pesos de cada capa al cuadrado. La función de pérdida regularizada $l2$ se define como $\sum_i^n w_i^2$ (donde n es el número de neuronas de esa capa). Sustituyendo en la ecuación 2.7.1, nos queda

$$\text{loss}_r(w) = \text{loss}(w) + \beta \cdot \sum_i^n w_i^2 \quad (2.43)$$

donde β es el parámetro de regularización que controla la importancia de la penalización y w_i los pesos [28]. Este método tiende a producir soluciones más estables y reduce la complejidad de los modelos, tendiendo los coeficientes hacia valores más pequeños.

La **regularización $l1$** (o regresión Lasso) añade una penalización proporcional a la suma del valor absoluto de los pesos de cada capa. La función de pérdida regularizada $l1$ se define como $\sum_i^n |w_i|$ (donde n es el número de neuronas de esa capa). Sustituyendo en la ecuación 2.7.1, nos queda

$$\text{loss}_r(w) = \text{loss}(w) + \beta \cdot \sum_i^n |w_i| \quad (2.44)$$

Esta técnica puede generar un modelo más sencillo, donde algunos coeficientes se anulan, seleccionando las características más importantes de los datos. Sin embargo, produce soluciones menos estables que aplicando $l2$ [28].

2.7.2. Dropout

Otro método para evitar el sobreajuste de nuestro modelo es el *dropout*, que fue propuesto por Hinton et al. [37] como una forma de regularización para capas de redes neuronales completamente conectadas. El *dropout* consiste en que cada elemento de la salida de una capa se mantiene en cada iteración con una probabilidad p (tasa de *dropout*), de lo contrario se establece en 0, con una probabilidad $(1 - p)$. Según [28], este valor de p suele variar entre 0.1 y 0.5, siendo este último uno de los más típicos [83], aunque la red neuronal que se esté utilizando influye en el valor óptimo [28]. La elección de este parámetro es muy importante, ya que si tomamos un valor muy alto, puede llevarnos a *underfitting*.

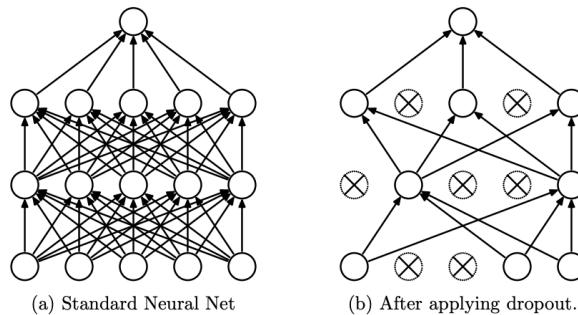


Figura 2.23: Esquema de la técnica dropout. Fuente [84]

Al eliminar una neurona, esta se elimina de la red con todas sus conexiones entrantes y salientes (ver Figura 2.23). La principal ventaja de esto es que la red no depende excesivamente de ninguna neurona, mejorando de esta forma la generalización del modelo. La eliminación se aplica de manera independiente a cada capa oculta y a cada iteración del entrenamiento.

Por lo tanto, aplicar *dropout* a una red neuronal equivale a seleccionar una submuestra de la red original. Como cada neurona puede estar activa o inactiva, hay un total de 2^n redes posibles, donde n es el número de neuronas que se pueden desactivar [83]. Puede interpretarse como entrenar una gran colección de redes neuronales diferentes y usar sus promedios para hacer predicciones.

Experimentos como los que se harán en el Capítulo 3 muestran como el *dropout* mejora la capacidad de generalización de la red, proporcionando un mejor rendimiento del modelo.

El uso del *dropout* en una red neuronal se implementa fácilmente en frameworks como Keras. Por ejemplo:

```

1 from tensorflow.keras.layers import Dropout
2
3 model = Sequential([
4     Dense(128, activation='relu'),
5     Dropout(0.5),
6     Dense(10, activation='softmax') ])

```

En este código, se aplica *dropout* con una tasa del 50% después de una capa densa con 128 unidades.

2.7.3. Parada temprana

Además de las técnicas de regularización $l1$ y $l2$ y el uso de *dropout* para prevenir el sobreajuste en redes neuronales, otra estrategia ampliamente utilizada es la parada temprana o *Early Stopping*.

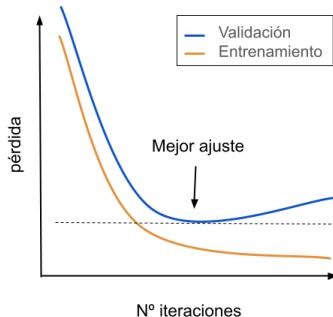


Figura 2.24: Curvas de error de entrenamiento y validación con *Early Stopping*.

El *Early Stopping* consiste en que el modelo deja de entrenar cuando el error en el conjunto de validación alcanza un mínimo y comienza a aumentar nuevamente. Esto se observa durante el proceso de entrenamiento, cuando el error de entrenamiento continúa disminuyendo mientras que el error de validación inicialmente disminuye, pero eventualmente comienza a incrementarse debido al sobreajuste [28]. La Figura 2.24 ilustra este comportamiento, donde la precisión del modelo en el conjunto de validación deja de mejorar después de cierto punto y comienza a empeorar.

Formalmente, si denotamos el error en el conjunto de entrenamiento después de la t -ésima época como $E_{\text{tr}}(t)$ y el error en el conjunto de validación como $E_{\text{val}}(t)$, *Early Stopping* interrumpe el entrenamiento cuando $E_{\text{val}}(t)$ deja de mejorar durante un número definido de épocas consecutivas.

Esta técnica no solo ayuda a prevenir el sobreajuste sino que también optimiza el uso de recursos al detener el entrenamiento cuando ya no se obtienen mejoras en el desempeño del modelo [87].

Para implementarlo en la práctica, *frameworks* como Keras ofrecen *callbacks* que facilitan este proceso. A continuación se presenta un ejemplo de código que utiliza *EarlyStopping* para detener el entrenamiento cuando no se observa ninguna mejora en el error de validación:

```

1 from tensorflow.keras.callbacks import EarlyStopping
2
3 early_stopping_cb = keras.callbacks.EarlyStopping(patience=10,
4                                                 restore_best_weights=True)
5 history = model.fit(X_train, y_train, epochs=100,
6                      validation_data=(X_valid, y_valid),
7                      callbacks=[early_stopping_cb])

```

En este ejemplo, *EarlyStopping* detiene el entrenamiento después de un número definido de épocas sin mejora (*patience* = 10), restaurando los pesos que lograron el mejor desempeño [28].

Así, *Early Stopping* se complementa eficazmente con las técnicas de regularización $l1$, $l2$ y *dropout*, proporcionando una capa adicional de protección contra el sobreajuste y mejorando la robustez del modelo.

El sobreajuste es uno de los principales desafíos en el entrenamiento de modelos de aprendizaje automático, incluyendo las redes neuronales. Técnicas como la regularización $l2$, el *dropout* y la parada temprana son estrategias efectivas para mitigar este problema. Es crucial seleccionar y combinar adecuadamente estas técnicas según las características del problema y los datos disponibles. A través de estos métodos, podemos mejorar la capacidad de generalización de nuestros modelos y lograr un rendimiento más robusto en datos no vistos [28, 74].

2.8. Evaluación del modelo.

Una vez discutidos los fundamentos del aprendizaje supervisado y no supervisado, y explorado una variedad de algoritmos de *machine learning*, profundizaremos ahora en la evaluación de los diferentes modelos. Nos centraremos en los métodos supervisados, tanto en regresión como en clasificación, ya que la evaluación y selección de modelos en aprendizaje no supervisado suele ser un proceso más cualitativo [68].

Para evaluar los modelos supervisados, primero se divide el conjunto de datos en un conjunto de entrenamiento y un conjunto de prueba utilizando la función `train_test_split`, que divide los datos de manera aleatoria. Algunos modelo como la CNN o los *autoencoders* también necesitan un conjunto de validación durante su entrenamiento para ir evaluando el rendimiento del modelo en cada iteración. Además, proporciona información muy valiosa para detectar si un modelo empieza a sobreaprender de los datos de entrenamiento. Los porcentajes de estos subconjuntos suelen encontrarse entre el 70 - 80 %, 10 - 20 % y 10 - 20 % para entrenamiento, prueba y validación respectivamente. El conjunto de entrenamiento sirve para entrenar la red neuronal. Por otro lado, el conjunto de validación sirve para ajustar hiperparámetros y evaluar el modelo con datos diferentes al entrenamiento, ayudando a evitar el sobreajuste. Por último, los datos de prueba sirven para medir definitivamente el rendimiento del modelo con nuevas muestras de datos que no han sido utilizadas durante del entrenamiento para evaluar el modelo [68].

El problema de esta técnica radica en que las distribuciones de los datos pueden no ser uniformes en los tres subconjuntos y que no sea una división muy óptima. Para evitar este inconveniente, se puede usar una técnica de evaluación del modelo más avanzada llamada **validación cruzada** (*cross-validation*), donde los datos se dividen en varios subconjuntos, y el modelo se entrena y valida múltiples veces, garantizando una evaluación más robusta del rendimiento. La variante más comúnmente utilizada es la validación cruzada *k-fold*. Aquí, el conjunto de datos se divide en *k* partes aproximadamente iguales, llamadas pliegues. Cada uno de los *k* pliegues se utiliza una vez como conjunto de prueba, mientras que los *k* - 1 pliegues restantes se combinan para formar el conjunto de entrenamiento. Este proceso se repite *k* veces, generando *k* modelos y *k* evaluaciones. La Figura 2.25 ilustra este procedimiento [68].

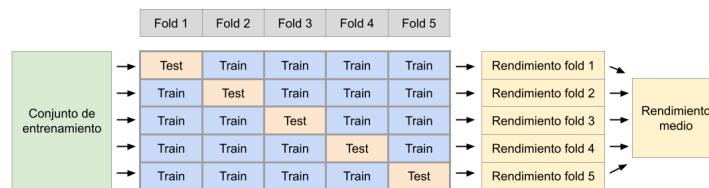


Figura 2.25: Validación cruzada *k-fold*.

Al final de la validación cruzada *k-fold*, se recopilan *k* valores de desempeño (por ejemplo, precisión en clasificación) que se pueden promediar para obtener una estimación más robusta del rendimiento del modelo. La validación cruzada no solo proporciona una evaluación más precisa del modelo, sino que también ayuda a detectar variaciones en el desempeño debido a diferentes particiones de datos, ofreciendo así una visión más completa de la capacidad del modelo para generalizar [68]. Aunque la validación cruzada ofrece bastantes beneficios, su uso es menos frecuente en redes neuronales debido a los altos costos computacionales asociados [28]. En su defecto, es más frecuente usar `train_test_split()`.

Una vez ya se ha entrenado el modelo, utilizamos el conjunto de prueba para medir el rendimiento del modelo. Para ello, se utilizan una serie de métricas que nos permiten entender el rendimiento y efectividad de un modelo. A continuación, se describen las principales.

2.8.1. Matriz de Confusión

La matriz de confusión es una herramienta que nos permite visualizar el rendimiento del modelo al mostrar los conteos de verdaderos positivos (TP), falsos positivos (FP), verdaderos negativos (TN) y falsos negativos (FN). Cada fila de la matriz representa las etiquetas reales mientras que cada columna representa las etiquetas predichas.

La matriz se define como:

	Predicción Positiva	Predicción Negativa
Real Positiva	TP	FN
Real Negativa	FP	TN

Para calcularla, utilizamos el conjunto de predicciones del modelo y las etiquetas reales. La matriz de confusión nos ofrece una visión clara de cómo se distribuyen los errores del modelo entre las diferentes clases.

2.8.2. Métricas

La matriz de confusión ofrece una gran cantidad de información, pero a veces te interesa una métrica más concreta para evaluar un modelo de clasificación, como la sensibilidad, la tasa de falsos negativos o su especificidad. A continuación vamos a definir y explicar las principales métricas a tener en cuenta según [25] junto con su fórmula.

La **sensibilidad** es la fracción de casos positivos que el modelo predice correctamente como positivos. También se conoce como *recall* o tasa de verdaderos positivos (TPR). Se calcula utilizando la fórmula:

$$\text{Sensibilidad} = \frac{TP}{TP + FN}$$

La **especificidad** es la fracción de casos negativos que el modelo predice correctamente como negativos. También se conoce como selectividad o tasa de verdaderos negativos (TNR). Se calcula utilizando la fórmula:

$$\text{Especificidad} = \frac{TN}{TN + FP}$$

La **tasa de falsos positivos** (FPR) es la fracción de casos negativos que el modelo predice incorrectamente como positivos. También se conoce como *fall-out* o probabilidad de alarma falsa. Se calcula utilizando la fórmula:

$$\text{FPR} = \frac{FP}{TN + FP}$$

La **tasa de falsos negativos** (FNR) es la fracción de casos positivos que el modelo predice incorrectamente como negativos. También se conoce como tasa de error tipo II o tasa de omisión. Se calcula utilizando la fórmula:

$$\text{FNR} = \frac{FN}{TP + FN}$$

El **valor predictivo positivo** (PPV) es la fracción de casos que el modelo predijo como positivos que realmente son positivos. También se conoce como precisión. Se calcula utilizando la fórmula:

$$\text{PPV} = \frac{TP}{TP + FP}$$

El **valor predictivo negativo** (NPV) es la fracción de casos que el modelo predijo como negativos que realmente son negativos. Se calcula utilizando la fórmula:

$$\text{NPV} = \frac{TN}{TN + FN}$$

El **accuracy** es la fracción de casos que el modelo predijo correctamente, ya sean positivos o negativos. Se calcula utilizando la fórmula:

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

Por último la **puntuación F1** es la media armónica del valor predictivo positivo y la sensibilidad. También se conoce como puntuación F, medida F o coeficiente de similitud de Dice. Se calcula utilizando la fórmula:

$$\text{Puntuación F1} = \frac{2TP}{2TP + FP + FN}$$

Un buen modelo de clasificación, evaluado mediante el F1-score, es aquel que presenta un F1-score cercano a 1. Esto indica que el modelo tiene un excelente equilibrio entre precisión y exhaustividad, identificando correctamente la mayoría de los verdaderos positivos y minimizando tanto los falsos positivos como los falsos negativos.

Estas métricas son aplicables a problemas de clasificación binaria, pero para problemas de clasificación multiclase, es necesario utilizar versiones extendidas de estas métricas. En estos casos, la sensibilidad y especificidad se pueden calcular para cada clase individualmente y luego promediarse para obtener una visión global del rendimiento del modelo. La precisión y el *recall* también deben evaluarse por clase y combinarse adecuadamente. Para evaluar la precisión en estos problemas, se considera la proporción de predicciones correctas con respecto al total de predicciones [102].

2.8.3. Curva ROC y AUC

La curva ROC (*Receiver Operating Characteristic*) es otra herramienta utilizada en los modelos de clasificación binaria. Consiste en una representación gráfica que muestra la relación entre la tasa de verdaderos positivos (TPR) y la tasa de falsos positivos (FPR) para diferentes umbrales de clasificación [28].

La curva ROC se construye variando el umbral de decisión, que es el punto en el que el modelo decide clasificar una predicción como positiva o negativa. Cada punto en la curva ROC corresponde a un umbral específico que separa las predicciones en positivas y negativas.

Un umbral bajo aumenta la TPR, lo que incrementa los verdaderos positivos pero también los falsos positivos, desplazando el punto hacia la esquina superior derecha de la gráfica. Por el contrario, un umbral alto disminuye la TPR, resultando en más verdaderos negativos pero también en más falsos negativos, moviendo el punto hacia la esquina inferior izquierda de la gráfica. La

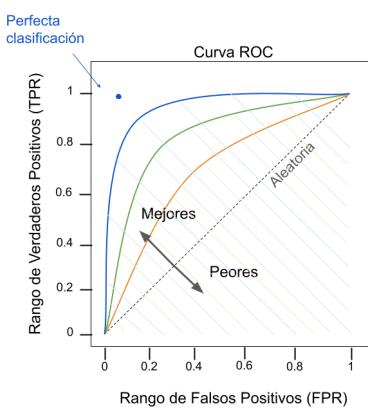


Figura 2.26: Interpretación de la curva ROC con el área bajo la curva (AUC).

curva ROC completa se genera trazando todos los puntos correspondientes a diferentes umbrales, desde el más bajo (donde todos los resultados se clasifican como positivos) hasta el más alto (donde todos se clasifican como negativos). De esta forma, la curva ROC proporciona una visualización del rendimiento del modelo en todo el rango de umbrales posibles [28].

Otro término que está relacionado es el AUC (*area under the curve*). El AUC es el área que se encuentra encerrada entre la curva ROC, la recta $y = 0$ y $x = 1$. Cuantifica la capacidad que tiene el modelo para distinguir entre las diferentes clases. Cuanto mayor valor, mejor rendimiento. En la Figura 2.26 se representa con rallas de intensidad baja.

La interpretación de estos términos consiste en que una curva ROC más cercana al vértice superior izquierdo, o un AUC cercano a 1, indica un mejor rendimiento del modelo [28].

2.8.4. Curva de Precisión-Recall

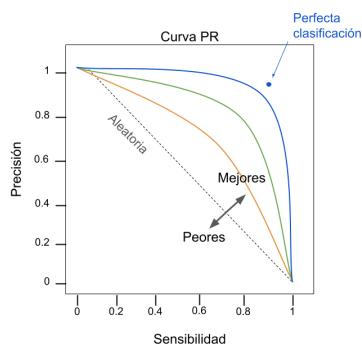


Figura 2.27: Curva de Precisión-Recall.

Por último, se estudiará la curva de Precisión-Recall (PR). Esta curva muestra la precisión frente a la sensibilidad para diferentes umbrales de decisión. Se identifica a partir de qué valor de sensibilidad comienza una disminución en la precisión, y viceversa. Idealmente, se busca una curva que se acerque lo más posible a la esquina superior derecha del gráfico, lo que representaría una combinación óptima de alta precisión y alto *recall*. Esta curva es especialmente útil para ajustar un umbral de decisión que permita obtener una precisión tan alta como se desee, pero a costa de la sensibilidad.

Por ejemplo, si se desea una precisión del 90 %, se puede aumentar el umbral hasta alcanzar este valor de precisión, aunque esto reducirá la sensibilidad. Este proceso se denomina *trade-off* entre precisión y sensibilidad.

Para calcular este umbral, se utilizan las puntuaciones de decisión obtenidas del clasificador. Primero, se obtienen estas puntuaciones usando la función `decision_function()` del clasificador. Luego, se emplea la función `precision_recall_curve()` de Scikit-Learn para calcular la precisión y la sensibilidad para todos los umbrales posibles. Finalmente, se selecciona el umbral que proporciona la precisión deseada. La Figura 2.28 ilustra cómo se comportan la precisión y la sensibilidad en función del umbral [28].

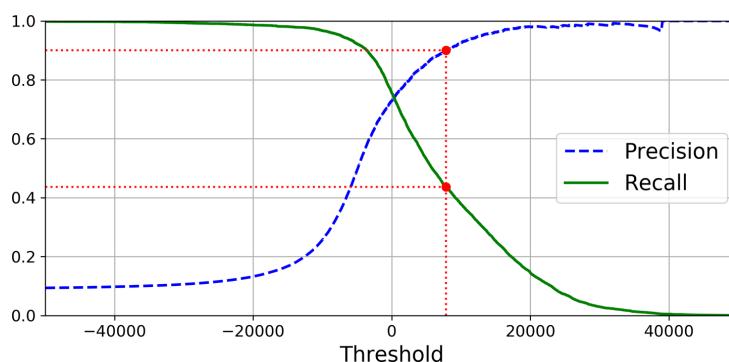


Figura 2.28: Precisión y Sensibilidad vs Umbral de decisión. Fuente [28]

2.9. Bibliotecas utilizadas en Python

Para nuestros experimentos, utilizaremos Python debido a su popularidad y versatilidad en el ámbito del aprendizaje automático y la inteligencia artificial. Python ofrece una amplia gama de bibliotecas especializadas que facilitan la creación, entrenamiento y evaluación de modelos, así como el análisis y visualización de datos. A continuación, se describen las principales bibliotecas y *frameworks* que emplearemos en este trabajo, destacando sus características y ventajas.

2.9.1. Principales frameworks.

Como las técnicas de aprendizaje profundo han ido ganando popularidad, muchas organizaciones académicas e industriales se han centrado en desarrollar marcos para facilitar la experimentación con redes neuronales profundas. En esta sección, ofrecemos una visión general de los marcos de trabajo más importantes que se pueden usar en Python, concluyendo con nuestra elección.

TensorFlow [32] es una biblioteca de código abierto desarrollada por el equipo de Google Brain para la computación numérica y el aprendizaje automático a gran escala. Diseñada para ser altamente flexible, TensorFlow soporta computación distribuida y permite la optimización de gráficos computacionales, lo que mejora significativamente la velocidad y el uso de memoria de las operaciones. En su núcleo, TensorFlow es similar a NumPy pero con soporte para GPU, lo que acelera considerablemente los cálculos. Además, incluye herramientas avanzadas como TensorBoard para la visualización de modelos y TensorFlow Extended para la producción de modelos de aprendizaje automático. Gracias a estas capacidades, TensorFlow se ha convertido en una herramienta esencial en la industria y la investigación, siendo utilizada en aplicaciones que van desde la clasificación de imágenes y el procesamiento de lenguaje natural hasta los sistemas de recomendación y la previsión de series temporales.

Keras [31] es una API de alto nivel para redes neuronales que ahora es parte integral de TensorFlow. Fue desarrollada por François Chollet y ganó popularidad rápidamente gracias a su simplicidad y diseño elegante. Inicialmente, Keras soportaba múltiples *backends*, pero desde la versión 2.4, funciona exclusivamente con TensorFlow [68]. Keras permite a los usuarios construir, entrenar y evaluar modelos de aprendizaje profundo de manera rápida y eficiente. Su facilidad de uso y extensa documentación la convierten en una herramienta valiosa tanto para la investigación como para la implementación de aplicaciones de inteligencia artificial.

PyTorch [26], desarrollado por el equipo de investigación de IA de Facebook, es una biblioteca de aprendizaje profundo que destaca por su enfoque en la computación dinámica, lo que permite una mayor flexibilidad en la creación de modelos complejos. A diferencia de TensorFlow, que utiliza gráficos computacionales estáticos, PyTorch permite que la topología de la red neuronal cambie durante la ejecución del programa [60]. Esto, junto con su capacidad de auto-diferenciación en modo inverso⁷, hace que PyTorch sea popular entre los investigadores y desarrolladores. Su facilidad de uso y robusta comunidad de apoyo han llevado a su adopción por parte de importantes organizaciones como Facebook, Twitter y NVIDIA.

Para escoger con cuál de estas librerías se realizará la parte práctica de este trabajo, vamos a utilizar, además de las características previamente vistas, los resultados de [60]. En él se hace un estudio de eficiencia, convergencia, tiempo de entrenamiento y uso de memoria de los diferentes *frameworks* con varios conjuntos de datos. Entre sus resultados podemos observar como Keras

⁷Técnica en la que PyTorch calcula automáticamente las derivadas de las funciones de pérdida con respecto a los parámetros del modelo.

destaca por encima de las demás en el entorno de la CPU. No solo logra el mejor *accuracy* en los tres datasets (MNIST, CIFAR-10, CIFAR-100), sino que además también tiene los tiempos de ejecución más bajos y una de las mejores tasas de convergencia. En cuanto al entorno de la GPU, las tres librerías obtienen unos resultados semejantes. En conclusión, podemos afirmar que estos resultados junto con su facilidad de uso, accesibilidad y documentación bien estructurada, han sido determinantes para optar por usar Keras (junto con TensorFlow) en vez de PyTorch o solo TensorFlow en nuestros estudios posteriores. Aakash Nain resume perfectamente las ventajas de Keras [1] al señalar que:

“Keras is that sweet spot where you get flexibility for research and consistency for deployment. Keras is to Deep Learning what Ubuntu is to Operating Systems.”

De manera similar, Matthew Carrigan destaca la intuitividad y facilidad de uso de Keras [64], afirmando:

“The best thing you can say about any software library is that the abstractions it chooses feel completely natural, such that there is zero friction between thinking about what you want to do and thinking about how you want to code it. That’s exactly what you get with Keras.”

2.9.2. Librerías y herramientas esenciales.

De forma complementaria, también es importante conocer y utilizar diversas librerías y herramientas esenciales que facilitan el desarrollo y análisis de los modelos de Keras. Estas incluyen herramientas para la manipulación, visualización y análisis de datos.

Scikit-Learn [80] es una librería de código abierto con herramientas simples y eficientes para el análisis predictivo de datos. Contiene varios algoritmos de aprendizaje automático, desde clasificación y regresión hasta *clustering* y reducción de dimensionalidad, con la documentación completa sobre cada algoritmo. Está construida sobre otras librerías que veremos más adelante como Numpy, SciPy y matplotlib. Aunque no se aprovecharán todas estas funcionalidades de scikit-learn, si que se va a utilizar una de sus funciones más populares, `train_test_split()` [81]. Esta función divide el conjunto de datos en dos subconjuntos de forma aleatoria, manteniendo la correspondencia en caso de que el conjunto de datos contenga dos o más partes. Usualmente, a estos subconjuntos se les llama conjunto de prueba y conjunto de entrenamiento, cuyo tamaño se indica con un valor entre 0 y 1 (`test_size`). Además, también se suele asignar una semilla a esa división para que cada vez que se quieran reproducir los experimentos, pueda usarse la misma partición. Esa semilla es un número natural que se introduce como parámetro de entrada en la variable `random_state`. Veamos un ejemplo de como utilizar esta función.

```

1 # Ejemplo de código en Python
2 from sklearn.model_selection import train_test_split
3
4 X_train, X_test, y_train, y_test = train_test_split(data, labels,
5                                                 test_size=0.25, random_state=42)

```

Las variables `X_train`, `X_test` y `compañía` son numpy arrays. **NumPy** [20] es el paquete fundamental de Python para la computación científica. Es una biblioteca general de estructuras de datos, álgebra lineal y manipulación de matrices para Python, cuya sintaxis y manejo de estructuras de datos y matrices es comparable al de MATLAB [12]. NumPy es la base de todas las

estructuras de datos utilizadas en el entrenamiento de todas las redes neuronales. Se utilizarán estas estructuras de datos para almacenar los datos y entrenar las redes neuronales con ellas. Aunque también se pueden utilizar tensores [18], se ha decidido utilizar numpy arrays por su alta eficiencia operacional y por su uso en la industria.

Otro paquete que se va a utilizar durante los experimentos y que Scikit-Learn utiliza es **matplotlib** [63]. Es la principal biblioteca de gráficos científicos en Python y proporciona funciones para crear visualizaciones de calidad como gráficos de barras, histogramas, gráficos de dispersión, etc. Se utilizará este paquete para representar gráficamente los datos de cada dataset para poder obtener bastante información con un simple vistazo.

Capítulo 3

Clasificación de Malware

Hoy en día, uno de los principales retos que enfrenta el software antimalware es la enorme cantidad de datos y archivos que se requieren evaluar en busca de posibles amenazas maliciosas. Una de las razones principales de este volumen tan elevado de archivos diferentes es que los creadores de malware introducen variaciones en los componentes maliciosos para evadir la detección. Esto implica que los archivos maliciosos pertenecientes a la misma “familia” de malware (con patrones de comportamiento similares), se modifican constantemente utilizando diversas tácticas, lo que hace que parezcan ser múltiples archivos distintos [65].

Para poder analizar y clasificar eficazmente estas cantidades masivas de archivos, es necesario agruparlos e identificar sus respectivas familias. Además, estos criterios de agrupación pueden aplicarse a nuevos archivos para detectarlos como maliciosos y asociarlos a una familia específica.

En este capítulo vamos a enfrentar este problema, escogiendo una de las bases de datos disponibles en [77] para poder clasificar distintos tipos de ciberataques. Como el objetivo principal de este trabajo es el estudio y puesta en práctica de diferentes algoritmos de aprendizaje automático, se ha decidido tomar la base de datos *Microsoft Malware Classification Challenge* [65]. La principal razón de esta decisión ha sido que con este *dataset* tenemos a nuestra disposición dos algoritmos documentados diferentes de *machine learning*.

3.1. Microsoft Malware Classification Challenge

El conjunto de datos utilizado en este estudio proviene del *Microsoft Malware Classification Challenge (BIG 2015)*, una competición dirigida a la comunidad científica con el objetivo de promover el desarrollo de técnicas efectivas para agrupar diferentes variantes de malware. Se decidió escoger este dataset porque el objetivo que tengo en este trabajo es el de aprender y desarrollar diferentes métodos de aprendizaje automático y este dataset nos permite utilizar tanto una CNN como un *autoencoder* según [77].

Se puede descargar desde su página web [65]. Tiene un tamaño de 0.5 TB sin comprimir. Para poder manipularla en mi ordenador, tuve que, primero, descargarme la carpeta comprimida (7z) con todo el dataset. Después, subirla al servidor Simba de la facultad de informática y finalmente, usando el comando `7zz x file_name.7z`, descomprimirla.

Este dataset contiene 5 archivos:

- *dataSample.7z* - Carpeta comprida(7z) con una muestra de los datos disponibles.
 - *train.7z* - Carpeta comprida(7z) con los datos para el conjunto de entrenamiento.
 - *trainLabels.csv* - Archivo csv con las etiquetas asociadas a cada archivo de train.
 - *test.7z* - Carpeta comprida 7z con los datos sin procesar para el conjunto de prueba.
 - *sampleSubmission.csv* - Archivo csv con el formato de envío válido de las soluciones.

Para nuestro estudio, nos enfocaremos exclusivamente en el conjunto de datos de entrenamiento, que consta de los archivos *train.7z* y *trainLabels.csv*. Los archivos *test.7z* y *sampleSubmission.csv* están destinados específicamente para la competición. Nosotros no los utilizaremos debido a que son programas de malware sin etiquetar y para este problema de clasificación, es necesario conocerlas. Además, la carpeta *dataSample.7z* proporciona dos programas que se encuentran también en la carpeta *train.7z*, por lo que tampoco la utilizaremos.

Cada programa malicioso tiene un identificador, un valor hash de 20 caracteres que identifica de forma única el archivo, y una etiqueta de clase, que es un número entero que representa una de las 9 familias de malware al que puede pertenecer. Por ejemplo, el programa *0ACdbR5M3ZhBJajygTuf* tiene como etiqueta el valor 7. Esta información se puede consultar en el archivo *trainLabels.csv*. Cada programa tiene dos archivos, uno asm con el código extraído por la herramienta de desensamblado IDA y otro bytes¹ con la representación hexadecimal del contenido binario del programa pero sin los encabezados ejecutables (para garantizar esterilidad). Para nuestro estudio vamos a utilizar únicamente este ultimo archivo.

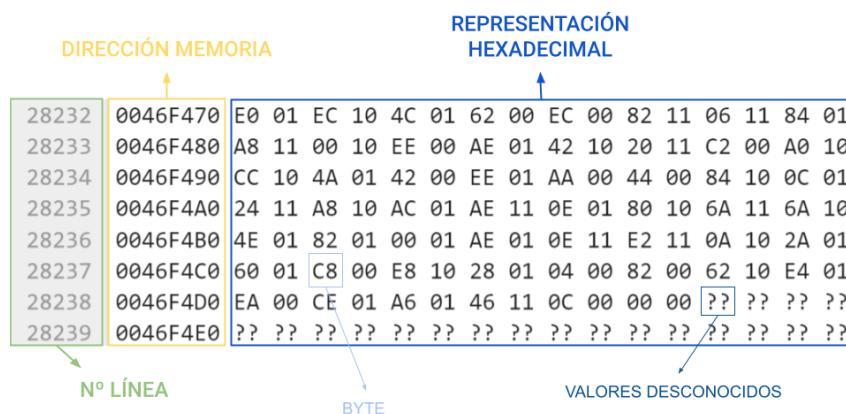


Figura 3.1: Explicación del contenido de “0ACDbR5M3ZhBJajygTuf.bytes”.

Como aparece en la figura 3.1, los ocho primeros caracteres son direcciones de memoria, seguido de la representación hexadecimal del contenido binario del programa, que contiene 16 bytes (cada uno dos caracteres). A veces nos podemos encontrar con “?” en el lugar de un byte. Este símbolo se utiliza en estos archivos para representar que se desconoce su información porque su memoria no se puede leer [13].

¹Realmente no es un archivo bytes, sino un fichero de texto con caracteres.

3.1.1. Distribución del dataset

Hay un total de 21.741 programas de malware, pero solo 10.868 de ellos tienen etiquetas. Estos programas pertenecen a una de estas 9 familias de malware: Ramnit, Lollipop, Kelihos_ver3, Vundo, Simda, Tracur, Kelihos_ver1, Obfuscator y Gatak. Según [40], podemos definirlas como:

1. **Ramnit** es un malware tipo gusano que infecta archivos ejecutables de Windows, archivos de Microsoft Office y archivos HTML. Cuando se infectan, el ordenador pasa a formar parte de una red de *bots* controladas por un nodo central de forma remota. Este malware puede robar información y propagarse a través de conexiones de red y unidades extraíbles.

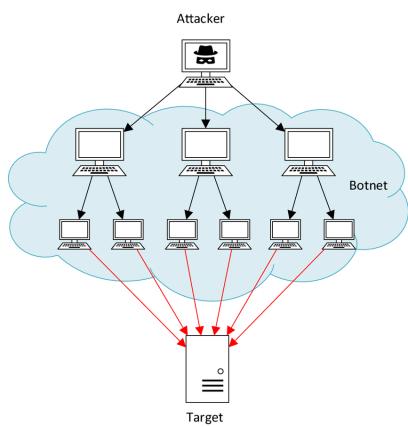


Figura 3.2: Estructura de un botnet. Imagen sacada de [69].

2. **Lollipop** es un tipo de programa *adware*² que muestra anuncios no deseados en los navegadores web. También puede redirigir los resultados de búsqueda a recursos web ilegítimos, descargar aplicaciones maliciosas y robar la información del ordenador monitorizando sus actividades web. Este *adware* se puede descargar desde el sitio web del programa o empaquetarse con algunos programas de terceros.

3. **Simda** es un troyano *backdoor*³ que infecta ordenadores descargando y ejecutando archivos arbitrarios que pueden incluir malware adicional. Los ordenadores infectados pasar a ser parte de una *botnet*, lo que les permite cometer acciones criminales como robo de contraseñas, credenciales bancarias o descargar otros tipos de malware.

4. **Vundo** es otro troyano conocido por causar publicidad emergente para programas de antivirus falsos. A menudo se distribuye como un archivo DLL(Dynamic Link Library)⁴ y se instala en el ordenador como un Objeto Auxiliar del Navegador (BHO) sin su consentimiento. Además, utiliza técnicas

avanzadas para evitar su detección y eliminación.

5. **Kelihos_ver3** es un troyano tipo *backdoor* que distribuye correos electrónicos que pueden contener enlaces falsos a instaladores de malware. Consta de tres tipos de *bots* [47]: controladores (operados por los dueños y donde se crean las instrucciones), enruteadores (redistribuyen las instrucciones a otros *bots*) y trabajadores (ejecutan las instrucciones).
6. **Tracur** es un descargador troyano que agrega el proceso 'explorer.exe' a la lista de excepciones del *Firewall* de Windows para disminuir deliberadamente la seguridad del sistema y permitir la comunicación no autorizada a través del *firewall*. Además, esta familia también te puede redirigir a enlaces maliciosos para descargar e instalar otros tipos de malware.
7. **Kelihos_ver1** es una versión más antigua del troyano Kelihos_ver3, pero con las mismas funcionalidades.

²Es una variedad de malware que muestra anuncios no deseados a los usuarios, típicamente como ventanas emergentes o *banners*.

³Un *backdoor* permite que una entidad no autorizada tome el control completo del sistema de una víctima sin su consentimiento.

⁴Una parte del programa que se ejecuta cuando una aplicación se lo pide. Se suele guardar en un directorio del sistema.

8. **Obfuscator.ACY** es un tipo de malware sofisticado que oculta su propósito y podría sobrepasar las capas de seguridad del software. Se puede propagar mediante archivos adjuntos de correo electrónico, anuncios web y descargas de archivos.
9. **Gatak** es un troyano que abre una puerta trasera en el ordenador. Se propaga a través de sitios web falsos que ofrecen claves de licencias de productos. Una vez infectado el sistema, Gatak recopila información del ordenador.

Como ya mencionamos antes, solo hay 10.868 programas con etiquetas, luego vamos a hacer el análisis descriptivo de los datos solo con estos archivos. De estos programas, solo son válidos 10.860 porque en los 8 archivos restantes⁵ (pertenecientes a la familia Ramnit), todo sus bytes son “??”, es decir, información desconocida. Con estos datos, vamos a ver gráficamente como se distribuyen en las 9 clases de malware (Figura 3.3).

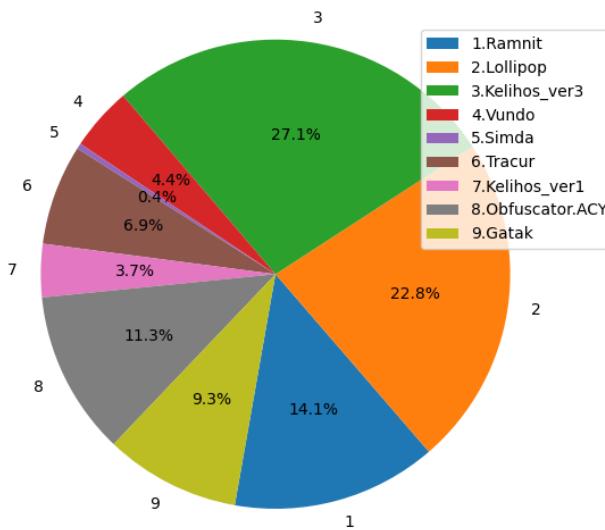


Figura 3.3: Distribución del BIG 2015 *training* dataset.

Analizando la Figura 3.3, podemos observar como la distribución entre las clases no es uniforme. Mientras que de la clase Simbda hay 42 muestras, de la clase Kelihos_ ver3 hay 2.942, es decir, 70 veces más de muestras. En [46] deciden prescindir de esta clase, pero nosotros hemos decidido hacer el análisis con las 9 clases.

A la hora de crear nuestros modelos, hemos dividido el conjunto de datos aleatoriamente usando la función `train_test_split()` en grupos del 75 %, 15 % y 10 % para entrenamiento, test y validación respectivamente. La tabla 3.1 muestra como quedarían distribuidas las clases en los diferentes grupos.

	Ramnit	Lollipop	Kelihos3	Vundo	Simda	Tracur	Kelihos1	Obfus	Gatak
Total	1533	2478	2942	475	42	751	398	1228	1013
Train	1177	1835	2228	337	26	543	306	925	768
Test	223	394	436	75	9	124	42	177	149
Valid	133	249	278	63	7	84	50	126	96

Cuadro 3.1: Distribución de los tipos de malware en los conjuntos de datos

⁵Los identificadores de estos archivos son 58kxhXouHzFd4g3rmInB, 6tfw0xSL2FNHOCJBdlaA, a9oIzfw03ED4iTBCt52Y, cf4nzsoCmudt1kwleOTI, d0iHC6ANYGon7myPFzBe, da3XhOZzQEbKVtLgMYWv, fRLS3aKkijp4GH0Ds6Pv, IidxQvXrlBkWPZAfcqKT.

Para abordar este problema de clasificación, vamos a realizar dos modelos de aprendizaje automático diferentes para luego comparar sus resultados. El primer método que vamos a utilizar es una Red Neuronal Convolucional (CNN). El segundo será entrenar un *autoencoder* junto con una capa de clasificación, primero obteniendo una representación comprimida de los datos y después clasificando esta representación con una red neuronal profunda.

3.2. Red Neuronal Convolucional

Para abordar este problema utilizando como modelo una CNN, solo podemos utilizar los datos etiquetados ya que este método es un método supervisado. Como ya vimos en la sección 2.4.4, para entrenar estas redes neuronales es necesario tener los datos en forma matricial. Uno de los principales motivos para convertir el malware en imagen es porque los creadores de malware suelen modificar sus implementaciones para producir nuevo malware [71], pero si lo representamos de forma matricial, estos pequeños cambios pueden ser detectados fácilmente [43].

3.2.1. Visualizar el malware como imagen

Para visualizar los archivos .bytes como imagen en escala de grises, cada byte debe ser interpretado como un píxel en la imagen. Inspirado en [72], para pasar de código hexadecimal a imagen primero pasamos cada byte a su número decimal correspondiente que se encuentra en el rango [0,255]⁶. Como vimos en el apartado anterior, hay algunos bytes que son “??” lo que significa que se desconoce su información. Para solucionar este problema con los datos, en el apéndice B de [28], se plantea eliminar estos caracteres y tratar el resto de bytes. Otra solución la proponen Narayanan et al. [70], que es sustituir estos bytes por el valor 255 (color blanco). Después de probar con ambas propuestas y además la de cambiando el “??” por el valor 0, finalmente hemos decidido sustituirlo por 0 (color negro) en base a los resultados obtenidos.

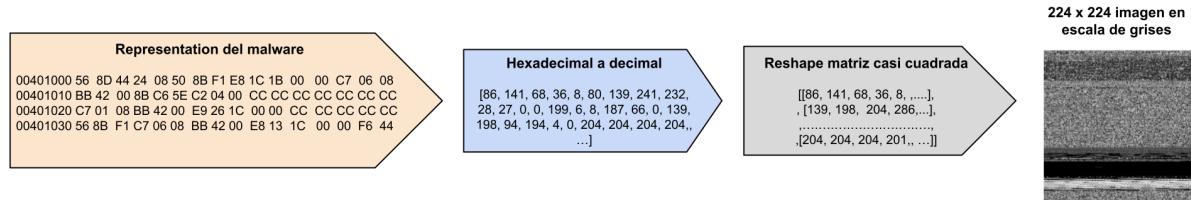


Figura 3.4: Proceso de visualización del malware. Adaptación de [72]

Después de tener todos los bytes agrupados en un vector con sus elementos en formato decimal, hacemos un reshape a una matriz 2D de forma que se consiga una matriz lo más cuadrada posible. Como las dimensiones de cada archivo son diferentes, las dimensiones después de hacer el reshape también lo serán, luego tenemos que fijar un tamaño fijo para poder entrenar nuestra CNN [53]. Para ello, siguiendo el ejemplo de [43], decidimos escoger como tamaño 224×224 . Para obtener estas dimensiones, Simonyan et al. [82] deciden recortar aleatoriamente un cuadrado de la imagen de tamaño 224×224 , pero nosotros hemos decidido usar interpolación bilineal en base al artículo [36]. En la figura 3.5, se puede observar cuales son los patrones que sigue cada tipo de malware en su forma matricial.

⁶Cada valor en este intervalo tiene un color asociado donde 0 es el negro y 255 el color blanco.

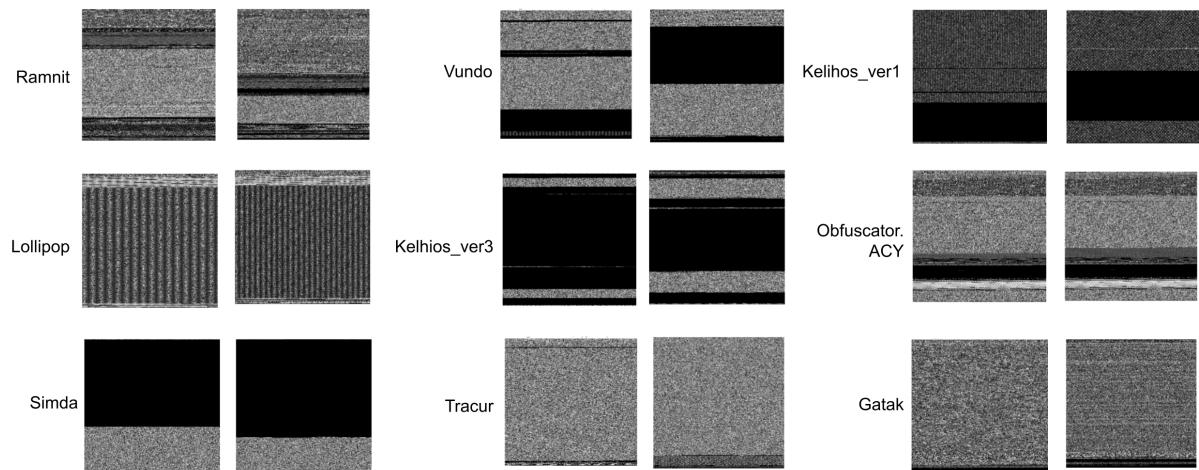


Figura 3.5: Visualización familias malware.

3.2.2. Visualización del modelo

Una vez ya hemos explorado y preparado los datos, el siguiente paso es crear nuestra red neuronal convolucional. Para ello hemos seguido el artículo [43], en el que se crea una M-CNN (malware CNN) con múltiples capas. Vemos su arquitectura en la Figura 3.6.

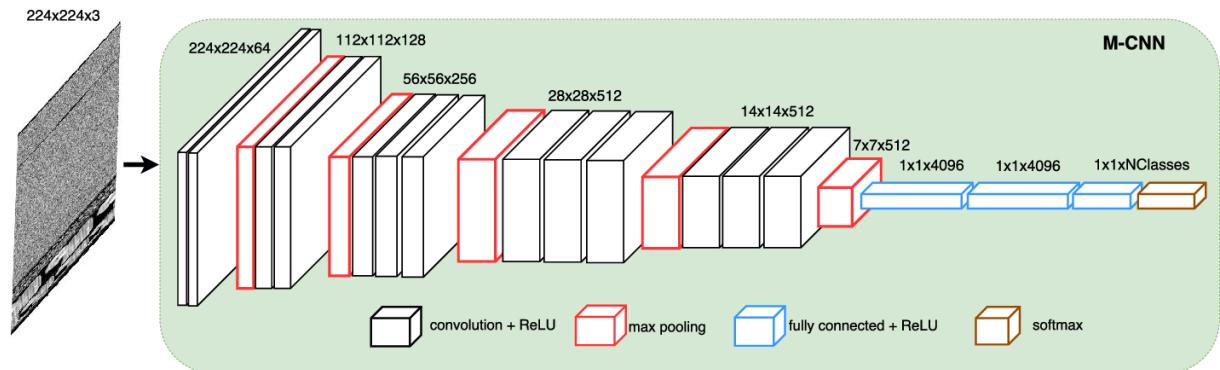


Figura 3.6: Arquitectura de la CNN. Fuente [43].

En las capas iniciales, se aplican filtros de tamaño 3×3 , que permiten extraer características locales importantes de la imagen de entrada. Las capas convolucionales aprenden a detectar bordes, texturas y otros patrones básicos al principio, y conforme se agregan más capas, las características detectadas se vuelven más abstractas y complejas [28]. Después de cada operación de convolución, se aplica la función de activación ReLU para introducir no linealidad en el modelo.

Las capas de *max pooling* reducen la dimensionalidad de los mapas de características seleccionando el valor máximo en subregiones de 2×2 , lo que no solo reduce la carga computacional sino que también ayuda a hacer las características más robustas a pequeñas variaciones y traslaciones en la imagen de entrada [28].

La secuencia de capas convolucionales y de *pooling* se repite varias veces hasta obtener un conjunto final de características que es una matriz de 7×7 con 512 mapas de características. Esta salida se aplana para convertirla en un vector unidimensional, que luego pasa a través de

varias capas completamente conectadas (*fully connected layers*). Finalmente, la capa de salida utiliza una función de activación softmax para generar las predicciones finales del modelo.

Una vez definida la arquitectura de la red, procedemos a compilar y entrenar el modelo. Para la compilación del modelo, hemos seguido el artículo [43] para la elección de las funciones de pérdida y de optimización, con los parámetros adecuados. Utilizamos el optimizador SGD con un *learning rate* inicial de 0.001. Este valor se reduce en un factor de 10 cada 20 *epochs*. Además, fijamos el *momentum* en 0.9 y el *weight decay* en 0.0005. La función de pérdida utilizada es *categorical_crossentropy* ya que nuestro problema involucra múltiples clases de etiquetas [19].

El entrenamiento se realizó con un *batch size* de 8 y se ejecutó durante 25 *epochs*. Utilizamos *callbacks* para ajustar dinámicamente el *learning rate*, detener el entrenamiento temprano si no se observan mejoras en la pérdida de validación (*EarlyStopping*), y registrar los resultados de las métricas utilizadas durante el entrenamiento. Además, se baraja el conjunto de datos de entrenamiento antes de cada *epoch*.

Después de las 25 *epochs*, este modelo obtiene un *accuracy* en el conjunto de validación del 0.959, mientras que en el conjunto de entrenamiento obtiene 1,0. Por otro lado, en la función de pérdida se obtiene un 0.331 en el conjunto de validación y un 0.0005 en el conjunto de entrenamiento. En la figura 3.7 podemos visualizar la evolución de ambas métricas a lo largo de las 25 iteraciones. En ellas se puede observar como durante las primeras épocas el modelo esta generalizando bastante bien debido a la similitud en los resultados obtenido. Sin embargo a partir de la época 5 se puede observar como tanto el *accuracy* como el *loss* de validación se estabiliza en el 0.94 y 0.3 respectivamente mientras los de entrenamiento siguen optimizándose. Estas diferencias significativas en los resultados de ambos conjuntos de datos, unido al análisis gráfico, nos sugieren que se puede estar produciendo un sobreajuste de los datos de aprendizaje, lo que evita la generalización.

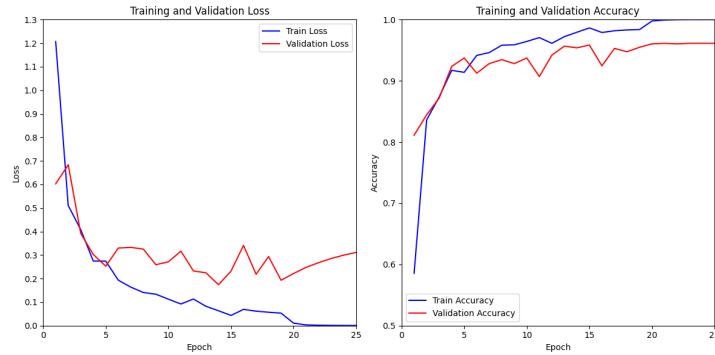


Figura 3.7: Evolución del modelo en términos de *accuracy* y *loss* siguiendo los pasos del artículo [43].

3.2.3. Aportaciones al modelo

Dado los resultados obtenidos previamente, intentaremos mejorar nuestro modelo. Primero de todo, trataremos de aplicar algunas de las técnicas mencionadas en el capítulo 2 para lograr un mejor ajuste de los datos al modelo y obtener una mejor generalización. A continuación, se realizarán diferentes experimentos para comprobar que los hiperparámetros indicados en el artículo [43] son los que mejores resultados obtiene.

Evitar el sobreajuste del modelo

Primero, vamos a añadir capas de *dropout* después de cada capa densa, como se sugiere en [37, 21]. Probaremos con diferentes tasas de *dropout* (0.25, 0.4, 0.5 y 0.6) para determinar cuál proporciona los mejores resultados.

El primer experimento va a consistir en añadir estas capas de *dropout* con una tasa de 0.25. Los resultados obtenidos se muestran en la Figura 3.8. Observamos como el modelo generaliza bien hasta la época 7, pero a partir de esa época, empieza a sobreajustar los datos de entrenamiento. Al final de entrenamiento alcanza 1.0 y 0.962 en entrenamiento y validación respectivamente para el *accuracy*, y un valor en la función de pérdida de 0.0008 y 0.2479 en los datos de entrenamiento y validación respectivamente. Aunque hemos mejorado en términos de generalización y resultados finales después de 25 épocas, incrementaremos la tasa de *dropout* a 0.4 para observar los efectos.

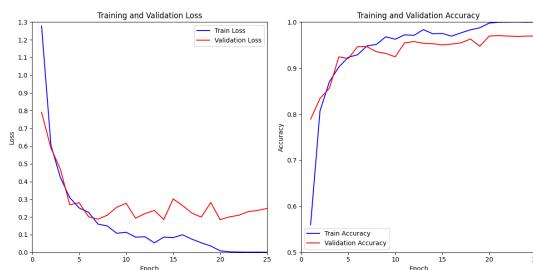


Figura 3.8: Evolución del modelo en términos de *accuracy* y *loss* con *dropout* 0.25.

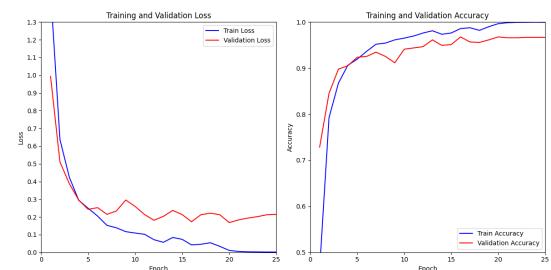


Figura 3.9: Evolución del modelo en términos de *accuracy* y *loss* con *dropout* 0.4.

Con una tasa de *dropout* de 0.4, como se observa en la Figura 3.9, los resultados obtenidos son similares, lo cual nos lleva a probar con una tasa mayor. Al aplicar una tasa de 0.5, observamos en la Figura 3.10 que el *accuracy* aumenta progresivamente en ambos conjuntos, alcanzando 0.9985 en entrenamiento y 0.9714 en validación. La función de pérdida, por otra parte, muestra una estabilización hasta la época 19 (0.1708), antes de empezar a empeorar, llegando a 0.2614 en la época 25, mientras que la pérdida en entrenamiento sigue descendiendo hasta 0.004.

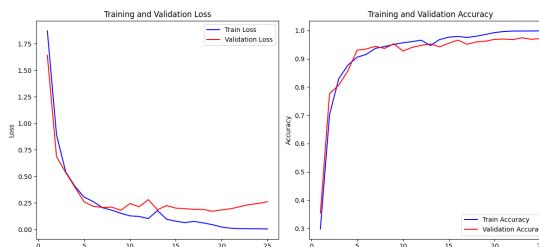


Figura 3.10: Evolución del modelo en términos de *accuracy* y *loss* con *dropout* 0.5.

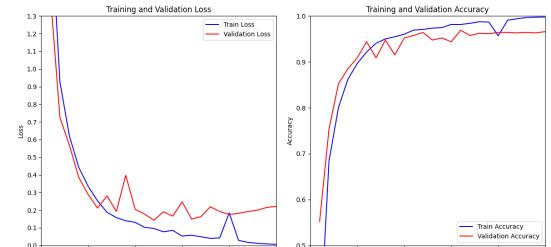


Figura 3.11: Evolución del modelo en términos de *accuracy* y *loss* con *dropout* 0.55.

Para verificar que esta tasa es la más óptima, probamos con una tasa de 0.55 y, como se muestra en la Figura 3.11, obtenemos resultados casi idénticos al caso con una tasa de 0.4. Concluimos que los mejores resultados se obtienen con una tasa de *dropout* del 0.5.

Para evitar el sobreaprendizaje de los datos de entrenamiento, añadimos el callback *EarlyStopping*. Aplicamos esta función a la métrica por defecto *val_loss*, con una *patience* de 4 épocas. Nuestro entrenamiento se detiene en la época 20 después de alcanzar su mejor valor de pérdida en el conjunto de validación con una pérdida de 0.1716 y un *accuracy* de 0.965, comparado con una pérdida de 0.026 y un *accuracy* de 0.991 en el entrenamiento respectivamente. Aunque este

modelo no alcanza el mejor *accuracy*, es el que mejor generaliza los datos. Véase la Figura 3.12.

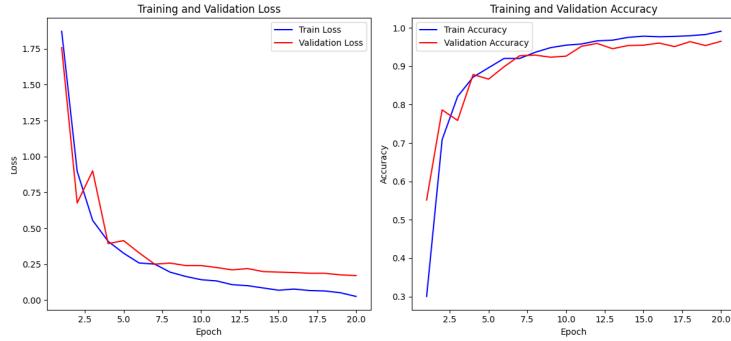


Figura 3.12: Evolución del modelo en términos de *accuracy* y *loss* con *dropout* 0.5 y *EarlyStopping*.

Una vez se ha obtenido el mejor modelo, se van a realizar más experimentos para verificar que efectivamente el resto de hiperparámetros que se han propuesto en el artículo [43] son los que mejor resultados obtiene. Para ello, se ha probado con un diferente *step_decay* (reducir el *learning rate* en 10 unidades cada 10 épocas, 20 épocas o nunca) y con diferentes *batch sizes* (8, 16 y 32). Para todos estos experimentos, se utilizó la arquitectura explicada anteriormente con una tasa de *dropout* del 0.5 y sin *EarlyStopping*.

Experimentos con diferentes *step decay*

En esta sección, se analizan tres experimentos realizados con diferentes configuraciones de *step decay* en el entrenamiento de la CNN. Los experimentos variaron en la frecuencia de reducción del *learning rate*: cada 20 épocas, cada 10 épocas, y sin reducción. A continuación, se presentan los resultados obtenidos.

En el **primer experimento**, se aplicó una reducción del *learning rate* **cada 20 épocas**, como se menciona en [43]. Los resultados ya han sido estudiados anteriormente ya que son los mismos hiperparámetros que se aconsejaban usar. La reducción del *learning rate* cada 20 épocas permite al modelo ajustar su aprendizaje de forma gradual. En las primeras 20 épocas, el modelo se beneficia de un *learning rate* más alto que permite realizar ajustes significativos a los pesos. Después de la reducción, el *learning rate* más bajo ayuda a afinar estos ajustes, lo que puede mejorar la convergencia y la estabilidad del modelo al evitar grandes oscilaciones en la pérdida. Este enfoque balancea bien la exploración y la explotación durante el entrenamiento.

El **segundo experimento** redujo el *learning rate* **cada 10 épocas** (ver Figura 3.13). La pérdida en el entrenamiento disminuyó de 1.809 a 0.014 y la precisión aumentó de 33.0 % a 99.6 %. En el conjunto de validación, la pérdida disminuyó de 1.121 a 0.200 y la precisión mejoró de 61.9 % a 96.7 %. Como se observa en la Figura 3.13, hasta la época 10 tanto la pérdida como la precisión del conjunto de validación presentan resultados similares al conjunto de entrenamiento, pero a partir de esta época, se estabilizan y dejan de mejorar significativamente.

La reducción del *learning rate* cada 10 épocas puede ser demasiado frecuente, lo que podría limitar la capacidad del modelo para realizar grandes ajustes necesarios al principio del entrenamiento. Esto puede conducir a una estabilización temprana, limitando la mejora de la precisión después de una cierta cantidad de épocas. La reducción más frecuente también puede hacer que el modelo se “comode” demasiado pronto, antes de haber explorado completamente el espacio

de pesos, lo que podría explicar la estabilización observada a partir de la época 10.

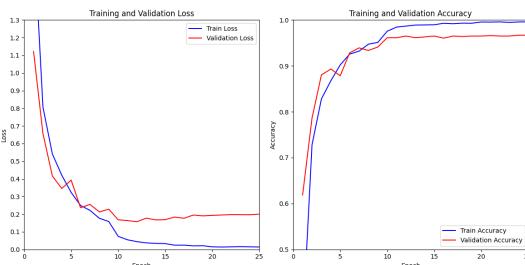


Figura 3.13: Resultados para *step decay* cada 10 épocas.

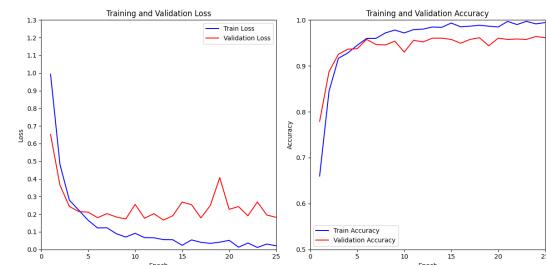


Figura 3.14: Resultados sin reducción de *step decay*.

Finalmente, se realizó un **experimento sin reducción del *learning rate*** (ver Figura 3.14). Los resultados fueron inferiores en comparación con los experimentos anteriores. La pérdida en el entrenamiento disminuyó de 1.916 a 0.059 y la precisión aumentó de 26.7 % a 98.3 %. En el conjunto de validación, la pérdida se redujo de 1.918 a 0.184 y la precisión mejoró de 43.6 % a 95.9 %. Aunque los resultados no son malos, fueron peores en comparación con los otros modelos que aplicaron *step decay*.

No reducir el *learning rate* en ningún momento puede llevar a que el modelo realice ajustes demasiado grandes durante todo el entrenamiento, lo que puede resultar en una convergencia menos precisa. El *learning rate* constante puede ser demasiado agresivo en las últimas etapas del entrenamiento, evitando que el modelo afine los pesos y, por lo tanto, logre una precisión óptima. Esto explica por qué los resultados fueron inferiores, ya que el modelo no se beneficia del refinamiento progresivo que proporciona la reducción del *learning rate*.

En conclusión, el experimento con *step decay* cada 20 épocas obtuvo los mejores resultados en términos de *accuracy* y *loss*, además de mostrar una mejor generalización en los datos de validación.

Experimentos con diferentes *batch sizes*

Para evaluar el impacto de diferentes *batch sizes*, se realizaron experimentos con *batch sizes* de 8, 16 y 32. Primero, se realizó un experimento con `batch_size = 8`, cuyos resultados ya se han discutido en el apartado anterior y en el capítulo 3. A continuación, se presentan los resultados obtenidos con `batch_size = 16` y `batch_size = 32`. Los datos de cada experimento se recogen en las Figuras 3.15 y 3.16 respectivamente.

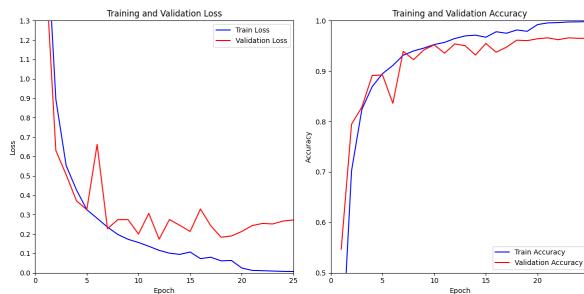


Figura 3.15: Entrenamiento y validación del modelo de clasificación usando un `batch_size = 16`.

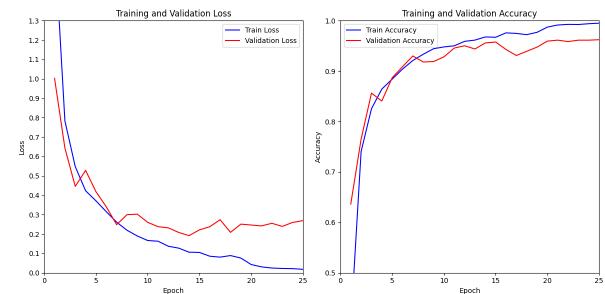


Figura 3.16: Entrenamiento y validación del modelo de clasificación usando un `batch_size = 32`.

El aumento del *batch size* a 16 mejoró ligeramente la estabilidad del entrenamiento. La pérdida de entrenamiento comenzó en 1.809 y disminuyó a 0.014, mientras que la precisión aumentó de 33.0 % a 99.6 %. En validación, la pérdida se redujo de 1.121 a 0.200, y la precisión mejoró de 61.9 % a 96.7 %.

Con un *batch size* de 32, la pérdida de entrenamiento comenzó en 1.809 y disminuyó a 0.014, mientras que la precisión aumentó de 33.0 % a 99.6 %. En validación, la pérdida se redujo de 1.121 a 0.200, y la precisión mejoró de 61.9 % a 96.7 %.

Observando las gráficas (Figura 3.15 y Figura 3.16), se puede apreciar que, en términos generales, no hay diferencias significativas entre los *batch sizes* de 16 y 32 en términos medios, si en valores particulares ya que con 16 se pueden observar más fluctuaciones. Sin embargo, comparándolas con la Figura 3.10 con `batch_size = 8`, se observa que este valor ajusta mejor el modelo. Mientras que con *batch sizes* superiores a 8, el modelo deja de aprender características principales a partir de la época 7, con `batch_size = 8` esta época se extiende hasta la 12. Los *batch sizes* más grandes pueden llevar a que el modelo se ajuste a los patrones de los lotes en lugar de generalizar adecuadamente a todo el conjunto de datos, lo que podría explicar por qué el modelo deja de aprender características principales después de la época 7, ya que los grandes lotes suavizan demasiado las actualizaciones de los pesos.

En base a estos resultados, verificamos que con un `batch_size = 8`, se obtiene los mejores resultados, al igual que con un *step decay* de 10 unidades cada 20 *epochs*, como se había indicado en el artículo [43].

3.3. Autoencoder

Veamos ahora otro planteamiento para resolver este problema de clasificación de malware. Hemos visto en la sección 2.4.3 que los *autoencoders* son métodos de *machine learning* no supervisados y que entre sus funcionalidades no se encuentra la clasificación. Sin embargo, sus principales utilidades son la reducción de dimensionalidad, la detección de intrusiones o la eliminación de ruido. La motivación detrás del uso de un *autoencoder* es su capacidad para aprender una representación comprimida de los datos de entrada, que luego se puede utilizar para mejorar el rendimiento de un modelo de clasificación usando una secuencia de capas densas *fully connected*. A diferencia del modelo de CNN, no hemos usado de base ningún artículo que ya haya realizado experimentos con un *autoencoder* para resolver este problema de clasificación de malware, sino que todo ha sido trabajo propio.

Para este método de *machine learning*, vamos a usar todos los datos de la base de datos Microsoft Malware Classification (BIG 2015), tanto los 10.873 programas sin etiquetar como los 10.868 programas de entrenamiento con etiquetas. Primero, se preparan los datos para utilizarlos en el modelo. Para ello, vamos a utilizar el mismo procedimiento explicado en la sección 3.2.2 y crearemos imágenes 224×224 en escala de grises. A continuación, dependiendo de si las entradas del *autoencoder* tiene que ser en imagen o en vector, se aplanarán los datos o no. En segundo lugar, se entrenará el *autoencoder*. La dimensión del cuello de botella es de 4,096 para que coincida con la dimensión de la primera capa densa de la CNN. Para entrenarla, vamos a utilizar los 21.741 programas disponibles ya que no necesita las etiquetas para entrenarlo, además, el *autoencoder* se entrena minimizando el error cuadrático medio (MSE) entre los datos de entrada y su reconstrucción [34], optimizado con el algoritmo Adam [17]. Una vez se ha entrenado el AE y ya tenemos los parámetros entrenados para obtener una versión comprimida de los datos de entrada, ahora se entrena la segunda parte del modelo, una ANN. Para ello, las primeras capas del

ANN se corresponden con el *encoder* entrenado antes. Durante el entrenamiento del clasificador no seguirán entrenándose los pesos y bias de esta primera parte, sino que estos parámetros se bloquean [56]. A continuación de estas capas, se añaden varias capas densas intercaladas con capas de Dropout(0,5). Esta parte final coincide con la parte final de la CNN. Para entrenar esta segunda parte del modelo solo se utilizan los 10.860 programas etiquetados.

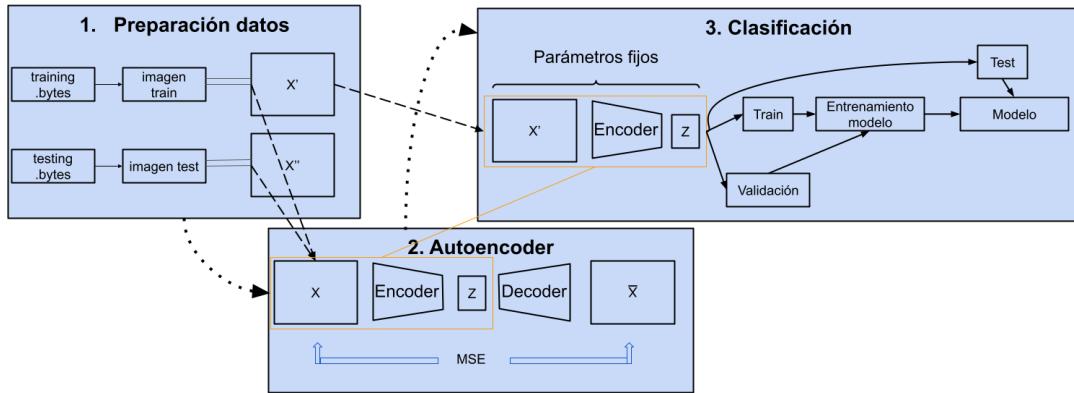


Figura 3.17: Esquema de la arquitectura de un modelo usando como primer paso un *autoencoder*.

Para realizar este modelo, se utilizan 3 formas diferentes para comprimir los datos: un *autoencoder* simple, un *autoencoder* con varias capas ocultas y un *autoencoder* convolucional, siguiendo el ejemplo de [17] para reconocimiento de dígitos MNIST.

3.3.1. Autoencoder simple

Para ello, como las capas utilizadas en esta arquitectura son densas, es necesario que los datos de entrada estén aplanados en un vector, luego lo primero que hacemos es preparar los datos para que tengan de dimensión 50,176. Esta capa se conecta completamente con el cuello de botella 4,096 que a su vez se vuelve a conectar con otra capa densa de 50,176 neuronas. Se entrenará este modelo durante 36 épocas y con mini-lotes de 8. Al acabar su entrenamiento, obtenemos una pérdida en los datos de entrenamiento del 0.04598 y en los datos de validación (20 % del total), un 0.04741.

A continuación, se extrae el *encoder* y se agrega una secuencia de 2 capas densas con activación ReLU junto con una capa de *dropout* cada una para evitar el sobreajuste. Para finalizar, se añade una capa densa con 9 neuronas y activación softmax para clasificar cada una de las instancias. La tasa de *dropout* utilizada es 0,4. Entrenamos el modelo durante 25 épocas con mini-lotes de 8. Además añadimos el *callback EarlyStopping* con *patience* = 4 y restaurando los pesos del mejor modelo en caso de parada temprana para prevenir el sobreajuste. El optimizador utilizado para entrenar el modelo de clasificación es el optimizador SGD (*Stochastic Gradient Descent*) con un *learning rate* inicial de 0,001. Además, fijamos el *momentum* en 0,9 y el *weight decay* en 0,0005. En cuanto a la función de pérdida, vamos a hacer 2 experimentos con *categorical_crossentropy* y con *mse* para comparar el rendimiento de diferentes configuraciones del modelo de clasificación. A continuación, se presentan los resultados de ambos modelos y se determina cuál es el más adecuado.

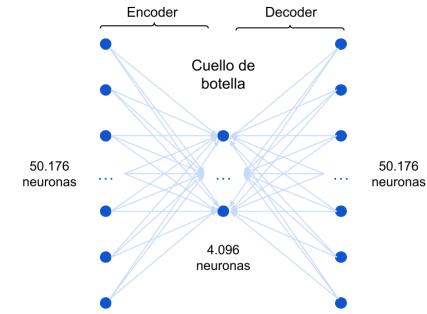


Figura 3.18: Esquema de la arquitectura usando un *autoencoder* simple.

El primer método para comprimir la información de entrada es usando un *autoencoder* simple con una capa oculta que se corresponde con el cuello de botella. La arquitectura de este método se puede observar en la figura 3.18

En el primer modelo, se utiliza como función de coste `mse`. Podemos observar en la figura 3.19 como muestra una pérdida de entrenamiento y validación que disminuyen consistentemente a lo largo de las épocas, con valores que oscilan entre 0.076 y 0.066. La precisión tanto en entrenamiento como en validación se mantiene alrededor de 0.6, lo que indica una capacidad limitada del modelo para generalizar. Aunque el modelo es estable y no muestra signos de sobreajuste, la precisión relativamente baja sugiere que podría beneficiarse de una arquitectura más compleja o de más datos de entrenamiento.

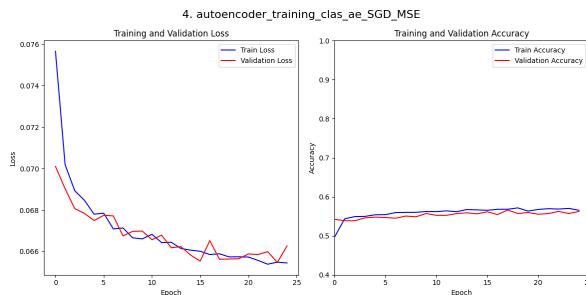


Figura 3.19: Entrenamiento y validación del modelo de clasificación usando un *autoencoder* con pérdida `mse`.

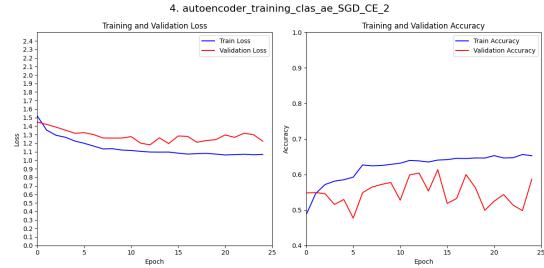


Figura 3.20: Entrenamiento y validación del modelo de clasificación usando un *autoencoder* con pérdida `categorical_crossentropy`.

En el siguiente modelo, se utiliza como función de coste `categorical_crossentropy`. Podemos observar en la figura 3.20 como obtiene una la función de pérdida presenta una dinámica diferente. La pérdida de entrenamiento disminuye de manera constante, alcanzando un valor alrededor del 1.1, mientras que la pérdida de validación también disminuye, aunque de manera menos pronunciada, estabilizándose alrededor de 1.3. La precisión de entrenamiento aumenta de manera constante hasta alcanzar aproximadamente 0.7, mientras que la de validación, aunque con más fluctuaciones, muestra una tendencia positiva alcanzando alrededor de 0.6. Este modelo muestra una mejor capacidad de aprendizaje en comparación con el modelo anterior y tiene un buen balance entre la pérdida de entrenamiento y validación, lo que sugiere que es capaz de generalizar mejor.

La diferencia de rendimiento entre `mse` y `categorical_crossentropy` en los modelos de clasificación se debe a sus diferentes enfoques hacia la salida esperada. MSE, al penalizar las diferencias cuadrática, puede no ajustarse adecuadamente a problemas de clasificación multiclas que requieren distribuciones de probabilidad sobre las clases. Por otro lado, `categorical_crossentropy` está diseñado específicamente para este tipo de problemas, evaluando la distancia entre la distribución predicha y la distribución real de las clases. En el análisis presentado, el modelo que utiliza `categorical_crossentropy` mostró una mejor capacidad para aprender y generalizar, reflejado en unos resultados más favorables en comparación con el modelo que emplea `mse`.

3.3.2. Autoencoder profundo

No tenemos que limitarnos a una sola capa como codificador o decodificador, en su lugar podríamos usar una pila de capas. El segundo método que vamos a abordar para resolver este problema de clasificación es usando un *autoencoder* con 3 capas densas en el *encoder* y otras tres capas densas en el *decoder*. En la figura 3.21 podemos ver su arquitectura y el número de

neuronas en cada capa. Se ha decidido ese número de neuronas en base a poder tener una red profunda de capas densas sin llegar a perjudicar computacionalmente el problema. Aun así, al aumentar el número de capas y de parámetros con respecto al modelo anterior, el tiempo de entrenamiento de cada época se multiplica por 4.

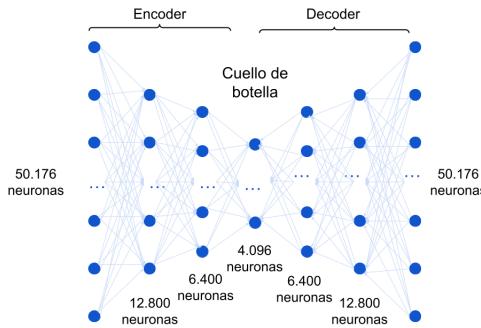


Figura 3.21: Esquema de la arquitectura usando un *autoencoder* profundo.

Al igual que en el modelo anterior, como vamos a usar capas densas, es necesario que los datos de entrada estén aplanados en un vector de dimensión 50,176. Este modelo se entrenará durante 25 épocas y con mini-lotes de 8. Se usará el optimizador Adam y función de pérdida `mse` como ya se comentó antes. Al acabar el entrenamiento, obtenemos una pérdida en los datos de entrenamiento del 0.04027 y en los datos de validación (20 % del total), un 0.04053. A pesar de que el modelo nos mostraba indicios de que podía seguir mejorando su rendimiento, debido a su alta carga computacional, decidimos parar su entrenamiento en su vuelta 25 ya que sus resultado ya mejoran a los del *autoencoder* con 1 única capa oculta. Con esta nueva representación comprimida de los datos, se vuelve a aplicar la capa de clasificación pero ahora solo con optimizador SGD y función de pérdida `categorical_crossentropy`.

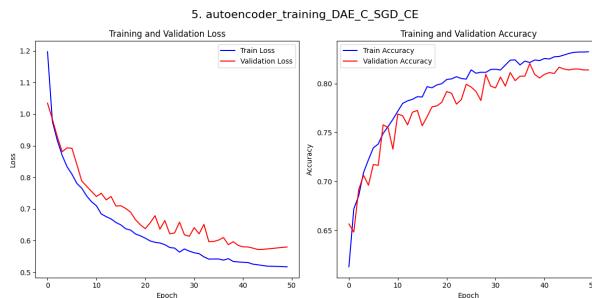


Figura 3.22: Entrenamiento y validación del modelo de clasificación usando un *autoencoder* profundo

A continuación, se crea la misma arquitectura que en el ejemplo anterior para clasificación. Entrenamos el modelo durante 50 épocas con mini-lotes de 8. En la gráfica proporcionada con los resultados obtenidos, se observa el comportamiento de las curvas de pérdida y precisión tanto para el conjunto de entrenamiento como para el de validación. La pérdida de entrenamiento disminuye constantemente desde un valor inicial cercano a 1.2 hasta aproximadamente 0.5, lo que indica un buen ajuste del modelo a los datos de entrenamiento. La pérdida de validación también baja, aunque de manera más moderada, desde cerca de 1.0 hasta alrededor de 0.65 en la época 30. Posteriormente, el `val_loss` muestra fluctuaciones y se estabiliza en torno a 0.57 a partir de la época 44, sugiriendo que el modelo ha dejado de mejorar significativamente en términos de generalización. La precisión de entrenamiento incrementa consistentemente de 0.62 a

aproximadamente 0.83, mientras que la precisión de validación, aunque inicialmente ascendente, se estabiliza alrededor de 0.82 después de la época 44, reflejando un estancamiento en la mejora del modelo para los datos de validación. Este patrón sugiere que el modelo ha alcanzado su capacidad óptima de generalización, y futuras mejoras en la precisión de entrenamiento no se reflejan en los datos de validación.

Comparado con el modelo anterior, este modelo muestra un mejor rendimiento en todas sus métricas, tanto en la pérdida como en la precisión, lo que sugiere una mejor capacidad de aprendizaje. La arquitectura mejorada de este modelo, con más capas ocultas en el *autoencoder*, permite un aprendizaje más profundo y efectivo, resultando en un ajuste más fino a los datos de entrenamiento y una mejor generalización.

3.3.3. Autoencoder convolucional

Por último, como nuestros datos de entrada los tenemos agrupados en imágenes, se nos ha ocurrido utilizar redes neuronales convolucionales como codificadores y decodificadores. Para ello, se van a utilizar capas convolucionales y capas de *max pooling* como *encoders* y capas convolucionales y de *upsampling* como *decoders*. En la imagen 3.23 se muestra la arquitectura de este modelo. Se ha seguido la arquitectura propuesta por Kayikci et al. [45] en su artículo *Convolutional Autoencoder Model for Reproducing Fingerprint* añadiendo alguna capa extra para llegar al cuello de botella con una dimensión exacta que en la CNN, 7 × 7 con 512 mapas de características.

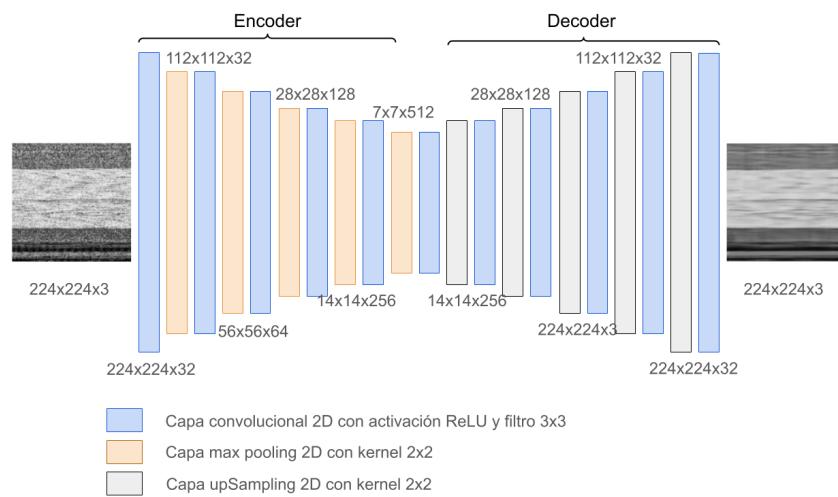


Figura 3.23: Esquema de la arquitectura usando un *autoencoder* convolucional.

Este modelo se entrenará durante 100 épocas y con mini-lotes de 8. Se usará el optimizador Adam y función de pérdida `mse` como ya se comentó antes. Al acabar el entrenamiento, observamos que la pérdida en la segunda época ya mejora la perdida obtenida en los otros dos modelos anteriores, con un 0.03967 en los datos de entrenamiento y un 0.03885 en los datos de validación. Después de las 100 vueltas, obtenemos 0.03528 en los datos de entrenamiento y 0.03498 en los datos de validación. A pesar de que el modelo nos mostraba indicios de que podía seguir mejorando su rendimiento, debido a su alta descenso continuo de la pérdida en los valores de entrenamiento y validación, se decide parar en la vuelta 100 debido a sus excelentes resultados obtenido hasta ese momento. En la imagen 3.24, podemos ver como funciona nuestro *autoencoder* convolucional y como la predicción de nuestro modelo se asemeja bastante a la imagen original.

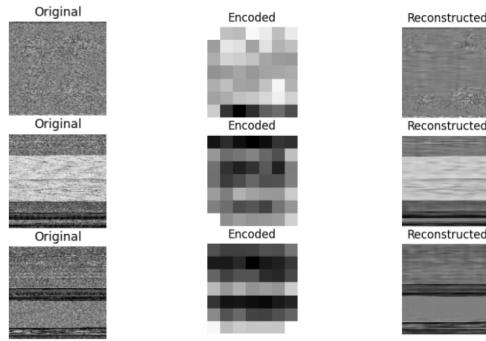


Figura 3.24: Visualización de la imagen original, versión comprimida y versión decodificada usando el *autoencoder* convolucional.

Con esta nueva representación comprimida de los datos, se vuelve a aplicar la capa de clasificación con optimizador SGD y función de pérdida `categorical_crossentropy` que vimos en la sección 3.2 de la CNN.

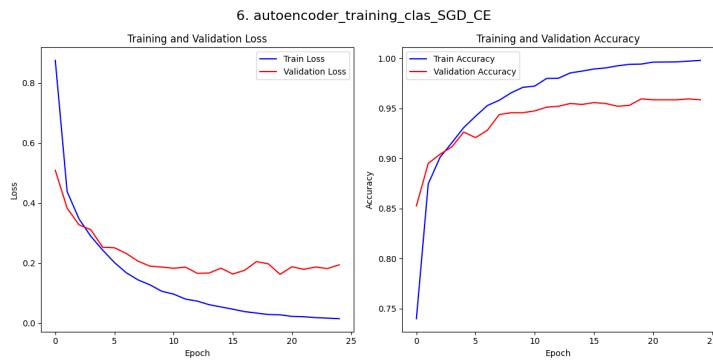


Figura 3.25: Entrenamiento y validación del modelo de clasificación usando un *autoencoder* convolucional.

En la Figura 3.25 se observa el comportamiento de las curvas de pérdida y precisión tanto para el conjunto de entrenamiento como para el de validación a lo largo de 25 épocas. La pérdida de entrenamiento disminuye de manera constante desde un valor inicial cercano a 0.9 hasta aproximadamente 0.015, indicando un buen ajuste del modelo a los datos de entrenamiento. La pérdida de validación también decrece, aunque de manera más moderada, desde cerca de 0.5 hasta estabilizarse alrededor de 0.2 a partir de la época 12, sugiriendo que el modelo ha dejado de mejorar significativamente en términos de generalización. La precisión de entrenamiento incrementa consistentemente de 0.85 a aproximadamente 1, mientras que la precisión de validación, aunque inicialmente asciende, se estabiliza alrededor de 0.95 después de la época 12, reflejando un estancamiento en la mejora del modelo para los datos de validación. Sin embargo, dado que el modelo utiliza *EarlyStopping* con restauración de los pesos del mejor modelo, se ha guardado el estado del modelo en el punto donde mejor generaliza, mitigando así el impacto del *overfitting*.

3.4. Resultados

En esta sección, se presentan y analizan los resultados obtenidos a partir de los diversos experimentos realizados. Se han explorado diferentes enfoques, incluyendo una CNN con técnicas de regularización como *dropout* y *EarlyStopping*, y tres tipos de *autoencoders*: simple, profundo y convolucional. De cada uno de estos enfoques se escogió el que tenía mejor rendimiento para ahora comparar ambos y quedarse con el modelo final para este problema de clasificación. A continuación, se detallan los resultados y conclusiones obtenidos de estos experimentos.

CNN con dropout y EarlyStopping

Tras definir la arquitectura de la CNN siguiendo el artículo [43] y realizar múltiples experimentos, encontramos que el modelo inicial, aunque alcanzaba un *accuracy* del 0.959 en el conjunto de validación, presentaba signos de sobreajuste. Para mejorar la generalización, se añadieron capas de *dropout* y se implementaron técnicas como *EarlyStopping*. Los resultados mostraron que una tasa de *dropout* del 0.5, combinada con la estrategia de detener el entrenamiento temprano, proporcionó el mejor balance entre precisión y capacidad de generalización, logrando un *accuracy* del 0.965 y un *loss* del 0.17 en validación. Estos ajustes permitieron mejorar significativamente el rendimiento del modelo. Para evaluar el rendimiento de este modelo en el problema de Microsoft Malware Classification, se ha generado una matriz de confusión con los datos de prueba que ya habíamos reservado para este análisis (15 % del total). A continuación, se procederá a calcular y analizar las métricas de precisión, TPR (sensibilidad), FPR, F1 Score, y proponer mejoras basadas en estos resultados.

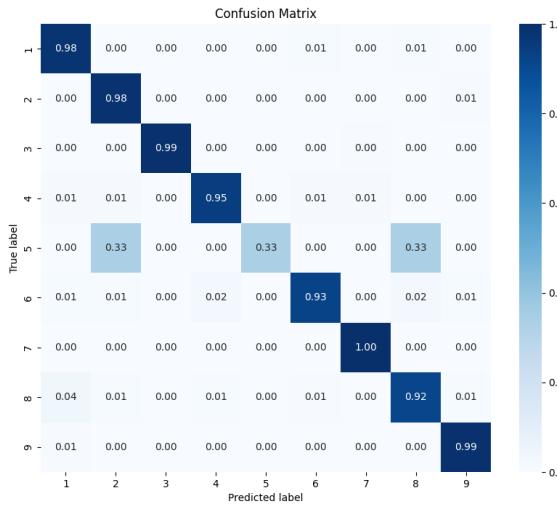


Figura 3.26: Matriz de confusión del modelo utilizando una CNN.

En la figura 3.26 se observa que la mayoría de las predicciones están bien clasificadas, ya que la diagonal principal de la matriz tiene los valores más altos (en azul oscuro), lo que indica un porcentaje mayor a 0.9. Sin embargo, hay algunas clases que no siguen este patrón, como la clase con etiqueta 5, que corresponde al tipo de malware Simbda. Esta clase obtiene un 33 % de éxito, lo que contrasta con el resultado global de éxito (96.5 %). Estos resultados sugieren que el modelo no ha generalizado adecuadamente para la clase 5, probablemente debido al bajo número de muestras en el dataset, ya que solo representa el 0.4 % del total.

Clase	Precisión	TPR (Sensibilidad)	FPR	F1 Score
1	0.98	0.98	0.01	0.98
2	0.98	0.98	0.00	0.98
3	0.99	0.99	0.00	0.99
4	0.95	0.95	0.00	0.95
5	0.33	0.33	0.67	0.33
6	0.93	0.93	0.02	0.93
7	1.00	1.00	0.00	1.00
8	0.92	0.92	0.01	0.92
9	0.99	0.99	0.00	0.99

Cuadro 3.2: Métricas de evaluación por clase para el modelo CNN.

Las métricas de la tabla 3.2 muestran un alto rendimiento para la mayoría de las clases, con F1 Scores que oscilan entre 0.92 y 1.00 para las clases 1, 2, 3, 4, 6, 7, 8 y 9. La clase 5, sin embargo, presenta un rendimiento significativamente inferior, con una precisión de 0.33, una sensibilidad de 0.33 y un F1 Score de 0.33, lo que indica dificultades en identificar correctamente los ejemplos de esta clase.

Para mejorar el rendimiento del modelo en la clase 5, se podría utilizar la técnica de pesos de clase [4]. Esta técnica consiste en asignar un peso específico a cada clase dentro de la red neuronal, permitiendo que la red preste más atención a aquellas clases con un peso mayor. Para determinar estos pesos, deben calcularse usando la siguiente fórmula:

$$W_j = \sqrt{\frac{s}{c \cdot S_j}}$$

Donde W_j representa el peso de la clase j , s es el número total de muestras, c es el número total de clases y S_j es el número de muestras de la clase j . Estos pesos se emplean durante el cálculo de la función de pérdida. Una vez que se ha calculado la pérdida para cada clase, esta se multiplica por el peso correspondiente de la clase. Este procedimiento permite “balancear” la importancia relativa de cada clase durante el entrenamiento de la red.

Autoencoder convolucional

En la segunda parte del capítulo 2, hemos explorado tres tipos de *autoencoders*: un *autoencoder* simple, un *autoencoder* profundo, y un *autoencoder* convolucional, cada uno evaluado en términos de pérdida y precisión tanto en entrenamiento como en validación. Los resultados muestran que el *autoencoder* convolucional destaca por su capacidad superior de compresión y generalización, alcanzando una precisión de validación del 95.8 % y una pérdida de validación de 0.192, superando significativamente a los *autoencoders* simples y profundos. Mientras que el *autoencoder* simple con una sola capa oculta presentó limitaciones en su capacidad de generalización, el *autoencoder* profundo mostró mejoras notables, aunque a costa de una mayor complejidad computacional. Por su parte, el *autoencoder* convolucional, al aprovechar la estructura espacial de los datos de entrada, logró una mayor precisión y una menor pérdida, estableciendo un gran equilibrio entre la complejidad del modelo y la capacidad de generalización, convirtiéndose así en la opción más precisa para esta tarea de clasificación.

Para evaluar el rendimiento del *autoencoder* convolucional en el problema de Microsoft Malware Clasification, se ha generado una matriz de confusión (Figura 3.27) con los datos de prueba.

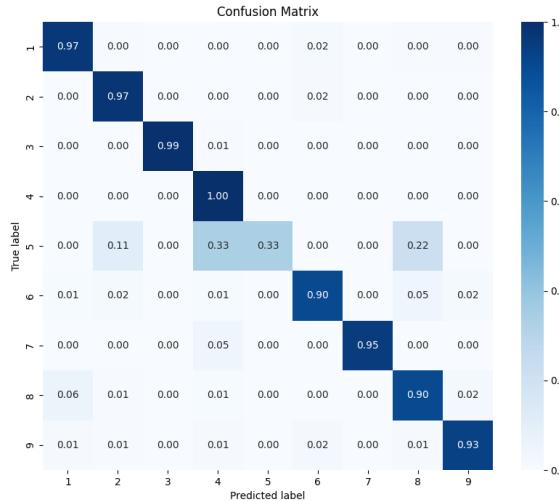


Figura 3.27: Matriz de confusión del modelo utilizando un *autoencoder* convolucional.

En ella se puede observar como la mayoría de las predicciones están bien clasificadas al igual que para la CNN, pero la clase con etiqueta 5 obtiene un 33 % de éxito también.

Además en la tabla 3.3, se puede observar una alta precisión y sensibilidad en la mayoría de las clases, con F1-scores que oscilan entre 0.89 y 0.98, lo que indica un gran equilibrio entre la precisión y la sensibilidad. La clase 3 sobresale con un F1-score de 0.98, reflejando una excelente precisión (0.97) y sensibilidad (0.99). Sin embargo, la clase 5 presenta un rendimiento significativamente inferior, con una precisión de 0.49, una sensibilidad de 0.33 y un F1-score de 0.39, señalando que el modelo tiene dificultades en identificar los ejemplos de esta clase. Este problema, al igual que con la CNN, podría resolverse usando pesos de clase.

Clase	Precisión	TPR (Sensibilidad)	FPR	F1 Score
1	0.94	0.97	0.06	0.96
2	0.92	0.97	0.07	0.94
3	0.97	0.99	0.03	0.98
4	0.91	1.00	0.09	0.95
5	0.49	0.33	0.06	0.39
6	0.94	0.90	0.03	0.92
7	0.95	0.95	0.05	0.95
8	0.88	0.90	0.12	0.89
9	0.94	0.93	0.06	0.94

Cuadro 3.3: Métricas de rendimiento por clase en el *autoencoder* convolucional.

En conclusión, a pesar de que tanto la CNN como el *autoencoder* convolucional presentan buenos resultados en general, la CNN demostró un mejor rendimiento en términos de *accuracy* y *loss* en el conjunto de validación, logrando un equilibrio óptimo entre precisión y capacidad de generalización. Este mejor rendimiento puede atribuirse a la estructura de la CNN, que es especialmente adecuada para capturar características espaciales complejas en los datos de entrada. Además, las técnicas implementadas, como el *dropout* y el *early stopping*, permitieron optimizar el modelo y prevenir el sobreajuste, haciendo de la CNN la opción más efectiva para el problema de *Malware Microsoft Classification Challenge (BIG 2015)*.

Capítulo 4

Detección de intrusiones

En la era actual, Internet se ha convertido en una herramienta esencial en nuestra vida cotidiana, facilitando actividades en áreas como los negocios, el entretenimiento y la educación [89]. Sin embargo, esta tendencia también ha acarreado un mayor riesgo de ataques en la red, lo que subraya la necesidad de sistemas de seguridad más potentes. Para garantizar la seguridad de las redes, es esencial contar con sistemas de detección de intrusiones capaces de identificar estos ataques (IDS, por sus siglas en inglés) [55].

La detección de anomalías se centra en identificar desviaciones de los patrones normales de uso, asumiendo que tales desviaciones pueden representar intentos de intrusión. Estos sistemas han evolucionado significativamente, incorporando técnicas avanzadas de aprendizaje automático para identificar diferencias entre datos anómalos y benignos con alta precisión. Además, estas técnicas han demostrado ser efectivas para manejar grandes volúmenes de datos y mejorar la capacidad de detección de nuevos tipos de ataques [55].

En este capítulo vamos a enfrentar este problema, escogiendo una de las bases de datos disponibles en [77] para poder identificar intrusiones. Como el objetivo principal de este trabajo es el estudio y puesta en práctica de diferentes algoritmos de aprendizaje automático, se ha decidido tomar como base de datos *KDD CUP 1999* [6], uno de los conjuntos de datos más utilizados para evaluar sistemas de detección de intrusiones. Otro de los motivos de esta elección ha sido la gran variedad de métodos que están documentados en el review para poder resolverlo, en particular de cuatro formas diferentes. Usando una Red Neuronal Convolutacional (CNN), una Red Neuronal Profunda (DNN), una Red Neuronal Recurrente (RNN) y un *autoencoder*.

4.1. KDD Cup 1999 Data

La base de datos *KDD CUP 1999* se desarrolló a partir de un conjunto de datos recopilado durante el Programa de Evaluación de Detección de Intrusiones DARPA de 1998, gestionado por los Laboratorios Lincoln del MIT (*Massachusetts Institute of Technology*). Este programa tenía como objetivo evaluar métodos de detección de intrusiones mediante la simulación de ataques en una red de área local (LAN) que imitaba el entorno de un área de la Fuerza Aérea de EE.UU [6].

Para crear el conjunto de datos, los Laboratorios Lincoln capturaron datos de TCP durante nueve semanas en esta red simulada. El conjunto de datos de entrenamiento incluye aproximadamente siete semanas de tráfico de red comprimido en cuatro *Gbytes* de datos binarios, procesados en unos cinco millones de registros de conexión. Los datos de prueba abarcan dos semanas adicionales y contienen alrededor de dos millones de registros de conexión [6].

Una conexión en el conjunto de datos se define como una secuencia de paquetes TCP entre una dirección IP de origen y una dirección IP de destino, con un inicio y fin definidos. Cada conexión está etiquetada como normal o como un ataque, con uno de los varios tipos específicos de ataque. Este enfoque permite analizar el comportamiento de las conexiones en un entorno controlado y detectar patrones asociados con actividades maliciosas [6].

En [6] nos encontramos con los siguientes archivos:

- *taskdescription*: Esta es la descripción original de la tarea proporcionada a los participantes de la competencia.
 - *kddcup.names*: Una lista de características.
 - *kddcup.data.gz*: El conjunto de datos completo (18M; 743M sin comprimir).
 - *kddcup.data_10_percent.gz*: Un subconjunto del 10% (2.1M; 75M sin comprimir).
 - *kddcup.newtestdata_10_percent_unlabeled.gz*: Un subconjunto del 10% de datos de prueba sin etiquetas (1.4M; 45M sin comprimir).
 - *kddcuptestdata.unlabeled.gz*: Datos de prueba sin etiquetas (11.2M; 430M sin comprimir).
 - *kddcuptestdata.unlabeled_10_percent.gz*: Un subconjunto del 10% de datos de prueba sin etiquetas (1.4M; 45M sin comprimir).
 - *training_attack_types*: Una lista de tipos de intrusión.
 - *typo-correction.txt*: Una breve nota sobre un error tipográfico en el conjunto de datos que ha sido corregido (26/6/07).

En este caso, utilizaremos únicamente el conjunto de datos *kddcup.data.gz* para hacer nuestros experimentos ya que necesitamos saber si una instancia es buena o mala para entrenarla en nuestras redes neuronales.

Cada registro del conjunto de datos contiene 41 características diferentes, las cuales se puede ver su descripción en la tabla 4.1. Además, contiene una característica final con el nombre con el que se clasifica la instancia. Podemos visualizar un ejemplo de como nos aparecen las instancias en el archivo *kddcup.data.gz* en la figura 4.1. Estas características se dividen en tres grupos principales: características de tráfico, características de conexión TCP y características de contenido.

Figura 4.1: Visualización del archivo **kddcup.data.gz**.

No.	Descripción	Tipo
Características de Conexiones TCP		
1	duración de la conexión (s)	int64
2	tipo protocolo	str
3	tipo servicio red destino	str
4	estado de la conexión	str
5	bytes origen-destino	int64
6	bytes destino-origen	int64
7	1 si puerto/host similar, 0 otro caso	int64
8	fragmentos incorrectos	int64
9	paquetes urgentes	int64
Características de Contenido		
10	indicaciones calientes	int64
11	intentos de inicio de sesión fallidos	int64
12	1 si inicio de sesión aceptado	int64
13	operaciones comprometidas	int64
14	1 si acceso root en shell	int64
15	invocación de su root	int64
16	accesos concedidos como root	int64
17	creaciones de archivos	int64
18	sesiones de shell	int64
19	operaciones de acceso a archivos	int64
20	instrucciones salientes en ftp	int64
21	1 si inicio de sesión es de host	int64
22	1 si inicio de sesión como invitado	int64
Características de Tráfico		
23	enlaces en 2s al mismo host	int64
24	enlaces en 2s al mismo servicio	int64
25	% fallos SYN	float64
26	% conexiones hacia mismo servicio con errores SYN	float64
27	% fallos REJ	float64
28	% conexiones hacia el mismo servicio que tuvieron errores REJ	float64
29	% conexiones a un solo servicio	float64
30	% conexiones a varios servicios	float64
31	% conexiones a varios hosts	float64
32	conexiones al mismo host de destino	int64
33	conexiones al mismo servicio	int64
34	% conexiones al mismo host	float64
35	% conexiones varios servicios	float64
36	% conexiones mismo puerto de origen	float64
37	% conexiones varios hosts al mismo servicio	float64
38	% conexiones fallos S0 al host actual	float64
39	% conexiones fallos S0 al host y servicio	float64
40	% fallos RST al host actual	float64
41	% fallos RST al host y servicio	float64
42	tipo de ataque	str

Cuadro 4.1: Características del Conjunto de Datos KDD

- **Características de conexiones TCP:** Examina las características que describen los atributos básicos de cada conexión TCP.

- **Características basadas en el contenido:** Estas características examinan el contenido de los paquetes para identificar patrones sospechosos.
- **Características de tráfico:** Estas características analizan patrones de tráfico en la red para detectar anomalías.

El conjunto de datos de entrenamiento del *KDD Cup 1999* comprenden diferentes formas de amenazas y datos normales. Cada uno de los datos anómalos pertenecen a una de las siguientes 22 formas diferentes de ciberataques (ver tabla 4.2). Estos ataques pueden clasificarse en 4 grupos, que según [7], podemos describirlos como:

- **Ataque de Denegación de Servicio (DoS):** ocurre cuando un intruso impide que los usuarios autorizados accedan a un sistema sobrecargando los recursos del sistema (computación o memoria), haciéndolo incapaz de ejecutar solicitudes válidas, como en el caso de una inundación SYN.
- **Ataque de Usuario a Root (U2R):** ocurre cuando el intruso obtiene permisos locales para acceder al sistema como un usuario legítimo y luego intenta explotar debilidades para obtener acceso root al sistema, ganando así capacidades de supervisión.
- **Ataque de Remoto a Local (R2L):** ocurre cuando el intruso transmite un paquete a través de la red desde una estación de trabajo remota sin proporcionar el permiso adecuado, como al intentar varias contraseñas.
- **Ataque de Sondeo:** ocurre cuando un intruso intenta obtener conocimiento sobre la red para descubrir fallos de seguridad. A través de esta operación, el intruso explora la arquitectura de la red e identifica las categorías de servicios disponibles en el sistema utilizando técnicas como un escaneo de puertos.

En la tabla 4.2 podemos ver los tipos de ataque que incluye cada grupo y el número total de muestras que tiene.

Categoría	Ataque	Total
DOS	pod, land, neptune, smurf, teardrop, back	3.883.370
R2L	warezclient, imap, phf, spy, ftpwrite, guesspswd, warezm, multihop	1.126
U2R	rootkit, loadmodule, perl, buffer_overflow	52
Probing	satan, nmap, portsweep, ipsweep	41.102
Normal		972.781

Cuadro 4.2: Categorías de Ataques

El propósito de la competición *KDD Cup 1999*, en la cual se utilizó este conjunto de datos, era desarrollar un modelo predictivo capaz de distinguir entre conexiones normales y ataques [6]. Este conjunto de datos se ha convertido en una referencia clave en estudios sobre la detección de intrusiones en redes debido a su riqueza y diversidad de ejemplos de ataques.

4.2. Preparación datos

De estas 41 características de tráfico, 38 son numéricas y 3 son simbólicas. Para unificar los formatos de datos, transformamos las características simbólicas en datos numéricos utilizando vectores *one-hot*. Las 3 características simbólicas son: el tipo de protocolo en la capa TCP/IP, el tipo de servicio del sistema objetivo, y el tipo de bandera que indica el estado de la conexión de la sesión. Tenemos tres tipos de protocolos, ICMP, TCP, y UDP. Estos protocolos se transforman en un vector de tres dimensiones, resultando en las representaciones (1,0,0), (0,1,0) y

(0,0,1) respectivamente. De manera similar, los 70 tipos de servicio, que incluyen HTTP y FTP, se transforman en vectores de 70 dimensiones, y las 11 características de tipo de bandera se convierten en vectores de 11 dimensiones. A través de estas transformaciones, generamos un vector de 84 dimensiones, que al combinar con las 38 características originales numéricas, obtenemos un vector final de 122 dimensiones.

Además, cada valor lo estandarizamos utilizando la siguiente fórmula:

$$X_{inorm} = \frac{x_i - x_{imin}}{x_{imax} - x_{imin}} \quad (4.1)$$

donde $X_{inorm} \in \mathbb{R}$ hace referencia al nuevo valor normalizado, $x_i \in \mathbb{R}$ al valor que se encuentra en la base de datos y $x_{imax}, x_{imin} \in \mathbb{R}$ al valor máximo y mínimo de esa característica respectivamente. De esta forma, todas las características se van a encontrar entre [0, 1] [51].

Ahora, al igual que en el problema de clasificación de malware del Capítulo 3, vamos a dividir el conjunto de datos aleatoriamente usando la función `train_test_split()` en grupos del 75 %, 15 % y 10 % para entrenamiento, test y validación respectivamente.

Para abordar este problema de clasificación, vamos a realizar cuatro modelos de aprendizaje automático diferentes para luego comparar sus resultados. El primer método que vamos a utilizar es una Red Neuronal Profunda (DNN), seguido de una Red Neuronal Recurrente (RNN) y un *autoencoder*. Para finalizar, utilizaremos una Red Neuronal Convolutacional (CNN) para la detección de intrusiones. Todos los modelos que vamos a crear han sido sacados de algún artículo que aparece en el *review* utilizado a lo largo de todo el trabajo [77]

4.3. Red Neuronal Profunda

Vamos a empezar creando modelo de Red Neuronal Profunda (DNN) con una capa de entrada, dos capas ocultas y una capa de salida. En esta sección se describe en detalle la arquitectura del modelo, los hiperparámetros utilizados y el proceso de entrenamiento siguiendo principalmente los artículos [61, 90]. Aunque la gran mayoría del trabajo de esta sección ha sido replicar los experimentos de estos artículos, se ha añadido el *callback EarlyStopping* para prevenir el sobreajuste del modelo.

4.3.1. Arquitectura del Modelo

La arquitectura del modelo DNN consta de varias capas densas interconectadas, comenzando con la capa de entrada que recibe los datos preprocesados. La capa de entrada está compuesta por 122 neuronas, cada una de las cuales corresponde a una característica del conjunto de datos tras el procesamiento visto anteriormente.

A continuación, se encuentran dos capas ocultas. La primera capa oculta contiene 50 neuronas con la función de activación ReLU, seleccionada por su capacidad para manejar de manera eficiente las no linealidades y su facilidad de entrenamiento. La segunda capa oculta incluye 30 neuronas, también con la función de activación ReLU. Esta estructura permite al modelo aprender representaciones complejas de los datos a través de la jerarquía de capas.

Por último, la capa de salida del modelo está diseñada para la tarea de clasificación, utilizando la función de activación `softmax`. Esta capa convierte las salidas en probabilidades de pertenencia

a cada una de las 5 clases. El número de neuronas en la capa de salida se corresponde entonces con 5, ya que con 5 neuronas podemos hacer un estudio multiclasificación y a su vez se puede modificar para obtener una clasificación binaria. La Figura 4.2 ilustra la arquitectura general de la DNN propuesta.

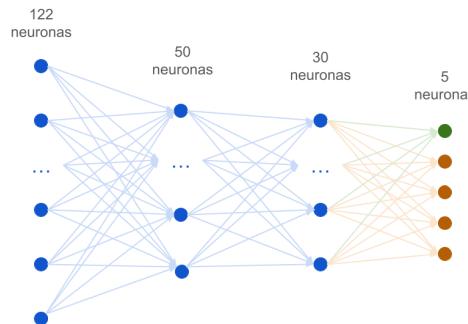


Figura 4.2: Arquitectura de la Red Neuronal Profunda (DNN), donde el color verde hace referencia a que se clasifica como benigno y el color rojo como uno de los cuatro tipos de malware.

Una vez tenemos definida la arquitectura de nuestro modelo, es hora de elegir los hiperparámetros necesarios para entrenarlo. El optimizador elegido es Adam (*Adaptive Moment Estimation*), configurado con una tasa de aprendizaje inicial de 0.001. Para la función de pérdida, se utilizó la entropía cruzada categórica (*categorical_crossentropy*). Esta función es adecuada para problemas de clasificación multiclasificación, como nuestro problema con la base de datos *KDD Cup 1999*. Aunque en el análisis que vamos a hacer posteriormente solo tenemos en cuenta si las instancias son malignas o benignas, se ha decidido hacer una clasificación multiclasificación ya que se obtiene mayor información y que además posteriormente se puede pasar a una clasificación binaria fácilmente.

El modelo se entrenó durante 30 épocas, utilizando un tamaño de lote de 32 muestras por lote. Esta configuración permite un balance adecuado entre la velocidad de entrenamiento y la estabilidad de la convergencia.

Además, he decidido añadir el *callback EarlyStopping* para detener el entrenamiento si la pérdida en el conjunto de validación no mejora después de 5 épocas, ayudando a prevenir el sobreajuste. Además, se emplea un *CSVLogger* para registrar las métricas de entrenamiento y validación en cada época. La Figura 4.3 presenta la evolución de la precisión y la pérdida a lo largo de las épocas de entrenamiento, destacando la mejora continua del modelo.

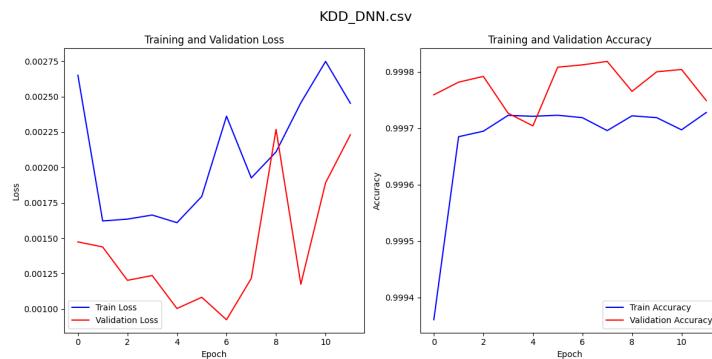


Figura 4.3: Evolución de la Precisión y la Pérdida durante el Entrenamiento del Modelo DNN.

En la imagen 4.3 se aprecian fluctuaciones en la pérdida de entrenamiento y validación a lo largo de todo el entrenamiento. La pérdida desciende hasta la época 6 y luego asciende. En cuanto a la precisión, se observa que es muy alta a lo largo de todo el entrenamiento, alrededor del

0.99. Aunque las curvas muestran fluctuaciones, estas son en realidad pequeñas variaciones en las centésimas del porcentaje.

Podemos apreciar que, a pesar de que el número de épocas iniciales era de 30, no aparecen resultados a partir de la 11. Esto es debido a que en la época 6 el modelo ha obtenido un `val_loss = 0,001` y desde esta época a la 11 no ha obtenido un valor menor, luego se ha producido la parada temprana, restaurando los pesos de la época 6. El valor del *training loss* en esta época es de 0,00231. Por otra parte, la precisión de clasificación multiclas en esta época es de 0,997 en el entrenamiento y 0,998 en la validación.

4.3.2. Evaluación del modelo

Después de entrenar el modelo, se evalúa como una clasificación binaria, usando el conjunto de datos que habíamos reservado para esta parte. Para empezar vamos a ver la matriz de confusión 4.4 con el número de instancias que están bien clasificadas y mal clasificadas.

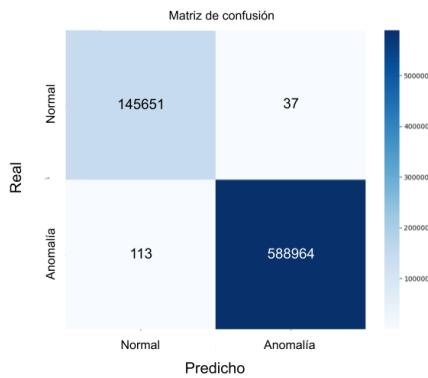


Figura 4.4: Matriz de confusión del modelo utilizando una DNN.

En ella se puede observar como la mayoría de las predicciones están bien clasificadas ya que la mayoría de muestras se encuentran en la diagonal. Para poder ver con más detalle estos resultados, veamos ahora la tabla 4.3, en la que podremos ver la precisión del modelo, su sensibilidad, la especificidad y el *F1-Score*. Los resultados indican un desempeño sobresaliente en todas las métricas evaluadas, cada una con un valor de 0.999.

Modelo	Precisión	Sensibilidad	Especificidad	F1 Score
DNN	0.999	0.999	0.999	0.999

Cuadro 4.3: Métricas de rendimiento de la red neuronal profunda.

La precisión de 0.999 indica que el modelo clasifica correctamente prácticamente todos los registros de conexión. La especificidad de 0.999 muestra que el modelo detecta casi todos los registros anómalos, minimizando los falsos benignos. La sensibilidad de 0.999 revela que el modelo identifica correctamente casi todos los datos benignos. Finalmente, el *F1-Score* obtiene un 0.999, lo que confirma gran capacidad del modelo para identificar correctamente tanto las instancias benignas como anómalas.

4.4. Red Neuronal Recurrente

Después de evaluar la red neuronal profunda, vamos a crear otro modelo predictivo para ver si podemos obtener unos resultados mejores. En este caso, vamos a enfocar el estudio en implementar una Red Neuronal Recurrente (RNN) con *Long Short-Term Memory* (RNN-LSTM) para este problema de detección de intrusiones. Se va a seguir principalmente las metodologías propuestas en [99, 48].

4.4.1. Arquitectura del Modelo

El modelo LSTM comienza con una capa de entrada que recibe los datos preprocesados anteriormente con una dimensión de 122 características. Estos datos son procesados por una capa LSTM que contiene 80 unidades ocultas. Este número de neuronas ha sido escogido después de obtener los mejores resultados comparado con otros números de neuronas ocultas en el artículo [99]. Esta capa LSTM es crucial ya que es capaz de capturar características a largo plazo en secuencias de datos, lo cual es esencial para nuestro estudio.

A continuación, la salida de la capa LSTM se pasa a una capa completamente conectada (capa Densa). Esta capa Densa tiene 5 neuronas, que corresponden a las 5 clases de salida. Se escoge 5 clases en vez de 2 porque creemos que este número de neuronas nos da más información y además nos da la posibilidad de hacer un análisis más extenso de sus resultados. La función de activación utilizada en esta capa es *softmax*, la cual convierte los valores de salida en una distribución de probabilidad sobre las posibles clases, permitiendo así realizar la clasificación. Podemos ver un esquema de la arquitectura en la imagen 4.5.

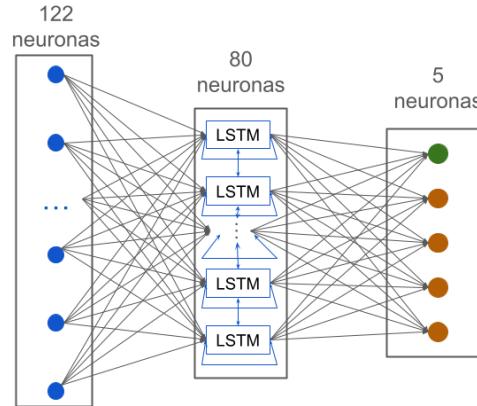


Figura 4.5: Arquitectura de la Red Neuronal Recurrente (RNN), donde el color verde hace referencia a que se clasifica como benigno y el color rojo como uno de los cuatro tipos de malware.

Una vez definido el modelo, se procede a su compilación utilizando el optimizador *Stochastic Gradient Descent* (SGD) con una tasa de aprendizaje que se va ajustando a lo largo del entrenamiento. Este optimizador se elige por su simplicidad y eficacia en muchos problemas de aprendizaje automático. Para la función de pérdida, se utilizó la entropía cruzada categórica (*categorical_crossentropy*) como en el modelo anterior ya que tenemos 5 neuronas en la capa de salida.

El modelo se entrena durante 50 épocas con un tamaño de lote de 50. Al igual que en el modelo de DNN, he decidido añadir los callbacks *CSVLogger* y *EarlyStopping* para registrar el proceso

de entrenamiento y para prevenir el sobreajuste respectivamente. La Figura 4.6 presenta la evolución de la precisión y la pérdida a lo largo de las épocas de entrenamiento.

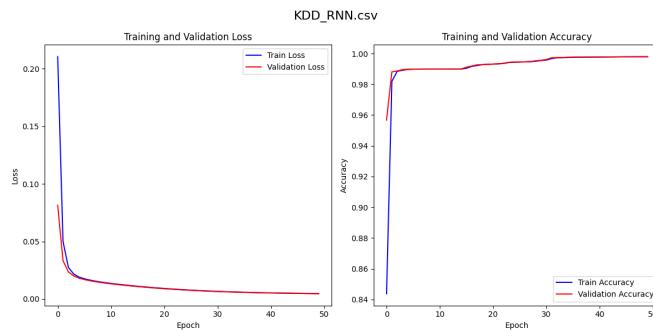


Figura 4.6: Evolución de la Precisión y la Pérdida durante el Entrenamiento del Modelo RNN - LSTM.

En la imagen 4.6, observamos la evolución de la pérdida de entrenamiento y validación (gráfica izquierda). La pérdida de entrenamiento comienza alta, con un valor de 0.21, y disminuye rápidamente en las primeras 10 épocas. La pérdida de validación sigue un patrón similar, disminuyendo rápidamente y convergiendo con la pérdida de entrenamiento alrededor de la época 10. Ambas curvas se estabilizan alrededor de la época 20 y permanecen bajas, llegando a un valor de 0.004 para ambos subconjuntos.

En la gráfica de la derecha, se observa la precisión de entrenamiento y validación. La precisión crece rápidamente en las primeras 10 épocas, para estabilizarse a partir de esa época. Los datos de entrenamiento comienzan con un valor de 0.843, y aumenta rápidamente, alcanzando un valor de 0.999. La precisión de validación sigue de cerca a la precisión de entrenamiento, también alcanzando valores muy altos, 0.999.

En ambas gráficas se observa un comportamiento muy parecido de ambas curvas, además de converger a unos resultados excelentes rápidamente, lo cual es un buen indicio de que el modelo está aprendiendo eficientemente y generalizando bien.

4.4.2. Evaluación del modelo

Después de entrenar el modelo, se evalúa como una clasificación binaria, usando el conjunto de prueba que habíamos reservado anteriormente. Para empezar, vamos a ver la matriz de confusión 4.7 con el número de registros de la *KDD CUP 1999* que están bien clasificados y mal clasificados.

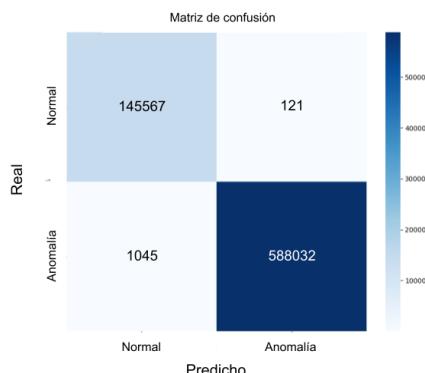


Figura 4.7: Matriz de confusión del modelo utilizando una RNN.

En ella, al igual que en el modelo anterior (DNN), se puede observar como la mayoría de las predicciones se encuentran en la diagonal. Para poder ver con más detalle estos resultados, veamos ahora la tabla 4.4, en la que podremos ver la precisión del modelo, su sensibilidad, la especificidad y el *F1-Score*. Los resultados indican un desempeño sobresaliente en todas las métricas evaluadas, cada una con un valor de 0.999.

Modelo	Precisión	Sensibilidad	Especificidad	F1 Score
RNN	0.998	0.999	0.998	0.998

Cuadro 4.4: Métricas de rendimiento de la red neuronal recurrente.

En la tabla podemos volver a apreciar el gran rendimiento de este modelo con el conjunto de datos *KDD CUP 1999*. Las métricas expuestas anteriormente nos muestran unos resultados excelentes, con valores prácticamente iguales a 1. Esto nos indica que la RNN - LSTM tiene una capacidad casi perfecta para distinguir entre registros anómalos y benignos.

4.5. Autoencoder

A continuación, se plantea el problema de detección de intrusiones utilizando uno de los métodos más utilizados para resolver este problema, el *autoencoder* [77]. La estrategia consiste en entrenar el *autoencoder* exclusivamente con un subconjunto de datos benignos. Posteriormente, se evalúa la capacidad del modelo para reconstruir los demás registros. En función del error de reconstrucción, los registros se clasificarán como normales o como malware, basándose en un umbral predefinido. Este enfoque se basa principalmente en el trabajo de [95].

4.5.1. Arquitectura del modelo

El *autoencoder* propuesto para esta sección contiene 5 capas densas en las que se codifica una representación de características de 122 dimensiones en un vector de 5 dimensiones en su capa latente. Antes de llegar a esta dimensión, primero se reduce de 122 a 32 para a continuación reducir aun más hasta llegar a 5. Después, este representación comprimida de los datos se decodifican de vuelta a un vector de 32 dimensiones y después a 122 como los datos de entrada. En la imagen 4.8 podemos ver de manera más gráfica su arquitectura.

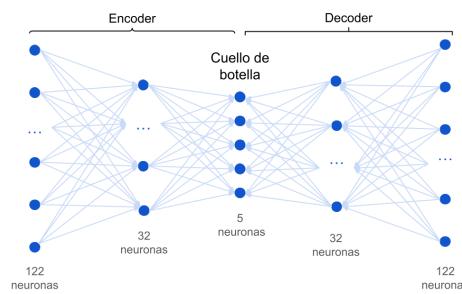


Figura 4.8: Arquitectura del *autoencoder* con 5 capas densas.

Las capas ocultas son capas densas, es decir, están completamente conectadas y utilizan como función de activación ReLU. El error de reconstrucción entre la entrada x y la salida reconstruida \hat{x} se calcula usando el error absoluto medio (MAE).

Una vez definido el modelo, se procede a compilarlo utilizando el optimizador Adam con un *learning rate* inicial de 0,001.

El modelo se entrena durante 30 épocas en mini lotes de 30 registros. Al igual que en los dos modelos anteriores, he decidido añadir los *callbacks* CSVLogger y EarlyStopping para registrar el proceso de entrenamiento y para prevenir el sobreajuste respectivamente. La Figura 4.9 presenta la evolución de la precisión y la pérdida a lo largo de las épocas de entrenamiento.

En la figura, se observa la evolución de la pérdida de entrenamiento y validación a lo largo de las 30 épocas. Al inicio del entrenamiento, se aprecia un descenso rápido en ambas curvas de pérdida. Esto indica que el modelo está aprendiendo eficazmente las características de los datos durante las primeras épocas, lo que es un buen signo de que la fase inicial del entrenamiento está ajustando bien los pesos.

A partir de la época 15, tanto la pérdida de entrenamiento como la de validación comienzan a estabilizarse y converger alrededor de un valor constante 0,0064. Este comportamiento sugiere que el modelo ha alcanzado un punto de equilibrio donde las actualizaciones de los pesos no producen mejoras significativas, indicando una buena adaptación del modelo a los datos. Es importante destacar que a pesar de la buena adaptación del modelo a los datos de entrenamiento, ambas curvas son muy similares a lo largo de todas las épocas, lo que indica que el modelo generaliza bien los datos de entrenamiento sin llegar al sobreajuste.

4.5.2. Evaluación del modelo

Después de entrenar el modelo con los datos normales de entrenamiento, se procede a la evaluación del modelo. Para ello, primero tenemos que fijar un umbral que nos diga si un registro es malware o normal dependiendo de su error de reconstrucción. Para ello, utilizando la función de `sklearn.metrics.precision_recall_curve`, se calcula la curva de precisión y recuperación (*Precision-Recall*) en función de diferentes umbrales que se utilicen para clasificar los registros. A partir de esta curva, se selecciona un umbral alto que balancee la precisión y la recuperación para la detección de anomalías. Este proceso se ilustra en la siguiente gráfica:

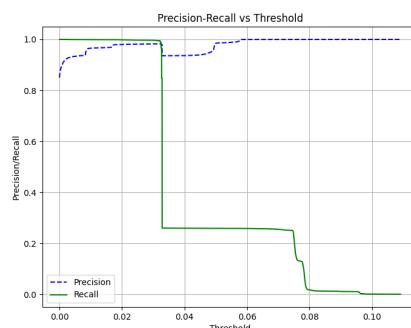


Figura 4.10: Gráfica de precisión y recuperación en función del umbral

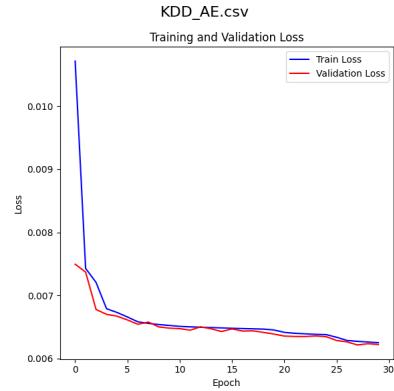


Figura 4.9: Evolución de la pérdida durante el entrenamiento del *autoencoder*

Nuestro objetivo consiste en minimizar los registros anómalos que se clasifiquen como normales. Para ello, tenemos que encontrar un umbral que maximice el *recall*. Un valor que maximiza tanto el recall como la precisión es 0.02, luego se decide utilizar este valor como umbral.

Con este umbral, vamos a evaluar el modelo usando los datos de prueba. Para empezar, vamos a ver la matriz de confusión 4.11 con el número de registros de la *KDD CUP 1999* que están bien clasificados y mal clasificados.

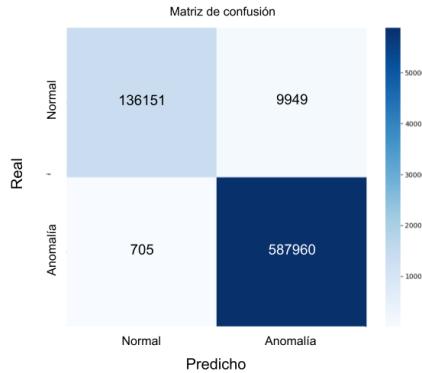


Figura 4.11: Matriz de confusión del modelo utilizando una *autoencoder*.

En ella, al igual que en los otros dos modelos, se obtienen unos resultados excelentes. Veamos ahora la tabla 4.5 con el resto de métricas.

Modelo	Precisión	Sensibilidad	Especificidad	F1 Score
Autoencoder	0.999	0.93	0.999	0.991

Cuadro 4.5: Métricas de rendimiento del *autoencoder*.

En la tabla podemos volver a apreciar el gran rendimiento de este modelo con el conjunto de datos *KDD CUP 1999*. Las métricas expuestas anteriormente nos muestran unos resultados excelentes, con valores prácticamente iguales a 1. Estos nos indica que la arquitectura de *autoencoder* utilizada tiene una capacidad casi perfecta para detectar tanto registros benignos como registros anómalos.

4.6. Red Neuronal Convolucional

Por último, vamos a plantear el problema de detección de intrusiones utilizando una CNN. Este problema no estaba clasificado en el *review* [77], pero con el conocimiento que ya poseíamos sobre este método unido a la búsqueda de un artículo que tratara este tema, hemos decidido optar por hacer un estudio con esta red neuronal. Para ello, se va a seguir la metodología que se usa en [49, 97].

4.6.1. Visualización de los datos

En la subsección 4.2 estudiamos como procesar los datos originales de la base de datos, a datos que pudieran ser utilizados por la redes neuronales. Pasamos de un vector de 41 características

con diferentes tipos de datos, a un vector de 122 características donde todos los elementos eran números reales entre 0 y 1. Como bien hemos visto en el capítulo 2, las CNN son redes neuronales que están diseñadas para procesar datos almacenados en matrices. En este caso, al igual que en el modelo de CNN utilizado en la clasificación de malware, vamos a pasar del vector unidimensional a un vector bidimensional.

En el artículo [49], los datos son más antiguos que los actuales y en vez de tener un vector de 122 dimensiones, ellos tienen un vector de 117 dimensiones que redimensionan a una matriz 13×9 . En el artículo [97], pasan del vector de 41 características a una matriz 7×7 y los valores restantes le asignan un 0. En nuestro caso, he decidido hacer un estudio más exhaustivo de las 122 características para ver que opciones podía plantear para convertir el vector en matriz.

Primero se observó que $122 = 61 \times 2$. Creí que esta matriz no sería muy efectiva de primera mano, ya que había una gran diferencia entre el número de filas y el número de columnas. En este punto, decidí hacer un estudio de las 122 características. Como para estandarizar los datos tuve que calcular los máximos y mínimos de cada característica, decidí fijarme en ellos. Se me ocurrió ver si había alguna característica que siempre valiera lo mismo. Esto era tan fácil como comprobar si algún valor máximo era igual a su valor mínimo. Sea D_i el conjunto de todas las características i -ésimas del *dataset*, entonces,

$$\exists i \in [1, 122] \quad \forall x \in D_i \quad \min(D_i) \leq x \leq \max(D_i) = \min(D_i) \quad (4.2)$$

luego $\forall x \in D_i, x = \min(D_i)$. Comprobando esta propiedad, me dí cuenta que la característica 101 siempre valía 0, es decir la característica 20 de los datos originales (las instrucciones salientes en FTP, que se refiere al número de comandos enviados desde el servidor FTP hacia un cliente durante una sesión de transferencia de archivos). Entonces, he decidido prescindir de esta característica para pasar de un vector de 122 dimensiones a un vector de 121 dimensiones que puede redimensionarse a una matriz 11×11 .



Figura 4.12: Visualización de un registro benigno del *KDD CUP 1999*.

4.6.2. Arquitectura del Modelo

La arquitectura del modelo CNN se compone de varias capas convolucionales, de *max-pooling* y densas, diseñadas para procesar las matrices 11×11 generadas en la fase de preprocesamiento. La primera capa convolucional aplica 64 filtros convolucionales con un tamaño de kernel de 3×3 , utilizando la función de activación ReLU. Esta capa inicial se encarga de detectar las características locales más importantes de la imagen.

A continuación, se utiliza una capa de Max-Pooling con un tamaño de ventana de 2×2 para reducir la dimensionalidad de los mapas de características, conservando la información relevante y disminuyendo la carga computacional.

La segunda capa convolucional aplica 128 filtros convolucionales con un tamaño de kernel de

3×3 , también con función de activación ReLU. Esta capa permite capturar características más complejas a un nivel superior de abstracción, detectando patrones más detallados en los datos.

Después de la segunda capa convolucional, se emplea otra capa de *max-pooling* con un tamaño de ventana de 2×2 , para continuar reduciendo la dimensionalidad de los mapas de características y conservar los patrones más importantes.

Los mapas de características obtenidos se aplanan y pasan a través de una capa densa de 128 neuronas con función de activación ReLU. Esta capa permite compactar la información obtenida por las capas convolucionales. Después, se pasa la información por una capa densa con 64 neuronas y función de activación ReLU refina aún más esta representación.

Finalmente, se pasa por la capa de salida con la función de activación softmax, que es una capa densa con 5 neuronas como en la RNN y la DNN, ya que de esta forma se obtiene más información aunque luego se haga un análisis de clasificación binaria.

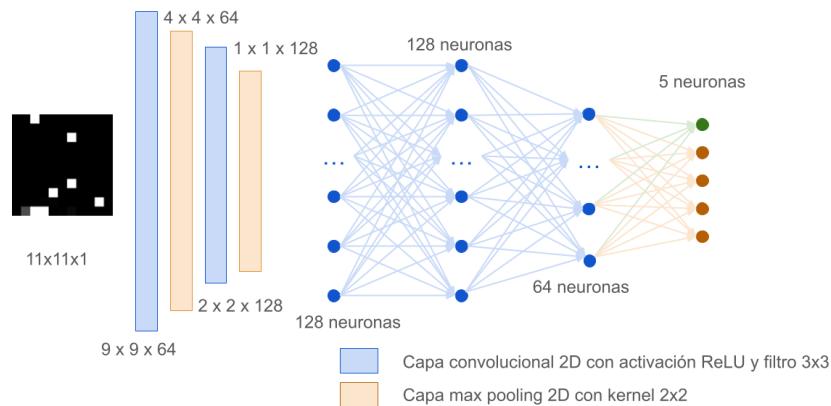


Figura 4.13: Arquitectura de la Red Neuronal Convolucional para la detección de intrusiones de la *KDD CUP*.

Una vez creada la arquitectura de la CNN, se compila el modelo con el optimizador Adam con una tasa de aprendizaje inicial de 0,001. La función de pérdida seleccionada es `categorical_crossentropy`. El modelo se entrena en lotes de tamaño 32 durante 30 épocas.

Como en los modelos anteriores, se añaden los *callbacks* `EarlyStopping` y `CSVLogger` para evitar el sobreajuste y registrar las métricas de entrenamiento respectivamente. La evolución del entrenamiento a lo largo de las 30 épocas se puede ver en la imagen 4.14.

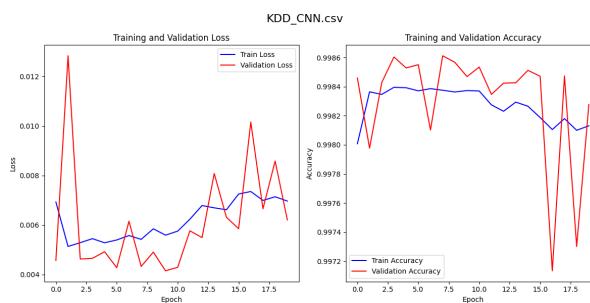


Figura 4.14: Evolución de la precisión y la pérdida durante el entrenamiento de la CNN.

En la imagen, podemos observar la evolución de la pérdida de entrenamiento y validación (gráfica izquierda). La pérdida de entrenamiento comienza en un valor bajo, 0.006, y muestra fluctuaciones a lo largo de las épocas, sin una tendencia clara de descenso o ascenso. Es importante notar

que estas fluctuaciones son de centésimas, lo que implica que son casi mínimas. La pérdida de validación muestra grandes picos y valles, especialmente al inicio, lo cual indica una inestabilidad en la capacidad del modelo para generalizar en diferentes épocas.

En la gráfica de la derecha, se observa la precisión de entrenamiento y validación. La precisión de entrenamiento comienza en un valor alto, 0.9972, y muestra fluctuaciones a lo largo de las épocas, similar a la pérdida. Nuevamente, estas fluctuaciones son de centésimas, indicando variaciones mínimas. La precisión de validación muestra un patrón muy variable, con grandes picos y valles, reflejando la inestabilidad observada en la pérdida.

Como se puede observar en la gráfica, a pesar de que el entrenamiento estaba previsto para 30 épocas, se ha detenido en la vuelta 19. Esto es porque en la época 9 se obtiene el menor `val_loss = 0.0041`. Además, en esta época, la pérdida de entrenamiento es `loss = 0.0056`, la precisión de validación es `val_accuracy = 0.998` y la precisión de entrenamiento vale `accuracy = 0.998`. Esto sugiere que, a pesar de las fluctuaciones observadas, el modelo alcanzó un rendimiento óptimo en esta época.

4.6.3. Evaluación del modelo

Después de entrenar el modelo, se evalúa como una clasificación binaria, usando el 15 % del conjunto de datos reservado para esta parte. Para empezar, vamos a ver la matriz de confusión 4.15 con el número de registros de la *KDD CUP 1999* que están bien clasificados y mal clasificados.

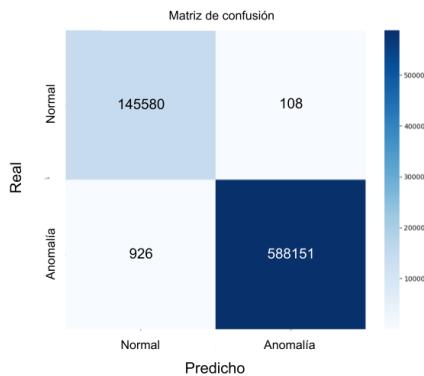


Figura 4.15: Matriz de confusión del modelo utilizando una CNN.

En ella, al igual que en los 3 modelos anteriores, se obtienen unos resultados excelentes. Veamos ahora la tabla 4.6 con el resto de métricas.

Modelo	Precisión	Sensibilidad	Especificidad	F1 Score
CNN	0.999	0.999	0.998	0.998

Cuadro 4.6: Métricas de rendimiento de la red neuronal convolucional.

En la tabla podemos volver a apreciar el gran rendimiento de este modelo con el conjunto de datos *KDD CUP 1999*. Las métricas expuestas anteriormente nos muestran unos resultados excelentes, con valores prácticamente iguales a 1. Estos nos indica que la arquitectura de CNN utilizada tiene una capacidad casi perfecta para detectar tanto registros benignos como registros anómalos.

4.7. Resultados

En este estudio, se han implementado y evaluado cuatro modelos de redes neuronales diferentes para la detección de intrusiones utilizando el conjunto de datos *KDD Cup 1999*. Los modelos considerados fueron una Red Neuronal Profunda (DNN), una Red Neuronal Recurrente (RNN), un *autoencoder* y una Red Neuronal Convolucional (CNN). A continuación, se resumen los resultados obtenidos por cada uno de estos modelos.

El modelo DNN mostró un rendimiento sobresaliente con una precisión, sensibilidad, especificidad y *F1-Score* de 0.999 en la clasificación binaria de registros benignos y anómalos. Aunque las fluctuaciones en las curvas de pérdida y precisión indicaron pequeñas variaciones, el modelo logró una excelente generalización. Los resultados detallados se presentan en la Tabla 4.7.

El modelo RNN-LSTM también obtuvo un rendimiento notable, con métricas de rendimiento casi perfectas, similar al DNN. La precisión y la pérdida se estabilizaron rápidamente, alcanzando valores de 0.999 en las métricas evaluadas. Las curvas de precisión y pérdida mostraron la mejor estabilidad, inclinándose de manera constante sin fluctuaciones notables, lo que indica una excelente capacidad de aprendizaje y generalización de las características temporales de los datos. Los resultados se detallan en la Tabla 4.7.

El *autoencoder* mostró un rendimiento excelente en la detección de intrusiones, con una precisión de 0.999 y un *F1-Score* de 0.991. Aunque su sensibilidad fue ligeramente inferior (0.93), su capacidad para reconstruir los registros benignos y detectar anomalías fue altamente efectiva. El umbral de 0.02 se utilizó para clasificar los registros, maximizando tanto la precisión como el *recall*. Los resultados se presentan en la Tabla 4.7.

Por último, la CNN también obtuvo resultados excelentes, con métricas de rendimiento cercanas a 1. Al transformar los vectores de características en matrices 11×11 , el modelo pudo aprender eficazmente las características espaciales de los datos. Los resultados indicaron una gran capacidad para detectar tanto registros benignos como anómalos, con valores muy altos en todas las métricas evaluadas, a pesar de sufrir fluctuaciones a lo largo de todo el entrenamiento (tabla 4.7).

Aunque todos los modelos obtuvieron resultados excelentes en la detección de intrusiones, el modelo RNN-LSTM destacó por su capacidad de aprender y generalizar eficientemente las características temporales de los datos. Su rendimiento consistente a lo largo de las épocas, junto con sus resultados en las métricas de precisión, sensibilidad, especificidad y *F1-Score*, lo convierten en el mejor modelo entre los evaluados. Por lo tanto, se concluye que la RNN-LSTM es la opción más efectiva para la detección de intrusiones en el conjunto de datos *KDD Cup 1999*, proporcionando resultados sobresalientes y un rendimiento superior en comparación con los otros modelos.

Modelo	Precisión	Sensibilidad	Especificidad	<i>F1 Score</i>
DNN	0.999	0.999	0.999	0.999
RNN	0.998	0.999	0.998	0.998
Autoencoder	0.999	0.93	0.999	0.991
CNN	0.999	0.998	0.999	0.999

Cuadro 4.7: Métricas de rendimiento de los diferentes modelos.

Capítulo 5

Conclusiones y Trabajo Futuro

En este capítulo, se resumen los principales resultados de este trabajo sobre la aplicación de las redes neuronales en la ciberseguridad, incluyendo una serie de direcciones futuras para posibles investigaciones en este campo.

5.1. Conclusiones

Los objetivos que se establecieron al principio del curso fueron el estudio de las redes neuronales y su aplicación para abordar problemas específicos de ciberseguridad, con un enfoque en la detección y clasificación de malware. A lo largo del trabajo, hemos logrado cumplir satisfactoriamente con estos objetivos, empezando con un estudio en profundidad del funcionamiento de las redes neuronales, sus diferentes arquitecturas y las diferentes métricas para medir su eficacia. A continuación, se puso en práctica estos conocimientos para abordar los diferentes problemas de ciberseguridad mencionados, para finalizar evaluando el rendimiento de las diferentes redes neuronales utilizadas.

Además, aunque nuestro objetivo inicial no era mejorar los modelos existentes ni realizar experimentos extra, el conocimiento adquirido durante todo el año junto con el análisis de los resultados obtenidos, ha llevado, como resultado secundario, a mejoras en la arquitectura de algunos modelos, obteniendo resultados destacables tanto en precisión como en eficiencia.

En el Capítulo 3, se realizaron experimentos extra para mejorar el rendimiento de los modelos. En la CNN se añadieron capas de `Dropout` después de las capas densas para obtener una mejor generalización del modelo. Además, se realizaron experimentos para garantizar que el uso de los hiperparámetros indicados en los artículos eran los idóneos para cada modelo. Por otro lado, en los modelos que se utilizaron *Autoencoders*, se desarrollaron y probaron diferentes modelos sin seguir investigaciones previas para este problema de clasificación. Con estas mejoras en los modelos, la CNN con capas de *dropout* de 0.5 y con `EarlyStopping` es el modelo que mejor resultados ha obtenido para este problema, por encima de los diferentes modelos de *autoencoders*, llegando a un nivel de precisión cercano al 97 % en el conjunto de prueba.

Por otro lado, en el Capítulo 4 se siguieron diferentes artículos para crear las arquitecturas de redes neuronales. Comparando sus resultados, podemos observar cómo todas ellas obtienen una precisión y una generalización óptima, con resultados superiores al 93 % en todas sus métricas estudiadas. Sin embargo, el modelo que mejor rendimiento ha demostrado a lo largo del entrenamiento y que mejor se ha adaptado a los datos de entrada ha sido la red neuronal recurrente

RNN-LSTM. Este modelo alcanzó un nivel de precisión del 99.79 % y una pérdida de 0.0045 en el conjunto de validación, resultados consistentes y prácticamente idénticos a los obtenidos en el conjunto de entrenamiento, lo que indica una excelente capacidad de generalización. Además, ha mostrado una gran estabilidad en sus curvas de precisión y pérdida a lo largo de las épocas. Todo ello la convierten en la opción más efectiva y robusta para la detección de intrusiones en comparación con los otros modelos evaluados.

En lo personal, este trabajo me ha dado la oportunidad de descubrir en profundidad el funcionamiento de las redes neuronales, un campo que siempre me había despertado curiosidad y del cual comencé a estudiar de manera autodidacta. En el segundo cuatrimestre, recibí una pequeña introducción formal en la asignatura *Geometría Computacional*, que consolidó mis conocimientos previos. Gracias a este trabajo, he podido experimentar de primera mano el gran potencial que tienen estas técnicas para abordar problemas complejos, como la ciberseguridad. Este estudio me ha permitido apreciar la importancia de aplicar el aprendizaje automático en la defensa contra ciberamenazas, destacando cómo la inteligencia artificial puede ser una herramienta crucial para mejorar la seguridad digital en un mundo cada vez más conectado.

5.2. Trabajo Futuro

A pesar de haber alcanzado los objetivos propuestos, el estudio sobre redes neuronales aplicadas a la ciberseguridad abre múltiples líneas de estudio para futuras exploraciones. A continuación, se proponen algunas de ellas:

1. **Manejo del desbalance de clases:** Como se observó en los resultados del Capítulo 3, la clase número 5 no obtuvo buenos resultados en las pruebas con las redes neuronales. Este problema de desbalanceo de clases, especialmente en la clasificación de malware, ha demostrado ser un desafío notable. Futuras estudios podrían centrarse en implementar técnicas de pesos de clase (como vimos en los resultados del capítulo 3) para abordar este desbalance.
2. **Entrenamiento Greedy de Autoencoders:** Otra ampliación interesante de este trabajo podría ser el entrenamiento de *autoencoders* de forma *greedy*, donde cada capa sea la representación comprimida de un *autoencoder* simple entrenado previamente. Según lo propuesto en [28] en el Capítulo 17, esta técnica podría adaptarse y aplicarse a nuestro contexto de ciberseguridad para mejorar la eficiencia del entrenamiento de *autoencoders*.
3. **Análisis multiclas de RNN, DNN y CNN:** Otra área de investigación futura podría ser realizar el análisis de detección de intrusiones en un contexto multiclas utilizando los modelos RNN, DNN y CNN y la base de datos *KDD CUP 1999* del capítulo 4. Este análisis permitiría evaluar la capacidad de estos modelos para clasificar distintos tipos de ataques en lugar de una simple clasificación binaria de registros benignos y anómalos.
4. **Detección de intrusiones usando RBM:** Una posible extensión de este trabajo podría ser la implementación y evaluación de un modelo utilizando Máquinas de Boltzmann Restringidas (RBM) [3] para la detección de intrusiones. Este método es una técnica de aprendizaje automático que ha demostrado ser efectiva en la modelización de datos complejos. Las RBM pueden aprender una representación probabilística de los datos y podrían ofrecer ventajas en términos de precisión y capacidad de detección de anomalías.

Bibliografía

- [1] Aakash Nain. Keras Documentation. <https://keras.io/>, (Consultado el 3 diciembre).
- [2] Yassir Acharki. Traffic signs image classification. <https://www.kaggle.com/code/yacharki/traffic-signs-image-classification-96-cnn#6.-Training-the-Model>, (Consultado el 13 de noviembre).
- [3] Khaled Alrawashdeh and Carla Purdy. Toward an online anomaly intrusion detection system based on deep learning. In *2016 15th IEEE international conference on machine learning and applications (ICMLA)*, pages 195–200. IEEE, 2016.
- [4] Pablo Amorós Becerra. Desbalanceo de datos en redes de clasificación binaria. *Escuela Politécnica Superior*, 2021.
- [5] Andrea Apicella, Francesco Donnarumma, Francesco Isgrò, and Roberto Prevete. A survey on modern trainable activation functions. *Neural Networks*, 138:14–32, 2021.
- [6] The UCI KDD Archive. Kdd cup 99. <http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html>, (Consultado el 7 de febrero).
- [7] Muhammad Arief and Suhono Harso Supangkat. Comparison of cnn and dnn performance on intrusion detection system. In *2022 International Conference on ICT for Smart Society (ICISS)*, pages 1–7. IEEE, 2022.
- [8] Dor Bank, Noam Koenigstein, and Raja Giryes. Autoencoders. *Machine learning for data science handbook: data mining and knowledge discovery handbook*, pages 353–374, 2023.
- [9] Wei Bao, Jun Yue, and Yulei Rao. A deep learning framework for financial time series using stacked autoencoders and long-short term memory. *PloS one*, 12(7):e0180944, 2017.
- [10] Yoshua Bengio, Pascal Lamblin, Dan Popovici, and Hugo Larochelle. Greedy layer-wise training of deep networks. *Advances in neural information processing systems*, 19, 2006.
- [11] Daniel S Berman, Anna L Buczak, Jeffrey S Chavis, and Cherita L Corbett. A survey of deep learning methods for cyber security. *Information*, 10(4):122, 2019.
- [12] Marcus D Bloice and Andreas Holzinger. A tutorial on machine learning and data science tools with python. *Machine Learning for Health Informatics: State-of-the-Art and Future Challenges*, pages 435–480, 2016.
- [13] Niken Dwi Wahyu Cahyani, Erwid M Jadied, Nurul Hidayah Ab Rahman, and Endro Ariyanto. The influence of virtual secure mode (vsm) on memory acquisition. *International Journal of Advanced Computer Science and Applications*, 13(11), 2022.
- [14] Natalia Caporale and Yang Dan. Spike timing-dependent plasticity: a hebbian learning rule. *Annu. Rev. Neurosci.*, 31(1):25–46, 2008.

- [15] Alex Castaño. Telepizza sufre un ciberataque de un grupo de hackers vinculado a rusia. <https://theobjective.com/economia/2023-03-24/telepizza-ciberataque-hackers-rusia/>. Consultado el 04-06-2024.
- [16] Rene Y Choi, Aaron S Coyner, Jayashree Kalpathy-Cramer, Michael F Chiang, and J Peter Campbell. Introduction to machine learning, neural networks, and deep learning. *Translational vision science & technology*, 9(2):14–14, 2020.
- [17] Keras developers. Building autoencoders in keras. <https://blog.keras.io/building-autoencoders-in-keras.html>. Consultado el 05-06-2024.
- [18] Keras Developers. Model training apis. https://keras.io/api/models/model_training_apis/. Consultado el 25-04-2024.
- [19] Keras developers. Probabilistic losses. https://keras.io/api/losses/probabilistic_losses/. Consultado el 05-06-2024.
- [20] NumPy Developers. Numpy documentation. <https://numpy.org/doc/stable/>. Consultado el 25-04-2024.
- [21] P Dileep, Dibyaiyoti Das, and Prabin Kumar Bora. Dense layer dropout based cnn architecture for automatic modulation classification. In *2020 national conference on communications (NCC)*, pages 1–5. IEEE, 2020.
- [22] Oscar R Dolling and Eduardo A Varas. Artificial neural networks for streamflow prediction. *Journal of hydraulic research*, 40(5):547–554, 2002.
- [23] Indira Kalyan Dutta, Bhaskar Ghosh, Albert Carlson, Michael Totaro, and Magdy Bayoumi. Generative adversarial networks in security: A survey. In *2020 11th IEEE Annual Ubiquitous Computing, Electronics & Mobile Communication Conference (UEMCON)*, pages 0399–0405. IEEE, 2020.
- [24] Wisam Elmasry, Akhan Akbulut, and Abdul Halim Zaim. Empirical study on multiclass classification-based network intrusion detection. *Computational Intelligence*, 35(4):919–954, 2019.
- [25] Bradley J Erickson and Felipe Kitamura. Magician’s corner: 9. performance metrics for machine learning models, 2021.
- [26] Facebook AI Research. Pytorch. <https://pytorch.org/>, (Consultado el 3 de diciembre).
- [27] Fahimeh Farahnakian and Jukka Heikkonen. A deep auto-encoder based approach for intrusion detection system. In *2018 20th International Conference on Advanced Communication Technology (ICACT)*, pages 178–183. IEEE, 2018.
- [28] Aurélien Géron. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow*. O’Reilly Media, Inc.”, 2022.
- [29] Yassine Ghouzam. Introduction to cnn keras. <https://www.kaggle.com/code/yassineghouzam/introduction-to-cnn-keras-0-997-top-6/notebook>, (Consultado el 10 de noviembre).
- [30] Daniel Gibert, Jordi Planes, Carles Mateu, and Quan Le. Fusing feature engineering and deep learning: A case study for malware classification. *Expert Systems with Applications*, 207:117957, 2022.
- [31] Google AI Team. Keras. <https://keras.io/>, (Consultado el 3 diciembre).

- [32] Google Brain Team. Tensorflow. <https://www.tensorflow.org/>, 2015.
- [33] Muni Prashneel Gounder and Mohammed Farik. New ways to fight malware. *Int. J. Sci. Technol. Res*, 6(06), 2017.
- [34] Xifeng Guo, Xinwang Liu, En Zhu, and Jianping Yin. Deep clustering with convolutional autoencoders. In *Neural Information Processing: 24th International Conference, ICONIP 2017, Guangzhou, China, November 14-18, 2017, Proceedings, Part II 24*, pages 373–382. Springer, 2017.
- [35] Sanchit Gupta, Harshit Sharma, and Sarvejeet Kaur. Malware characterization using windows api call sequences. In *Security, Privacy, and Applied Cryptography Engineering: 6th International Conference, SPACE 2016, Hyderabad, India, December 14-18, 2016, Proceedings 6*, pages 271–280. Springer, 2016.
- [36] Ke He and Dong-Seong Kim. Malware detection with malware images using deep learning techniques. In *2019 18th IEEE international conference on trust, security and privacy in computing and communications/13th IEEE international conference on big data science and engineering (TrustCom/BigDataSE)*, pages 95–102. IEEE, 2019.
- [37] Geoffrey E Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.
- [38] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.
- [39] Md Anwar Hossain and Md Shahriar Alam Sajib. Classification of image using convolutional neural network (cnn). *Global Journal of Computer Science and Technology*, 19(2):13–14, 2019.
- [40] Yen-Hung Frank Hu, Abdinur Ali, Chung-Chu George Hsieh, and Aurelia Williams. Machine learning techniques for classifying malicious api calls and n-grams in kaggle data-set. In *2019 SoutheastCon*, pages 1–8. IEEE, 2019.
- [41] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.
- [42] Daniel Ollero J. La comunitat sufre un ciberataque que expone información de miles de alumnos: "te aconsejamos la modificación de todas tus contraseñas". <https://www.elmundo.es/madrid/2024/05/10/663e4244e9cf4a2e3d8b4599.html>. Consultado el 04-06-2024.
- [43] Mahmoud Kalash, Mrigank Rochan, Noman Mohammed, Neil DB Bruce, Yang Wang, and Farkhund Iqbal. Malware classification with deep convolutional neural networks. In *2018 9th IFIP international conference on new technologies, mobility and security (NTMS)*, pages 1–5. IEEE, 2018.
- [44] Kavya. Auto encoder — implementation. <https://medium.com/@kavya.sa1996/auto-encoder-implementation-682be1d7e68c>. Consultado el 25-05-2024.
- [45] Şafak Kayıkçı. Convolutional autoencoder model for reproducing fingerprint. *ICATCES 2020 Proceeding Book*, page 35, 2020.
- [46] Temesguen Messay Kebede, Ouboti Djaneye-Boundjou, Barath Narayanan Narayanan, Anca Ralescu, and David Kapp. Classification of malware programs using autoencoders based deep learning architecture and its application to the microsoft malware classification

- challenge (big 2015) dataset. In *2017 IEEE National Aerospace and Electronics Conference (NAECON)*, pages 70–75. IEEE, 2017.
- [47] Max Kerkers, José Jair Santanna, and Anna Sperotto. Characterisation of the kelihos. b botnet. In *Monitoring and Securing Virtualized Networks and Services: 8th IFIP WG 6.6 International Conference on Autonomous Infrastructure, Management, and Security, AIMS 2014, Brno, Czech Republic, June 30–July 3, 2014. Proceedings 8*, pages 79–91. Springer, 2014.
 - [48] Jihyun Kim, Jaehyun Kim, Huong Le Thi Thu, and Howon Kim. Long short term memory recurrent neural network classifier for intrusion detection. In *2016 international conference on platform technology and service (PlatCon)*, pages 1–5. IEEE, 2016.
 - [49] Jiyeon Kim, Jiwon Kim, Hyunjung Kim, Minsun Shim, and Eunjung Choi. Cnn-based network intrusion detection against denial-of-service attacks. *Electronics*, 9(6):916, 2020.
 - [50] Sera Kim and Seok-Pil Lee. A bilstm–transformer and 2d cnn architecture for emotion recognition from speech. *Electronics*, 12(19):4034, 2023.
 - [51] Taejoon Kim, Sang C Suh, Hyunjoo Kim, Jonghyun Kim, and Jinoh Kim. An encoding technique for cnn-based network anomaly detection. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 2960–2965. IEEE, 2018.
 - [52] Donald E Knuth. *The Art of Computer Programming: Fundamental Algorithms, Volume 1*. Addison-Wesley Professional, 1997.
 - [53] Sushil Kumar et al. Mcft-cnn: Malware classification with fine-tune convolution neural networks using traditional and transfer learning in internet of things. *Future Generation Computer Systems*, 125:334–351, 2021.
 - [54] Runze Lin. Analysis on the selection of the appropriate batch size in cnn neural network. In *2022 International Conference on Machine Learning and Knowledge Engineering (MLKE)*, pages 106–109. IEEE, 2022.
 - [55] Hongyu Liu and Bo Lang. Machine learning and deep learning methods for intrusion detection systems: A survey. *applied sciences*, 9(20):4396, 2019.
 - [56] Ivandro O Lopes, Deqing Zou, Ihsan H Abdulqadder, Francis A Ruambo, Bin Yuan, and Hai Jin. Effective network intrusion detection via representation learning: A denoising autoencoder approach. *Computer Communications*, 194:55–65, 2022.
 - [57] Belen Lopez and Antonio Alcaide. Blockchain, artificial intelligence, internet of things to improve governance, financial management and control of crisis: Case study covid-19. *SocioEconomic Challenges*, 4:78–89, 01 2020.
 - [58] Mika Luoma-aho. Analysis of modern malware: obfuscation techniques. *JAMK University of Applied Sciences*, 2023.
 - [59] Andrew L Maas, Awni Y Hannun, Andrew Y Ng, et al. Rectifier nonlinearities improve neural network acoustic models. In *Proc. icml*, volume 30, page 3. Atlanta, GA, 2013.
 - [60] Nesma Mahmoud, Youssef Essam, Radwa Elshawi, and Sherif Sakr. Dlbench: an experimental evaluation of deep learning frameworks. In *2019 IEEE International Congress on Big Data (BigDataCongress)*, pages 149–156. IEEE, 2019.
 - [61] Mohammed Maithem and Ghadaa A Al-Sultany. Network intrusion detection system using deep neural networks. In *Journal of Physics: Conference Series*, volume 1804, page 012138. IOP Publishing, 2021.

- [62] Rosa Martínez Álvarez-Castellanos et al. Análisis de las máquinas sparse autoencoders como extractores de características. *Universidad Politécnica de Cartagena*, 2017.
- [63] matplotlib Developers. Matplotlib: Visualization with python. <https://matplotlib.org/>. Consultado el 25-04-2024.
- [64] Matthew Carrigan. Keras Documentation. <https://keras.io/>, (Consultado el 3 de diciembre).
- [65] Microsoft. Microsoft malware classification challenge (big 2015). <https://www.kaggle.com/c/malware-classification>, (Consultado el 30 de octubre).
- [66] Robert Monjo. Geometría computacional. https://cvdof.ucm.es/moodle/pluginfile.php/995348/mod_resource/content/3/GCOM_Teoria.pdf. Consultado el 05-06-2024. Acceso restringido a alumnos matriculados en la asignatura.
- [67] James Moor. The dartmouth college artificial intelligence conference: The next fifty years. *Ai Magazine*, 27(4):87–87, 2006.
- [68] Andreas C Müller and Sarah Guido. *Introduction to machine learning with Python: a guide for data scientists*. .O'Reilly Media, Inc.", 2016.
- [69] Mohammad Najafimehr, Sajjad Zarifzadeh, and Seyedakbar Mostafavi. A hybrid machine learning approach for detecting unprecedented ddos attacks. *The Journal of Supercomputing*, 78, 04 2022.
- [70] Barath Narayanan Narayanan, Ouboti Djaneye-Boundjou, and Temesguen M Kebede. Performance analysis of machine learning and pattern recognition algorithms for malware classification. In *2016 IEEE national aerospace and electronics conference (NAECON) and ohio innovation summit (OIS)*, pages 338–342. IEEE, 2016.
- [71] Lakshmanan Nataraj, Shanmugavadiel Karthikeyan, and BS Manjunath. Sattva: Sparsity inspired classification of malware variants. In *Proceedings of the 3rd ACM Workshop on Information Hiding and Multimedia Security*, pages 135–140, 2015.
- [72] Lakshmanan Nataraj, Sreejith Karthikeyan, Gregoire Jacob, and Bangalore S Manjunath. Malware images: visualization and automatic classification. In *Proceedings of the 8th international symposium on visualization for cyber security*, pages 1–7, 2011.
- [73] Sinh-Ngoc Nguyen, Van-Quyet Nguyen, Jintae Choi, and Kyungbaek Kim. Design and implementation of intrusion detection system using convolutional neural network for dos detection. In *Proceedings of the 2nd international conference on machine learning and soft computing*, pages 34–38, 2018.
- [74] Gonzalo Pajares Martinsanz et al. *Aprendizaje profundo*. Biblioteca Hernán Malo González, 2021.
- [75] Marek Pawlicki, Rafał Kozik, and Michał Choraś. A survey on neural networks for (cyber-) security and (cyber-) security of neural networks. *Neurocomputing*, 500:1075–1087, 2022.
- [76] Van Hiep Phung and Eun Joo Rhee. A deep learning approach for classification of cloud image patches on small datasets. *Journal of information and communication convergence engineering*, 16(3):173–178, 2018.
- [77] Prajjoy Podder, Subrato Bharati, M Mondal, Pinto Kumar Paul, and Utku Kose. Artificial neural network for cybersecurity: A comprehensive review. *arXiv preprint arXiv:2107.01185*, 2021.

- [78] Pablo Jiménez Poyatos. Repositorio github para el tfg. <https://github.com/pabloojp/TFG>, 2024.
- [79] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.
- [80] scikit-learn Developers. ssikit-llarn documentation. <https://scikit-learn.org/stable/>. Consultado el 25-04-2024.
- [81] scikit-learn Developers. Train test split de scikit-learn. https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html. Consultado el 25-04-2024.
- [82] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.
- [83] Nitish Srivastava. Improving neural networks with dropout. *University of Toronto*, 182(566):7, 2013.
- [84] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.
- [85] Sajedul Talukder. Tools and techniques for malware detection and analysis. *arXiv preprint arXiv:2002.06819*, 2020.
- [86] Mingdong Tang and Quan Qian. Dynamic api call sequence visualisation for malware classification. *IET Information Security*, 13(4):367–377, 2019.
- [87] Yingjie Tian and Yuqi Zhang. A comprehensive survey on regularization strategies in machine learning. *Information Fusion*, 80:146–166, 2022.
- [88] Jordi Torres. Red neuronal recurrente simepl. <https://torres.ai/redes-neuronales-recurrentes/>. Consultado el 04-05-2024.
- [89] Chih-Fong Tsai, Yu-Feng Hsu, Chia-Ying Lin, and Wei-Yang Lin. Intrusion detection by machine learning: A review. *expert systems with applications*, 36(10):11994–12000, 2009.
- [90] Rahul K Vigneswaran, R Vinayakumar, KP Soman, and Prabaharan Poornachandran. Evaluating shallow and deep neural networks for network intrusion detection systems in cyber security. In *2018 9th International conference on computing, communication and networking technologies (ICCCNT)*, pages 1–6. IEEE, 2018.
- [91] Jin Wang, Zhongyuan Wang, Dawei Zhang, and Jun Yan. Combining knowledge with deep convolutional neural networks for short text classification. In *IJCAI*, volume 350, pages 3172077–3172295, 2017.
- [92] la enciclopedia libre Wikipedia. Perceptrón. <https://es.wikipedia.org/wiki/Perceptr%C3%B3n>. Consultado el 04-05-2024.
- [93] Xiaofei Xing, Xiang Jin, Haroon Elahi, Hai Jiang, and Guojun Wang. A malware detection approach using autoencoder in deep learning. *IEEE Access*, 10:25696–25706, 2022.
- [94] Jin Xu, Zishan Li, Bowen Du, Miaomiao Zhang, and Jing Liu. Reluplex made more practical: Leaky relu. In *2020 IEEE Symposium on Computers and communications (ISCC)*, pages 1–7. IEEE, 2020.

- [95] Wen Xu, Julian Jang-Jaccard, Amardeep Singh, Yuanyuan Wei, and Fariza Sabrina. Improving performance of autoencoder-based network anomaly detection on nsl-kdd dataset. *IEEE Access*, 9:140136–140146, 2021.
- [96] Zhongxue Yang and Adem Karahoca. An anomaly intrusion detection approach using cellular neural networks. In *Computer and Information Sciences-ISCIS 2006: 21th International Symposium, Istanbul, Turkey, November 1-3, 2006. Proceedings 21*, pages 908–917. Springer, 2006.
- [97] Ruizhe Yao, Ning Wang, Zhihui Liu, Peng Chen, and Xianjun Sheng. Intrusion detection system in the advanced metering infrastructure: a cross-layer feature-fusion cnn-lstm-based approach. *Sensors*, 21(2):626, 2021.
- [98] Juan Yepez and Seok-Bum Ko. Stride 2 1-d, 2-d, and 3-d winograd for convolutional neural networks. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 28(4):853–863, 2020.
- [99] Chuanlong Yin, Yuefei Zhu, Jinlong Fei, and Xinzheng He. A deep learning approach for intrusion detection using recurrent neural networks. *Ieee Access*, 5:21954–21961, 2017.
- [100] Ilsun You and Kangbin Yim. Malware obfuscation techniques: A brief survey. In *2010 International conference on broadband, wireless computing, communication and applications*, pages 297–300. IEEE, 2010.
- [101] Junhai Zhai, Sufang Zhang, Junfen Chen, and Qiang He. Autoencoder and its various variants. In *2018 IEEE international conference on systems, man, and cybernetics (SMC)*, pages 415–419. IEEE, 2018.
- [102] Min-Ling Zhang and Zhi-Hua Zhou. A review on multi-label learning algorithms. *IEEE transactions on knowledge and data engineering*, 26(8):1819–1837, 2013.
- [103] Mohamad Fadli Zolkipli and Aman Jantan. Malware behavior analysis: Learning and understanding current malware threats. In *2010 Second International Conference on Network Applications, Protocols and Services*, pages 218–221. IEEE, 2010.
- [104] Manuel Ángel Méndez. Banco santander sufre un ciberataque que afecta a datos de clientes en españa. https://www.elconfidencial.com/tecnologia/2024-05-14/santander-acceso-no-autorizado-base-datos-clientes_3883386/. Consultado el 04-06-2024.

