

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

**Resumen**

***Palabras clave:***

**Abstract**

***Keywords:***



# Índice general

<b>1. Introducción</b>	<b>1</b>
1.1. Motivación y objetivos del trabajo . . . . .	1
1.2. Contexto y antecedentes del trabajo . . . . .	1
1.2.1. Redes neuronales . . . . .	1
1.2.2. Importancia de la detección y prevención de ataques . . . . .	1
1.2.3. Evolución de las amenazas cibernéticas . . . . .	1
1.2.4. Avances en el aprendizaje automático para ciberseguridad . . . . .	1
1.3. Metodología . . . . .	2
1.4. Estructura de la memoria . . . . .	2
<b>2. Fundamentos de las redes neuronales</b>	<b>3</b>
2.1. Revisión teórica . . . . .	3
2.2. Arquitecturas relevantes . . . . .	3
2.2.1. Autoencoder . . . . .	3
2.2.2. Deep Belief Networks . . . . .	5
2.2.3. Red Neuronal Convolutiva . . . . .	5
2.2.4. Red Neuronal Recurrente . . . . .	8
2.3. Bibliotecas utilizadas en Python . . . . .	8
2.3.1. Principales frameworks. Keras . . . . .	8
2.3.2. Librerías y herramientas esenciales. . . . .	9
<b>3. Clasificación de Malware</b>	<b>11</b>
3.1. Microsoft Malware Classification Challenge . . . . .	11

3.1.1. Distribución del dataset . . . . .	13
3.2. Red Neuronal Convolutacional . . . . .	15
3.2.1. Visualizar el malware como imagen . . . . .	15
3.2.2. Visualización del modelo . . . . .	16
3.2.3. Mejora del modelo . . . . .	17
3.2.4. GPU (Unidad de Procesamiento Gráfico) . . . . .	18
3.2.5. CPU (Unidad Central de Procesamiento) . . . . .	18
3.3. Autoencoder . . . . .	19
3.4. Resultados . . . . .	19
<b>4. Detección de intrusiones</b>	<b>21</b>
4.1. KDD Cup 1999 . . . . .	21
4.2. Autoencoder . . . . .	21
4.3. Red Neuronal Convolutacional . . . . .	21
4.4. Red Neuronal Profunda . . . . .	22
4.5. Red Neuronal Recurrente . . . . .	22
4.6. Restricted Boltzmann Machine . . . . .	22
4.7. Resultados . . . . .	22
<b>5. Conclusiones y Trabajo Futuro</b>	<b>23</b>
5.1. Conclusiones . . . . .	23
5.2. Trabajo futuro . . . . .	23
<b>Bibliografía</b>	<b>25</b>
.1. Anexo A . . . . .	30

# Capítulo 1

## Introducción

### 1.1. Motivación y objetivos del trabajo

### 1.2. Contexto y antecedentes del trabajo

#### 1.2.1. Redes neuronales

#### 1.2.2. Importancia de la detección y prevención de ataques

Destaca la importancia crítica de la detección y prevención de ataques cibernéticos en entornos empresariales y gubernamentales, así como en la protección de datos sensibles y la infraestructura crítica.

#### 1.2.3. Evolución de las amenazas cibernéticas

Describe brevemente cómo han evolucionado las amenazas en el ámbito de la ciberseguridad a lo largo del tiempo, desde virus simples hasta ataques sofisticados como el ransomware y el phishing.

#### 1.2.4. Avances en el aprendizaje automático para ciberseguridad

Proporciona una visión general de cómo los algoritmos de aprendizaje automático han revolucionado el campo de la ciberseguridad, permitiendo la detección temprana de amenazas, el análisis de comportamiento anómalo y la automatización de respuestas.

### 1.3. Metodología

### 1.4. Estructura de la memoria

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.



## Capítulo 2

# Fundamentos de las redes neuronales

- Supervisado y no supervisado
- one-hot encoder
- validacion cruzada
- dropout, l2
- optiizadores
- arquitectuas
- metricas

en la pagina 458 de hands aparece la arquitectura mía

### 2.1. Revisión teórica

Puedo introducir los tipos de funciones de activavion. Esta bien explicado en el TFG wuolah o en el articulo de KDD cup 199 de DNN network intrusion. Puedo añadir overfitting y underfitting. lo que es aprendizaje supervisado y no supervisao Partes de una neurona y como trabaja(bias, pesos...)

### 2.2. Arquitecturas relevantes

Mini tabla resumen en Deep Cybersecurity: A Comprehensive Overview from Neural Network and Deep Learning Perspective y miniresumen de todos los tipos en review Deep Cybersecurity: A Comprehensive Overview from Neural Networkand Deep Learning Perspective y review

#### 2.2.1. 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 [34]. La arquitectura

básica de un autoencoder consta de tres partes: el encoder, el cuello de bottella y el decoder (Figura 2.1).

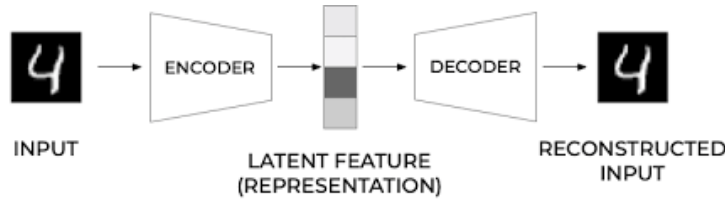


Figura 2.1: Arquitectura de un autoencoder. Fuente:[26].

El encoder mapea los datos de entrada a una representación oculta de menor dimensión utilizando funciones principalmente no lineales, mientras que el decoder reconstruye los datos de entrada a partir de esta representación oculta. 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 pérdida de entropía cruzada. Esto concluye el proceso de entrenamiento de un autoencoder.

El problema, tal como se define formalmente en [4], es aprender las funciones

$$A : \mathbb{R}^n \rightarrow \mathbb{R}^p \quad (\text{encoder})$$

y

$$B : \mathbb{R}^p \rightarrow \mathbb{R}^n \quad (\text{decoder})$$

que satisfacen

$$\arg \min_{A,B} \mathbb{E}[\Delta(x, B \circ A(x))],$$

donde  $\mathbb{E}$  es la esperanza sobre la distribución de  $x$ , y  $\Delta$  es la función de pérdida de reconstrucción, que mide la distancia entre la salida del decodificador y la entrada.

Las ecuaciones para obtener la salida de un autoencoder serían:

$$\begin{cases} z^{(1)} = W^{(1)} \cdot x + b^{(1)} \\ a^{(2)} = f(z^{(1)}) \\ z^{(2)} = W^{(2)} \cdot a^{(2)} + b^{(2)} \\ y = z^{(2)} \end{cases}$$

donde  $x$  es el input,  $b^{(1)}$  y  $b^{(2)}$  son los sesgos,  $W^{(1)}$  y  $W^{(2)}$  son los pesos,  $z^{(1)}$  es la salida lineal de la primera capa,  $a^{(2)}$  es la activación de la segunda capa,  $z^{(2)}$  es la salida lineal de la segunda capa e  $y$  es la salida final del modelo [38].

Los autoencoders se utilizan en una amplia variedad de aplicaciones, incluida la reducción de dimensionalidad, la extracción de características, la eliminación de ruido en los datos de entrada y la detección de anomalías. Su versatilidad y capacidad para aprender representaciones útiles de los datos los hacen herramientas poderosas.

Las principales capas que se utilizan en esta red neuronal son las capas densas y las de aplanamiento, aunque también se pueden utilizar capas convolucionales y de pooling para autocodificadores convolucionales<sup>1</sup> o LSTM en el caso de autocodificadores recurrentes<sup>2</sup>[17].

<sup>1</sup>Autocodificador para imágenes de gran tamaño

<sup>2</sup>Autocodificador específico para series temporales o secuencias.

### 2.2.2. Deep Belief Networks

#### Red Neuronal Profunda

### 2.2.3. 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. Son una parte de las redes profundas que procesa y analiza entradas de imágenes visuales, y están compuestas por neuronas con pesos y sesgos que aprenden a lo largo de su entrenamiento [50]. La arquitectura de una CNN (Figura 2.2) consta de tres tipos de capas: capas de convolución, capas de pooling y la capa de clasificación.

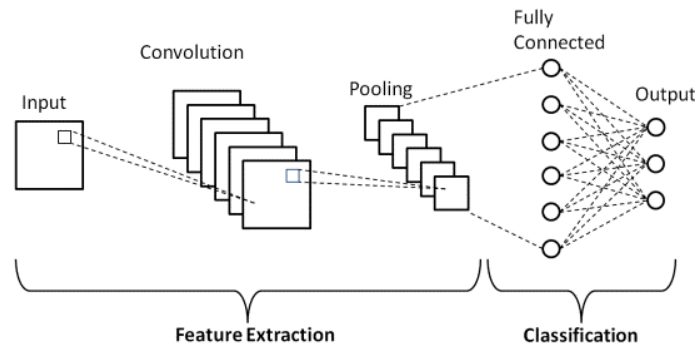


Figura 2.2: Arquitectura de una CNN con capas de convolución, pooling y clasificación. Fuente:[49].

#### Capas convolucionales

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  $\int(x \star w)(t)$  donde  $x$  representa la entrada y  $w$  el núcleo de convolución [48].

En una CNN, la entrada de la convolución es una matriz multidimensional mientras que  $w$  es una matriz de parámetros, llamada núcleo o filtro, que se ajusta durante el aprendizaje. Cada píxel de la capa de convolución tiene una neurona, que se conecta con la capa anterior aplicando la convolución con las neuronas de su campo receptivo<sup>3</sup> [17]. Esta convolución se realiza con un solapamiento total del filtro, lo que resulta en una imagen de menor dimensión (Figura 2.3a). Si se desea mantener la misma dimensión, se puede aplicar zero-padding, que consiste en rellenar con ceros la matriz para obtener las dimensiones deseadas (Figura 2.3b). En la figura 2.3 se muestran sendos campos receptivos de la imagen  $I$  que contribuyen a las salidas  $P$  y  $Q$  generadas por el filtro  $K$ . La matriz de respuesta al aplicarle el kernel se llama mapa de características.

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.

<sup>3</sup>Región de entrada que contribuye a la salida generada por el filtro

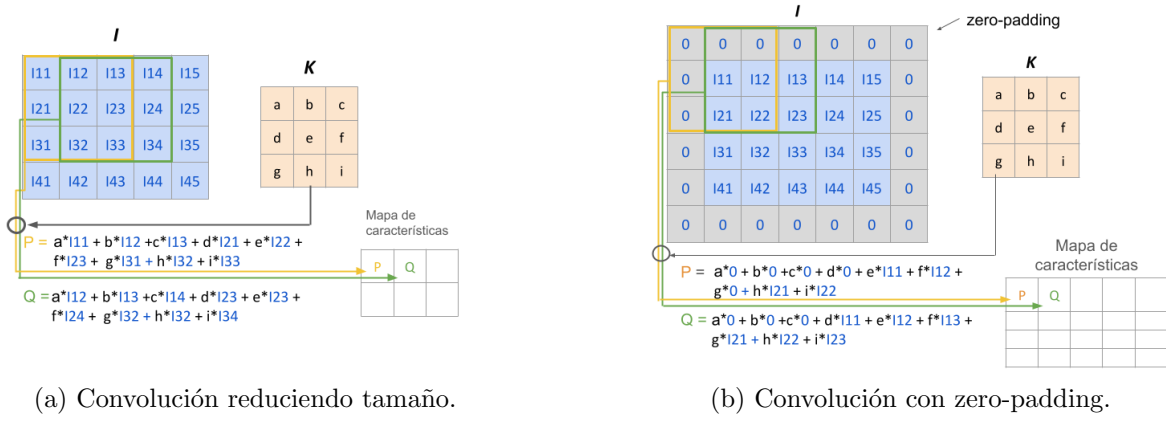


Figura 2.3: Convolución en 2D.

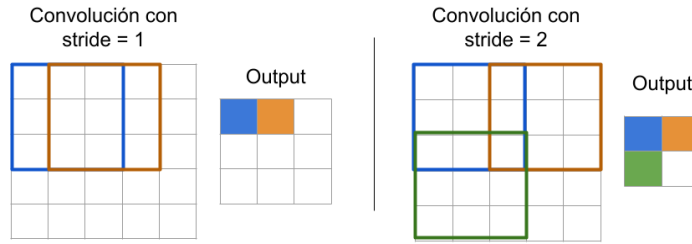


Figura 2.4: Campos receptivos en una convolución. Adaptación de imagen de [60]

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 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 [60].

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 pixel 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 [17].

Toda la información anterior se resume en la siguiente ecuación [48]:

$$z_{i,j,k} = b_k + \sum_{u=0}^{f_h-1} \sum_{v=0}^{f_w-1} \sum_{k'=0}^{f_n'-1} x_{i',j',k'} \cdot w_{u,v,k',k} \quad \text{con} \quad \begin{cases} i' = i \cdot s_h + u \\ j' = j \cdot s_w + v \end{cases} \quad (2.1)$$

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  $k'$ .

### Capas 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 [50]. 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 agregación<sup>4</sup>. Las tres funciones más comunes son el promedio, la suma y el máximo. La Figura 2.5 muestra una capa de max pooling, que es el tipo más común [17].

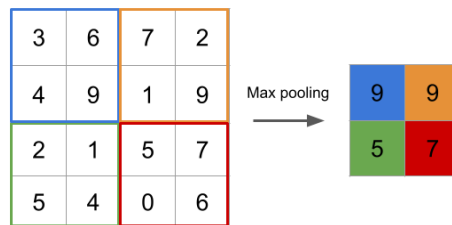


Figura 2.5: Capa de max pooling con un kernel de  $2 \times 2$ , stride 2 y sin padding.

Además de reducir el número de operaciones, el número de parámetros y ayudar con el overfitting, una capa de max pooling introduce cierto nivel de invarianza a pequeñas translaciones, ya que si un pixel se traslada hacia la derecha, la salida también debería trasladarse un pixel hacia la derecha, como se ilustra en la Figura 2.6. Esto significa que pequeñas variaciones en la posición de las características dentro de la imagen no afectan significativamente la salida.

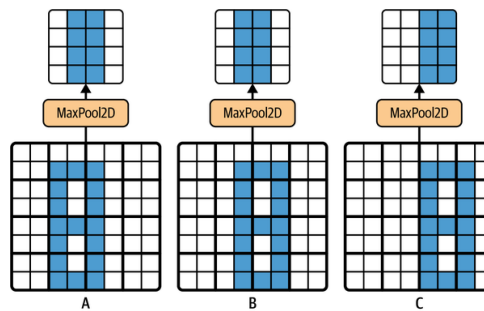


Figura 2.6: Invarianza a translaciones pequeñas mediante una capa de max pooling. Fuente [17]

### 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

<sup>4</sup>Las funciones de agregación devuelven un valor único de un conjunto de registros.

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. Este diseño permite a las CNN manejar datos complejos y variados, aprovechando la jerarquía de características aprendidas durante el entrenamiento. Este tipo de capa suele ir seguido de una capa de Dropout que mejora la capacidad de generalización del modelo al prevenir el sobreajuste, un problema común en el ámbito del aprendizaje profundo [22].

#### 2.2.4. Red Neuronal Recurrente

#### Restricted Boltzmann Machine

### 2.3. 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.

#### 2.3.1. Principales frameworks. Keras

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** [20] 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** [19] 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 [42]. 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** [15], 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 [36]. Esto, junto con su capacidad de auto-diferenciación en modo inverso<sup>5</sup>, 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 [36]. En él se hace un estudio de eficiencia, convergencia, tiempo de entrenamiento y uso de memoria de los diferentes frameworks con varios datasets. Entre sus resultados podemos observar como Keras 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 en vez de PyTorch o TensorFlow en nuestros estudios posteriores. Aakash Nain resume perfectamente las ventajas de Keras [2] 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 [40], 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.”

### 2.3.2. 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** [51] 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()` [52]. Esta función divide el dataset en dos subconjuntos de forma aleatoria, manteniendo la correspondencia en caso de que el dataset 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

---

<sup>5</sup>Técnica en la que PyTorch calcula automáticamente las derivadas de las funciones de pérdida con respecto a los parámetros del modelo.

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** [11] 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 [8]. En NumPy, se pueden crear arrays y realizar operaciones rápidas y eficientes sobre ellos. Se utilizarán estas estructuras de datos para almacenar los datos y entrenar las redes neuronales con ellas. Aunque también se pueden utilizar tensores [10], 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** [39]. 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 anti-malware 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 [1].

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 encontrados en computadoras para detectarlos como maliciosos y asociarlos a una familia específica.

Para enfrentar este tipo de problema, se va a escoger una de las bases de datos disponibles en [50] 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 como base de datos Microsoft Malware Classification Challenge. La principal razón de esta decisión ha sido que con este dataset tenemos a nuestra disposición dos algoritmos diferentes de machine learning que están referenciados en este review y que se aborda el problema usando cada uno su propio enfoque.

### 3.1. Microsoft Malware Classification Challenge

El conjunto de datos utilizado en este estudio proviene del Microsoft Malware Classification Challenge (BIG 2015) [1], 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 [50].

Se puede descargar desde su página web [1]. Tiene un tamaño de 0.5 TB sin comprimir. Para poder manipularla en mi ordenador, tuve que seguir los siguientes pasos. Primero, me descargué la carpeta comprimida (7z) con todo el dataset. Después, la subí al servidor Simba de la facultad de informática y finalmente, usando el comando `7zz x file_name.7z`, la descomprimí.

Este dataset contiene 5 archivos:

- dataSample.7z - Carpeta comprimida(7z) con una muestra de los datos disponibles.
- train.7z - Carpeta comprimida(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 comprimida 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 bytes<sup>1</sup> 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.

DIRECCIÓN MEMORIA		REPRESENTACIÓN HEXADECIMAL															
28232	0046F470	E0	01	EC	10	4C	01	62	00	EC	00	82	11	06	11	84	01
28233	0046F480	A8	11	00	10	EE	00	AE	01	42	10	20	11	C2	00	A0	10
28234	0046F490	CC	10	4A	01	42	00	EE	01	AA	00	44	00	84	10	0C	01
28235	0046F4A0	24	11	A8	10	AC	01	AE	11	0E	01	80	10	6A	11	6A	10
28236	0046F4B0	4E	01	82	01	00	01	AE	01	0E	11	E2	11	0A	10	2A	01
28237	0046F4C0	60	01	C8	00	E8	10	28	01	04	00	82	00	62	10	E4	01
28238	0046F4D0	EA	00	CE	01	A6	01	46	11	0C	00	00	00	??	??	??	??
28239	0046F4E0	??	??	??	??	??	??	??	??	??	??	??	??	??	??	??	??

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 [9].

<sup>1</sup>Realmente 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 [23], podemos definirlos 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.

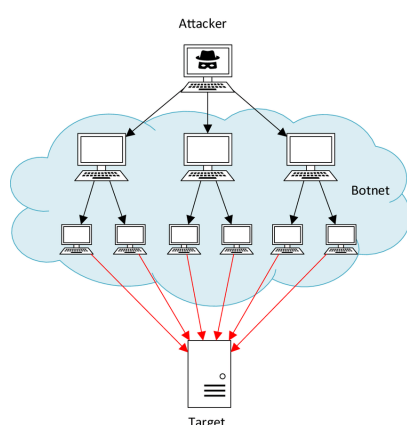


Figura 3.2: Estructura de un botnet. Imagen sacada de [43].

2. **Lollipop** es un tipo de programa adware<sup>2</sup> 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 monitoreando 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<sup>3</sup> que infecta ordenadores descargando y ejecutando archivos arbitrarios que pueden incluir malware adicional. Los ordenadores infectados pasan 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)<sup>4</sup> 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 [28]: controladores (operados por los dueños y donde se crean las instrucciones), enrutadores (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 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.

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

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

<sup>4</sup>Una 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 restantes<sup>5</sup> (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).

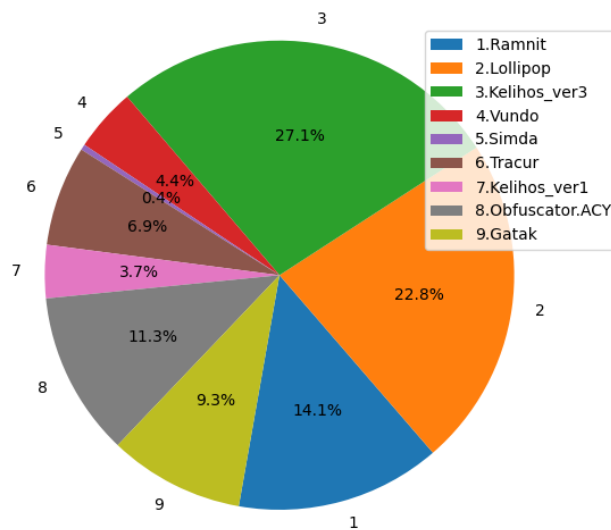


Figura 3.3: Distribución del BIG 2015 training dataset.

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 [27] 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

<sup>5</sup>Los identificadores de estos archivos son 58kxhXouHzFd4g3rmInB, 6tfw0xSL2FNHOCJBdlaA, a9oIzfw03ED4lTBCt52Y, 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 Convolutional Neural Network (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 Convolutiva

Para abordar este problema utilizando como modelo una CNN, solo podemos utilizar los datos etiquetados ya que este método es un método supervisado. De los 21.741 programas de malware con los que disponemos, solo podemos utilizar los 10.860 que están etiquetados y contienen información disponible. Como ya vimos en la sección 2.2.3, 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 [45], pero si lo representamos de forma matricial, estos pequeños cambios pueden ser detectados fácilmente [25].

### 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 pixel en la imagen. Inspirado en [46], para pasar de código hexadecimal a imagen primero pasamos cada byte a su número decimal correspondiente que se encuentra en el rango  $[0,255]$ <sup>6</sup>. 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 [17], se plantea eliminar estos caracteres y tratar el resto de bytes. Otra solución la proponen Narayanan et al. [44], que es sustituir estos bytes por el valor -1 (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.

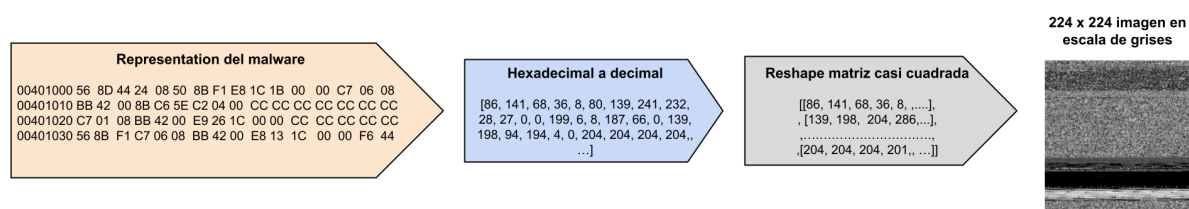


Figura 3.4: Proceso de visualización del malware. Adaptación de [46]

Después de tener todos los bytes en formato decimal dentro de un vector, 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 [32]. Para ello, siguiendo el ejemplo de [25], decidimos escoger como tamaño  $224 \times 224$ . Para obtener estas dimensiones, Simonyan et al. [53] deciden recortar aleatoriamente un cuadrado de la imagen de tamaño  $224 \times 224$ , pero nosotros hemos decidido usar interpolación lineal. En la figura 3.5, se puede observar cuales son los patrones que sigue cada tipo de malware en su forma matricial.

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

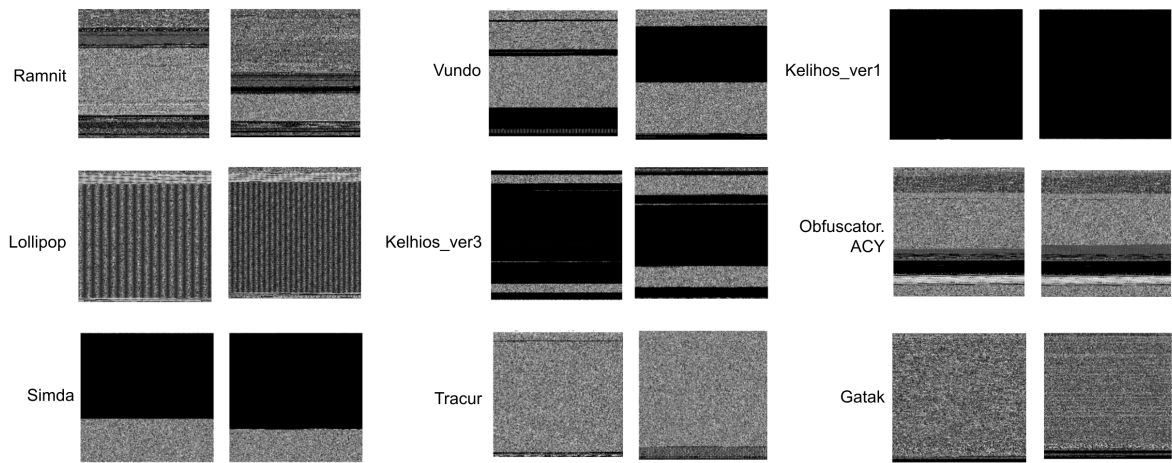


Figura 3.5: Visualización familias malware.

### 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 [25], en el que se crea una M-CNN (malware CNN) con múltiples capas. Vemos su arquitectura en la Figura 3.6.

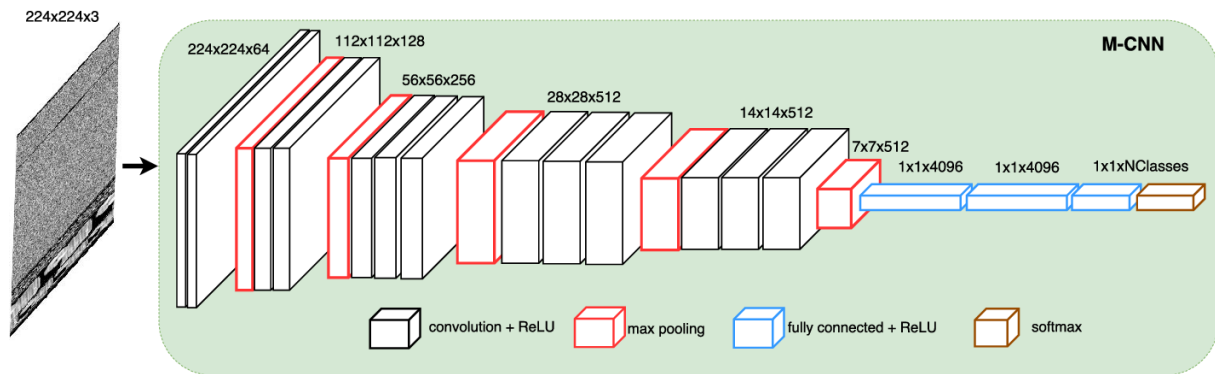


Figura 3.6: Arquitectura de la CNN. Fuente [25].

En las capas iniciales, se aplican filtros (kernels) de tamaño  $3 \times 3$  o  $5 \times 5$ , 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[17]. Después de cada operación de convolución, se aplica la función de activación ReLU para introducir no linealidades en el modelo.

Las capas de max pooling reducen la dimensionalidad de los mapas de características seleccionando el valor máximo en sub-regiones de  $2 \times 2$ , 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[17].

La secuencia de capas convolucionales y de pooling se repite varias veces hasta obtener un conjunto final de características que es una matriz de  $7 \times 7$  con 512 mapas de características. Esta salida se aplanar 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. Cabe recordar que los datos fueron divididos en tres conjuntos: entrenamiento (75 %), validación (10 %) y prueba (15 %).

Para la compilación del modelo, utilizamos el optimizador SGD (Stochastic Gradient Descent) con un learning rate inicial de 0.001. Este learning rate se reduce en un factor de 10 cada 20 epochs. Además, fijamos el momentum en 0.9 y el weight decay en 0.0005, siguiendo las recomendaciones de [25]. La función de pérdida utilizada es categorical\_crossentropy ya que nuestro problema involucra múltiples clases de etiquetas [? ].

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, y registrar el progreso del entrenamiento. Además, se barajea el conjunto de datos de entrenamiento antes de cada epoch.

### 3.2.3. Mejora del modelo

Este modelo obtiene un accuracy bastante bueno 0.9613, sin embargo, si lo comparamos con el éxito que obtienen los datos de entrenamiento, 1.0, vemos que hay una gran diferencia. Cuando evaluamos la función de pérdida ocurre lo mismo, el loss que obtiene los datos de validación son de 0.331 mientras que los datos de entrenamiento obtienen un 0.0005. Estos datos nos sugieren que se puede estar produciendo un sobreajuste de los datos de aprendizaje, lo que evita la generalización. Para salir vemos gráficamente en la Figura 3.7 su evolución a lo largo de las epochs.

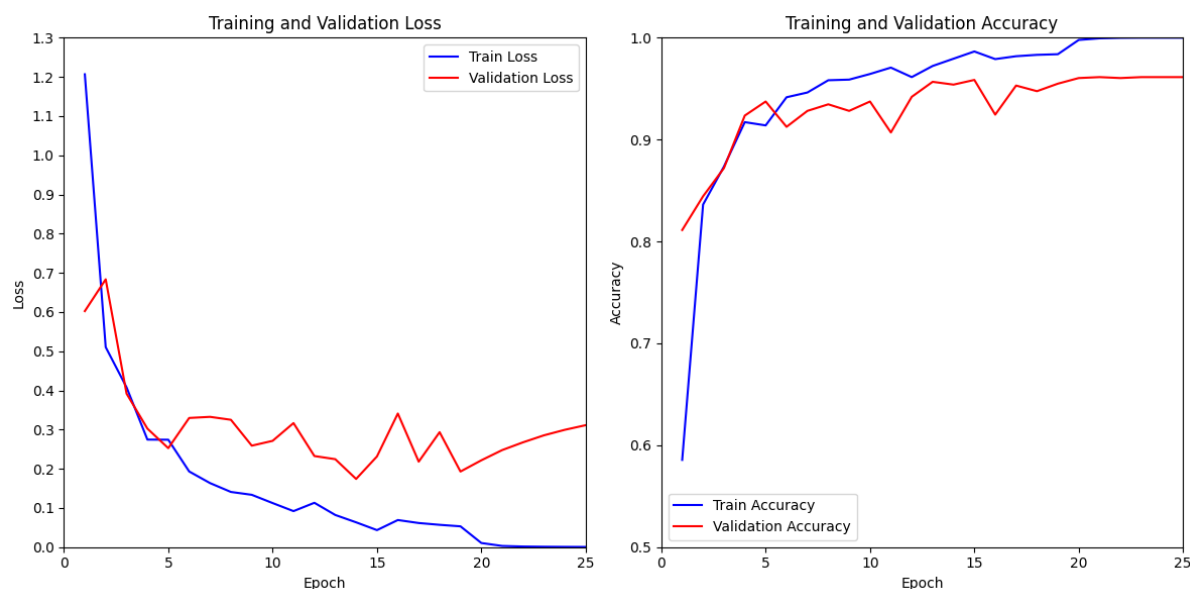


Figura 3.7: Evolución del modelo en términos de accuracy y loss.

10800 archivos de prueba de tipo .bytes (otros 10800 de tipo .asm) pero solo usaremos los .bytes. Convertimos cada archivo en una imagen. Primero, cada código hexadecimal lo convertimos a números decimales y estos los pasamos a un array de numpy. Hacemos reshape (lado,lado2) de forma que obtengamos la mayor dimension posible que sea casi cuadrado consiguiendo perder



la menor información posible. Este reshape lo pasamos a np.uint8 y finalmente lo interpolamos usando bilinal, cubic, bicubic, nearest y observamos cual es la mejor de todas zhao2023new (hacer una grafica o algo para comprobarlo). Además, para cargar los datos he usado multiprocessing con los diferentes datos de tiempo. En el caso de algunas imagenes, los archivos .bytes no contienen ningun tipo de informacion(?? ?? ?? ..) todos los byte son ??. Estos ficheros, no estan incluidos en las imagenes ni de entrenamiento ni validación porque no contienen ningun tpo de información.

La diferencia principal entre realizar el entrenamiento del modelo en la GPU o en la CPU radica en el rendimiento y la velocidad de entrenamiento.

### 3.2.4. GPU (Unidad de Procesamiento Gráfico)

#### Ventajas

- Las GPU están diseñadas específicamente para manejar operaciones matriciales y paralelas, que son comunes en el entrenamiento de modelos de redes neuronales.
- Pueden realizar cálculos en paralelo en grandes cantidades de datos, lo que acelera significativamente el entrenamiento de modelos, especialmente en tareas intensivas en cálculos, como las redes neuronales profundas.
- Ofrecen un rendimiento superior en comparación con las CPU para tareas de aprendizaje profundo.

#### Desventajas

- Pueden ser más costosas y consumir más energía que las CPU.
- Puede haber limitaciones en la cantidad de memoria de la GPU disponible, lo que podría ser un factor en modelos muy grandes.

### 3.2.5. CPU (Unidad Central de Procesamiento)

#### Ventajas

- Disponibles en la mayoría de las computadoras y servidores sin necesidad de hardware adicional.
- Adecuadas para tareas generales de propósito múltiple y no solo para aprendizaje profundo.
- Pueden ser más económicas en términos de hardware y consumo de energía.

#### Desventajas

- Las CPU no están diseñadas específicamente para tareas de aprendizaje profundo y pueden ser menos eficientes en términos de velocidad para ciertos tipos de operaciones, especialmente en modelos grandes.



### 3.3. Autoencoder

### 3.4. Resultados

[18] mete imagen de asm junto con bytes.



## Capítulo 4

# Detección de intrusiones

### 4.1. KDD Cup 1999

### 4.2. Autoencoder

Para la clasificación binaria usar autoencoder con el entrenamiento de las imágenes (buenas o malas) y según el error que den, se clasifica. Para la multclasificación, tenemos dos opciones:

- Usamos autoencoder para comprimir la información de entrada y después esa información la usamos para clasificarla usando una DNN [34]
- Usamos una cadena de autoencoders en el cual la salida de  $h$  es la entrada del autoencoder  $h+1$ . Utilizo el artículo [16] donde se desarrolla todo el modelo y explicación y además se hace referencia al artículo [6] porque se basa en él (lo de salida de  $h$  es la entrada de  $h+1$ ). Ver también:
  - Asymmetric Stacked Autoencoder
  - Constrained Nonlinear Control Allocation based on Deep Auto-Encoder Neural Networks.

El algoritmo consiste en entrenar las capas por separado en la que el input del autoencoder es la salida del autoencoder anterior. Lo que de verdad nos interesa es la capa oculta, que tiene una representación comprimida de los datos de entrada y sus pesos. Estos pesos son con los que se inicializa el entrenamiento de la stacked autoencoder acabando en softmax. He usado el url para enterlo <https://amiralavi.com/tied-autoencoders/>. Además en [5] explica bastante bien la diferencia entre capa autoencoder y un autoencoder.

### 4.3. Red Neuronal Convolutiva

Para clasificar los datos del dataset KDD 1999 usando las Convolutional Neural Network (CNN) vamos a seguir los siguientes artículos [29, 59, 47, 31]. Prácticamente todo el cuerpo del experimento se encuentra en el artículo [29], pero en el artículo [31] aparece la parte de normalización de los datos y algunos hiperparámetros de inicio.

#### 4.4. Red Neuronal Profunda

Por otro lado, el método Deep Neural Network (DNN) utiliza una arquitectura muy parecida a una CNN. Podemos ver todo el procesamiento de los datos y el modelo en el artículo [37]. Además, hay buena explicación del experimento en [57]. Por último, en el artículo [14] están los experimentos con DNN, RNN, RBM que puedo tomar también como referencia porque está muy bien explicado las capas e hiperparametros que utiliza.

#### 4.5. Red Neuronal Recurrente

En el artículo [14] están los experimentos con DNN, RNN, RBM que puedo tomar también como referencia porque está muy bien explicado las capas e hiperparametros que utiliza.

#### 4.6. Restricted Boltzmann Machine

En el artículo [14] están los experimentos con DNN, RNN, RBM que puedo tomar también como referencia porque está muy bien explicado las capas e hiperparametros que utiliza.

#### 4.7. Resultados

## Capítulo 5

# Conclusiones y Trabajo Futuro

### 5.1. Conclusiones

### 5.2. Trabajo futuro



# Bibliografía

- [1] Microsoft malware classification challenge (big 2015), 2015.
- [2] Aakash Nain. Keras Documentation. <https://keras.io/>. Consultado el 06-05-2024.
- [3] Apache Software Foundation. Apache mxnet, 2015.
- [4] Dor Bank, Noam Koenigstein, and Raja Giryes. Autoencoders. *Machine learning for data science handbook: data mining and knowledge discovery handbook*, pages 353–374, 2023.
- [5] 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.
- [6] Yoshua Bengio, Pascal Lamblin, Dan Popovici, and Hugo Larochelle. Greedy layer-wise training of deep networks. *Advances in neural information processing systems*, 19, 2006.
- [7] 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.
- [8] 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.
- [9] 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.
- [10] Keras Developers. Model training apis. Consultado el 25-04-2024.
- [11] NumPy Developers. Numpy documentation. Consultado el 25-04-2024.
- [12] Ali Diba, Mohsen Fayyaz, Vivek Sharma, Amir Hossein Karami, Mohammad Mahdi Arzani, Rahman Yousefzadeh, and Luc Van Gool. Temporal 3d convnets: New architecture and transfer learning for video classification. *arXiv preprint arXiv:1711.08200*, 2017.
- [13] 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.
- [14] Wisam Elmasry, Akhan Akbulut, and Abdul Halim Zaim. Empirical study on multiclass classification-based network intrusion detection. *Computational Intelligence*, 35(4):919–954, 2019.
- [15] Facebook AI Research. Pytorch, 2017.
- [16] 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.

- [17] Aurélien Géron. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow*. O'Reilly Media, Inc., 2022.
- [18] 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.
- [19] Google AI Team. Keras, 2015.
- [20] Google Brain Team. Tensorflow, 2015.
- [21] Sanchit Gupta, Harshit Sharma, and Sarvjeet 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.
- [22] 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.
- [23] 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.
- [24] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.
- [25] 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.
- [26] Kavya. Auto encoder — implementation. Consultado el 25-05-2024.
- [27] 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.
- [28] Max Kerkers, José Jair Santanna, and Anna Sperotto. Characterisation of the keliho. 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.
- [29] 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.
- [30] Sera Kim and Seok-Pil Lee. A bilstm-transformer and 2d cnn architecture for emotion recognition from speech. *Electronics*, 12(19):4034, 2023.
- [31] Taejoon Kim, Sang C Suh, Hyunjoon 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.



- [32] 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.
- [33] 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.
- [34] 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.
- [35] Mika Luoma-aho. Analysis of modern malware: obfuscation techniques. 2023.
- [36] Nesma Mahmoud, Youssef Essam, Radwa Elshawy, and Sherif Sakr. Dlbenc: an experimental evaluation of deep learning frameworks. In *2019 IEEE International Congress on Big Data (BigDataCongress)*, pages 149–156. IEEE, 2019.
- [37] 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.
- [38] Rosa Martínez Álvarez-Castellanos et al. Análisis de las máquinas sparse autoencoders como extractores de características. 2017.
- [39] matplotlib Developers. Matplotlib: Visualization with python. Consultado el 25-04-2024.
- [40] Matthew Carrigan. Keras Documentation. <https://keras.io/>. Consultado el 06-05-2024.
- [41] Montreal University. Theano, 2010.
- [42] Andreas C Müller and Sarah Guido. *Introduction to machine learning with Python: a guide for data scientists*. .O'Reilly Media, Inc.", 2016.
- [43] Mohammad Najafimehr, Sajjad Zarifzadeh, and Seyedakbar Mostafavi. A hybrid machine learning approach for detecting unprecedented ddos attacks. *The Journal of Supercomputing*, 78, 04 2022.
- [44] 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.
- [45] Lakshmanan Nataraj, Shanmugavadivel 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.
- [46] 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.
- [47] 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.
- [48] Gonzalo Pajares Martinsanz et al. Aprendizaje profundo. 2021.

- [49] 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.
- [50] Prajoy 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.
- [51] scikit-learn Developers. ssikit-llarn documentation. Consultado el 25-04-2024.
- [52] scikit-learn Developers. Train test split de scikit-learn. Consultado el 25-04-2024.
- [53] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.
- [54] Sajedul Talukder. Tools and techniques for malware detection and analysis. *arXiv preprint arXiv:2002.06819*, 2020.
- [55] Mingdong Tang and Quan Qian. Dynamic api call sequence visualisation for malware classification. *IET Information Security*, 13(4):367–377, 2019.
- [56] Tokyo University. Chainer, 2015.
- [57] 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.
- [58] 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.
- [59] 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.
- [60] 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.
- [61] 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.
- [62] 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.



**.1. Anexo A**

Cuadro 1: Tabla de códigos hexadecimales y decimales

Hex	Dec	Hex	Dec	Hex	Dec	Hex	Dec	Hex	Dec	Hex	Dec
00	0	2B	43	56	86	81	129	AC	172	D7	215
01	1	2C	44	57	87	82	130	AD	173	D8	216
02	2	2D	45	58	88	83	131	AE	174	D9	217
03	3	2E	46	59	89	84	132	AF	175	DA	218
04	4	2F	47	5A	90	85	133	B0	176	DB	219
05	5	30	48	5B	91	86	134	B1	177	DC	220
06	6	31	49	5C	92	87	135	B2	178	DD	221
07	7	32	50	5D	93	88	136	B3	179	DE	222
08	8	33	51	5E	94	89	137	B4	180	DF	223
09	9	34	52	5F	95	8A	138	B5	181	E0	224
0A	10	35	53	60	96	8B	139	B6	182	E1	225
0B	11	36	54	61	97	8C	140	B7	183	E2	226
0C	12	37	55	62	98	8D	141	B8	184	E3	227
0D	13	38	56	63	99	8E	142	B9	185	E4	228
0E	14	39	57	64	100	8F	143	BA	186	E5	229
0F	15	3A	58	65	101	90	144	BB	187	E6	230
10	16	3B	59	66	102	91	145	BC	188	E7	231
11	17	3C	60	67	103	92	146	BD	189	E8	232
12	18	3D	61	68	104	93	147	BE	190	E9	233
13	19	3E	62	69	105	94	148	BF	191	EA	234
14	20	3F	63	6A	106	95	149	C0	192	EB	235
15	21	40	64	6B	107	96	150	C1	193	EC	236
16	22	41	65	6C	108	97	151	C2	194	ED	237
17	23	42	66	6D	109	98	152	C3	195	EE	238
18	24	43	67	6E	110	99	153	C4	196	EF	239
19	25	44	68	6F	111	9A	154	C5	197	F0	240
1A	26	45	69	70	112	9B	155	C6	198	F1	241
1B	27	46	70	71	113	9C	156	C7	199	F2	242
1C	28	47	71	72	114	9D	157	C8	200	F3	243
1D	29	48	72	73	115	9E	158	C9	201	F4	244
1E	30	49	73	74	116	9F	159	CA	202	F5	245
1F	31	4A	74	75	117	A0	160	CB	203	F6	246
20	32	4B	75	76	118	A1	161	CC	204	F7	247
21	33	4C	76	77	119	A2	162	CD	205	F8	248
22	34	4D	77	78	120	A3	163	CE	206	F9	249
23	35	4E	78	79	121	A4	164	CF	207	FA	250
24	36	4F	79	7A	122	A5	165	D0	208	FB	251
25	37	50	80	7B	123	A6	166	D1	209	FC	252
26	38	51	81	7C	124	A7	167	D2	210	FD	253
27	39	52	82	7D	125	A8	168	D3	211	FE	254
28	40	53	83	7E	126	A9	169	D4	212	FF	255
29	41	54	84	7F	127	AA	170	D5	213	??	-/ -1
2A	42	55	85	80	128	AB	171	D6	214		