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

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. Introducción

En la última década, la inteligencia artificial (IA) se ha convertido en un tema popular tanto dentro como fuera de la comunidad científica. Una abundancia de artículos en revistas tecnológicas y no tecnológicas han cubierto los temas de aprendizaje automático (ML, por sus siglas en inglés), aprendizaje profundo (DL, por sus siglas en inglés) e IA. Sin embargo, todavía persiste confusión en torno a IA, ML y DL. Los términos están estrechamente relacionados, pero no son intercambiables.

En 1956, un grupo de científicos informáticos propuso que las computadoras podrían ser programadas para pensar y razonar, "que cada aspecto del aprendizaje o cualquier otra característica de la inteligencia podría, en principio, ser descrito tan precisamente que una máquina podría simularlo" [48]. Describieron este principio como "inteligencia artificial". En pocas palabras, la

IA es un campo enfocado en automatizar tareas intelectuales que normalmente realizan los humanos, y el Machine Learning es un método específico para lograr este objetivo. Es decir, está dentro del ámbito de la IA (Figura 2.1) [11].

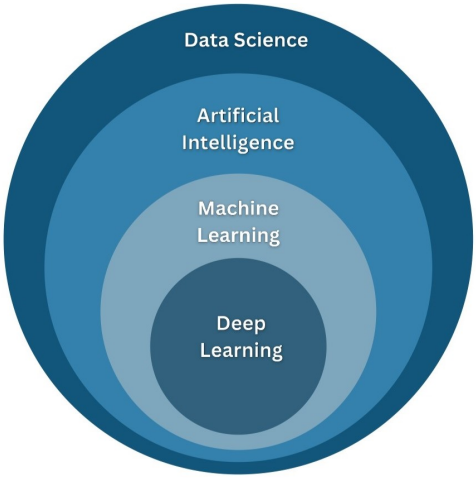


Figura 2.1: Relación entre Ciencia de Datos, Inteligencia Artificial, Machine Learning y Deep Learning.

2.3. Aprendizaje Automático

Por otro lado, el Aprendizaje Automático (ML, por sus siglas en inglés) es la ciencia o el arte de programar ordenadores para que puedan aprender a partir de datos. Arthur Samuel lo definió en 1959 como "el campo de estudio que otorga a las computadoras la capacidad de aprender sin ser explícitamente programadas". 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" [22]. 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, donde se codifican reglas explícitas, el aprendizaje automático permite que los sistemas descubran patrones y relaciones directamente a partir de los datos, adaptándose y mejorando con el tiempo.

Un ejemplo de aprendizaje automático es un filtro de spam que, dado ejemplos de correos electrónicos de spam y ejemplos de correos electrónicos normales (no spam, también llamados "ham"), puede aprender a marcar el spam [22]. 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 [22].

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 atributos de 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 [55].

En cuanto a la clasificación de los sistemas de aprendizaje automático, se distinguen cuatro tipos principales:

El **aprendizaje supervisado**, es un tipo de entrenamiento en el que los datos tienen asociados las salidas deseadas, también llamadas etiquetas. Un ejemplo de aprendizaje supervisado 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.

Al contrario que el aprendizaje supervisado, en el **aprendizaje no supervisado**, los datos de entrenamiento no están etiquetados, luego el modelo tiene que aprender sin "profesor". El objetivo de este tipo de algoritmos es otro como el de agrupamiento, detección de anomalías o reducción de dimensionalidad.

En el caso de encontrarnos ante un problema en el que tengamos datos tanto etiquetados como sin etiquetar, nos encontramos antes un tipo de **aprendizaje semisupervisado**, que se encuentra entre el supervisado y el no supervisado. 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.

Por último, está el **aprendizaje por refuenzo**, un tipo de aprendizaje un poco diferente a los otros tres. El sistema de aprendizaje, llamado agente, observa el entorno, selecciona y realiza acciones para obtener recompensas a cambio (o penalizaciones en forma de recompensas negativas). Luego debe aprender por sí mismo cuál es la mejor estrategia, llamada política, para obtener la mayor recompensa con el tiempo. Una política define qué acción debe elegir el agente cuando se encuentra en una situación determinada. Por ejemplo, muchos robots implementan algoritmos de aprendizaje por refuerzo para aprender a andar.

¿PONER O NO? El aprendizaje automático es excelente para:

- Problemas para los cuales las soluciones existentes requieren muchos ajustes o largas listas de reglas: un algoritmo de aprendizaje automático a menudo puede simplificar el código y funcionar mejor que el enfoque tradicional.
- Problemas complejos para los que el enfoque tradicional no ofrece una buena solución: las mejores técnicas de Machine Learning quizás puedan encontrar una solución.
- Entornos fluctuantes: un sistema de Machine Learning puede adaptarse a nuevos datos.
- Obtener información sobre problemas complejos y grandes cantidades de datos.

2.4. Aprendizaje profundo

Dentro del Machine Learning, se encuentra el Deep Learning, cuya base son las redes neuronales artificiales (ANN). Una red neuronal artificial es 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 [17]. 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.

2.4.1. Perceptrón

Antes de profundizar en los modelos de redes neuronales más complejos y profundos, veamos el funcionamiento del modelo 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 clases.

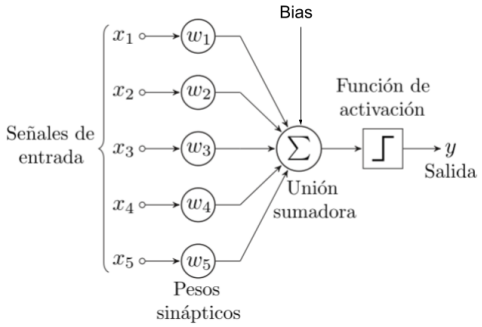


Figura 2.2: Modelo del perceptrón simple

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

$$f(x) = \sum_{i=1}^{n} w_i \cdot x_i + b \tag{2.1}$$

Los coeficientes w_i, $i = 1, 2, \ldots, n$ llamados pesos, 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: $f(x) = \sum_{i=1}^{n+1} w_i \cdot z_i = w^t \cdot z dondew = (w_1, \ldots, w_n, b)^t yz^t = (x_1, \ldots, x_n, 1)^t$

La segunda capa consiste en transformar la salida de la primera etapa mediante una función de activación. En el caso de un problema de clasificación binaria (0 o 1), si esta salida sobrepasa un cierto umbral predefinido al principio, su salida será 1 y en caso contrario, será 0. Es decir, siendo c una constante real:

$$\text{y} = \begin{cases} 1 \text{ si } \sum_{i=1}^{n} w_i \cdot x_i + b > c \\ 0 \text{ si } \sum_{i=1}^{n} w_i \cdot x_i + b <= c. \end{cases} \tag{2.2}$$

2.5. GENERALIZACIÓN

2.6. COMO SE APRENDE

2.7. OPTIMIZACIÓN

2.8. FUNCION DE ACTIVACION. AJUSTAR LAS FOTOS CUANDO ACABE

Las funciones de activación en redes neuronales son cruciales ya que introducen no linealidad en el modelo, permitiendo que la red neuronal pueda aprender, representar patrones complejos y modelar relaciones no lineales entre las características de entrada y la salida. En esta sección, describimos varias funciones de activación populares, sus propiedades y aplicaciones [55]. Todas las gráficas de las funciones se pueden ver en la Figura 2.3.

Función 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, donde la suma de todas las probabilidades es 1. La función Softmax está definida como:

$$\text{Softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{2.3}$$

donde z_i es el valor de la i-ésima neurona de salida, y K es el número total de clases. 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.

Función Sigmoide

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

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.4}$$

Proyecta salidas de números reales de entrada al intervalo $[0, 1]$. Sin embargo, tiene problemas como la saturación del gradiente (cuando los valores se aproximan a 0 o 1, el gradiente tiende

a 0, lo que afecta en la actualización de los pesos) o los pesos positivos de forma continua (la esperanza de la función de salida no es 0), lo que puede llevar a una convergencia lenta.

Función Sigmoide Dura

La función sigmoide dura (*hard-sigmoid*) es una aproximación simplificada de la sigmoide que reduce los problemas de saturación del gradiente:

$$f(x) = \text{máx}(0, \text{mín}(1, \frac{x+1}{2})) \tag{2.5}$$

Esta función es computacionalmente más eficiente y evita los problemas de saturación en los extremos.

Función Tangente Hiperbólica (tanh)

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

$$\tanh(x) = \frac{2}{1 + e^{-2x}} - 1 \tag{2.6}$$

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

Función Unidad Lineal Rectificada (ReLU)

La función ReLU es popular debido a su simplicidad y efectividad para solventar el problema de la saturación del gradiente. Para $x = 0$ no es derivable, por lo que se suele asignar un valor arbitrario como 0, 0.5 o 1:

$$f(x) = \text{máx}(0, x) \tag{2.7}$$

Sin embargo, puede llevar a neuronas muertas si reciben un gradiente negativo alto. Para ello, se introduce la siguiente función.

Función Leaky ReLU (LReLU)

Leaky ReLU es una variante de ReLU que introduce una pendiente pequeña para entradas negativas:

$$f(x) = \text{máx}(0{,}001x, x) \tag{2.8}$$

Esto ayuda a evitar el problema de neuronas muertas al permitir un pequeño gradiente para valores negativos.

Función Paramétrica ReLU (PReLU)

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:

$$f(x;\alpha) = \begin{cases} \alpha x & \text{si } x < 0 \\ x & \text{si } x \geq 0 \end{cases} \tag{2.9}$$

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 ReLU6

ReLU6 es una variante de ReLU que limita la activación a un máximo de 6:

$$\text{ReLU6}(x) = \text{mín}(\text{máx}(x, 0), 6) \tag{2.10}$$

Función Unidad Lineal Exponencial (ELU)

La función ELU es otra variante que proporciona una curva exponencial para entradas negativas, ayudando a suavizar el gradiente:

$$f(x;\alpha) = \begin{cases} \alpha(e^x - 1) & \text{si } x < 0 \\ x & \text{si } x \geq 0 \end{cases} \tag{2.11}$$

Función Unidad Lineal Exponencial Escalada (SELU)

La función SELU es una versión modificada de ELU, que introduce un parámetro adicional λ para escalar la salida:

$$f(x;\alpha) = \begin{cases} \lambda x & \text{si } x \geq 0 \\ \lambda\alpha(e^x - 1) & \text{si } x < 0 \end{cases} \tag{2.12}$$

SELU está diseñada para mantener la activación y la varianza de las entradas constantes en redes profundas, facilitando la normalización interna.

Funciones Swish y SiLU

Swish y SiLU combinan la sigmoide con la entrada misma para proporcionar una activación suave y no lineal:

$$\text{SiLU} : f(x) = x \cdot \sigma(\beta x) = \frac{x}{1 + e^{-\beta x}} \tag{2.13}$$

donde $\sigma(x)$ es la función sigmoide. La Figura 2.3 muestra Swish con $\beta = 3$ y SiLU con $\beta = 1$.

Función Mish

Mish combina suavidad y capacidad de modelado no lineal, definida como:

$$f(x) = x \cdot \tanh(\ln(e^x + 1)) \tag{2.14}$$

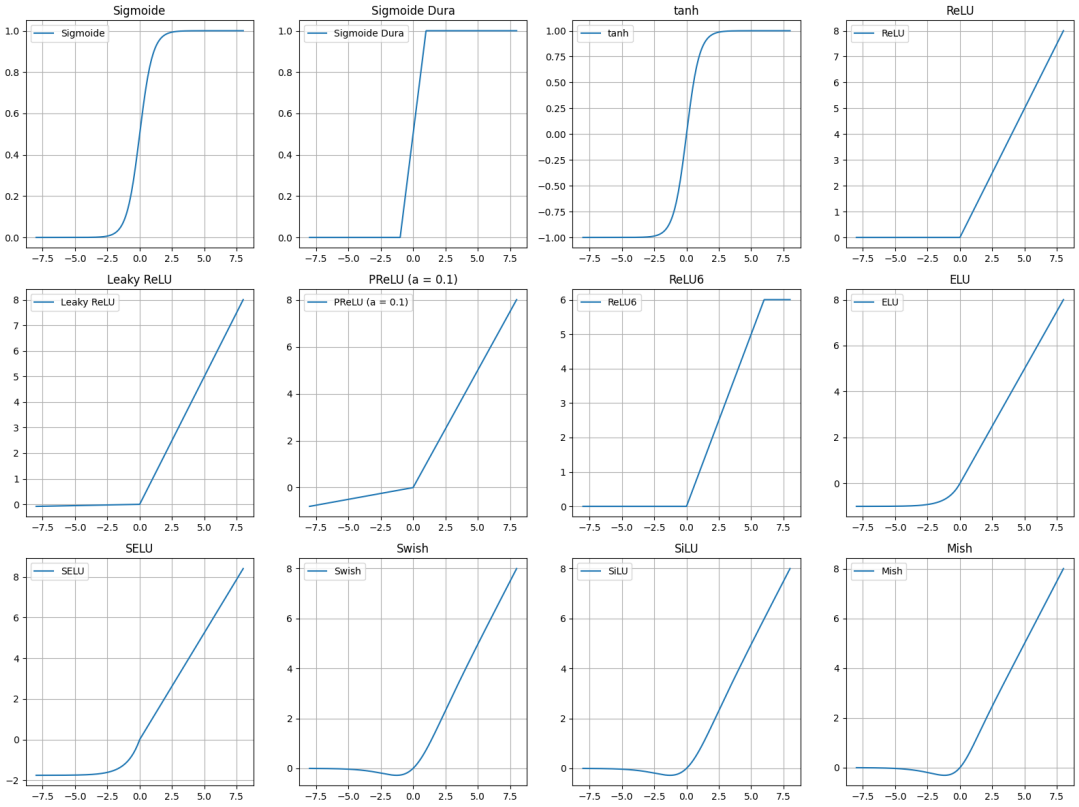


Figura 2.3: Gráficas de las funciones de activación.

Las funciones de activación juegan un papel esencial en la formación de redes neuronales al permitir que las redes aprendan y modelen relaciones complejas. Desde funciones clásicas como la sigmoide y *tanh*, hasta variantes más recientes como ReLU, LReLU, ELU, y SELU, cada función tiene sus propios beneficios y limitaciones. La elección de la función de activación adecuada puede influir significativamente en el rendimiento del modelo. Además, permiten resolver el desvanecimiento del gradiente durante el proceso de aprendizaje, evitando que el gradiente se acerque a cero, ya que es necesario que sean derivables [55].

2.9. Función de perdida

En el aprendizaje supervisado, las funciones de pérdida son herramientas muy importantes para evaluar qué tan bien un modelo predice los resultados esperados. Su propósito 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 [55]. Ambas buscan minimizar el error, pero lo hacen con diferentes enfoques dependiendo de la naturaleza de la variable de salida.

2.9.1. 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. Existen dos variantes principales:

Entropía Cruzada Categórica

La entropía cruzada categórica se utiliza cuando se predicen múltiples categorías (multiclase). Se calcula como:

$$H(p, q) = -\sum_{x} p(x) \log q(x) \tag{2.15}$$

Aquí, $p(x)$ es la distribución verdadera (generalmente representada como *one-hot* para una categoría) y $q(x)$ es la distribución predicha. Esta función penaliza las predicciones que se desvían de las probabilidades verdaderas, con valores más altos indicando mayores discrepancias.

Entropía Cruzada Binaria

La entropía cruzada binaria se aplica a problemas de clasificación binaria, donde solo hay dos posibles etiquetas. Se define como:

$$H(p, q) = -\left[p \log q + (1 - p) \log(1 - q)\right] \tag{2.16}$$

Aquí, 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.

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:

$$D = \frac{2|P \cap G|}{|P| + |G|} \tag{2.17}$$

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 [55].

Divergencia de Kullback-Leibler

La divergencia de Kullback-Leibler (KL-divergence) mide la diferencia entre dos distribuciones de probabilidad: la distribución verdadera P y la distribución predicha Q. Se expresa como:

$$D_{KL}(P\|Q) = \sum_x P(x) \log \frac{P(x)}{Q(x)} \tag{2.18}$$

Este valor indica cuánta información se pierde al usar Q para aproximar P. En el contexto de modelos de clasificación, una menor divergencia KL indica que las distribuciones de probabilidades predichas se acercan más a las verdaderas, haciendo al modelo más preciso [55].

2.9.2. 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:

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 \tag{2.19}$$

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 [55].

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:

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i| \tag{2.20}$$

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 [55].

Pérdida de Huber

La función de pérdida de Huber combina las características del ECM y el EAM. Se comporta de manera cuadrática para errores pequeños y de manera lineal para errores grandes. Está definida como:

$$L_\delta(y, \hat{y}) = \begin{cases} \frac{1}{2}(y - \hat{y})^2 & \text{si } |y - \hat{y}| \leq \delta \\ \delta|y - \hat{y}| - \frac{1}{2}\delta^2 & \text{si } |y - \hat{y}| > \delta \end{cases} \tag{2.21}$$

El parámetro δ controla el punto en el que la transición ocurre entre la penalización cuadrática y la lineal. Esto permite combinar la sensibilidad a los errores pequeños del ECM con la robustez frente a los errores grandes del EAM [55].

Pérdida Log-Cosh

La función de pérdida Log-cosh es otra opción para la regresión que suaviza los errores más que la pérdida cuadrática. Se define por:

$$L = \sum_{i=1}^{n}\log(\cosh(\hat{y}_i - y_i)) \tag{2.22}$$

Aquí, cosh es la función hiperbólica del coseno. La ventaja de esta función es que, para errores pequeños, se comporta como el ECM, mientras que para errores grandes se asemeja al EAM, lo que permite una suavización de los errores sin la severidad de la penalización cuadrática [55].

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. Se define como:

$$L_\tau(y, \hat{y}) = \sum_{i=1}^{n} (\tau \, \text{máx}(y_i - \hat{y}_i, 0) + (1 - \tau) \, \text{máx}(\hat{y}_i - y_i, 0)) \tag{2.23}$$

donde τ es el cuantil elegido.

2.10. 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 2.4, 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.

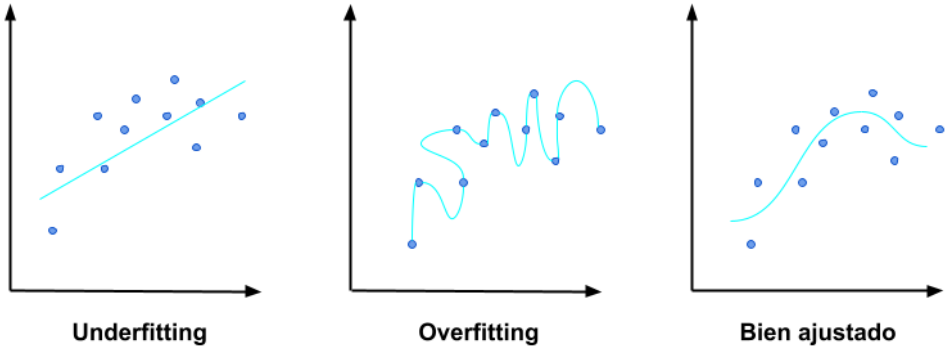


Figura 2.4: Ejemplo de sobreajuste, subajuste y buen ajuste.

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 $L1$ y $L2$, las capas dropout o la parada temprana.

2.10.1. 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 [55].

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 [22].

$$loss_{regul}(\theta) = loss(\theta) + \beta \cdot f(\theta) \tag{2.24}$$

Aquí, $loss_{regul}(\theta)$ es la función de pérdida final despues de aplicarle una regularización. $loss(\theta)$ es la función de pérdida original que mide el ajuste del modelo a los datos, y θ 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 $f(\theta)$ 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 al cuadrado. La función de pérdida regularizada $L2$ se define como:

$$\text{Loss} = \text{Original Loss} + \lambda \sum_i w_i^2 \tag{2.25}$$

donde λ es el parámetro de regularización que controla la importancia de la penalización y w_i los pesos [22]. 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:

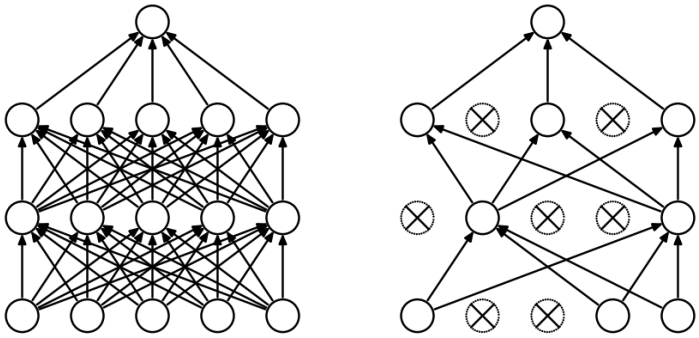
$$\text{Loss} = \text{Original Loss} + \lambda \sum_i |w_i| \tag{2.26}$$

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 $L2$ [22].

2.10.2. Dropout

Otro método para evitar el sobreajuste de nuestro modelo es el Dropout, que fue propuesto por Hinton et al. [27] como una forma de regularización para capas de redes neuronales completa-

mente 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 [22], este valor de p suele variar entre 0.1 y 0.5, siendo este último uno de los más típicos [60], aunque la red neuronal que se esté utilizando influye en el valor óptimo [22]. La elección de este parámetro es muy importnate, ya que si tomamos un valor muy alto, puede llevarnos a underfitting.



(a) Standard Neural Net (b) After applying dropout.

Figura 2.5: Esquema de la técnica dropout. Fuente [61]

Al eliminar una neurona, esta se elimina de la red con todas sus conexiones entrantes y salientes (ver Figura 2.5). 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 "reducida" 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 [60]. 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:

```
from tensorflow.keras.layers import Dropout

model = Sequential([
    Dense(128, activation='relu'),
    Dropout(0.5),
    Dense(10, activation='softmax') ])
```

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

2.10.3. 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.

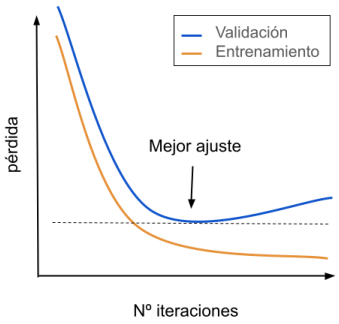


Figura 2.6: Curvas de error de entrenamiento y validación con Early Stopping.

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 [22]. La Figura 2.6 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 [64].

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:

```
from tensorflow.keras.callbacks import EarlyStopping

early_stopping_cb = keras.callbacks.EarlyStopping(patience=10,
                                                  restore_best_weights=True)
history = model.fit(X_train, y_train, epochs=100,
                    validation_data=(X_valid, y_valid),
                    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 [22].

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 [22, 55].

2.11. Métricas de evaluación de un modelo de clasificación.

Para evaluar el rendimiento de un modelo de clasificación, se utilizan una serie de métricas que nos permiten entender el rendimiento y efectividad de un modelo. A continuación, se describen las principales métricas utilizadas, así como las representaciones gráficas asociadas.

2.11.1. 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.

2.11.2. Metricas

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 sensibilidad. A continuación vamos a definir y explicar las principales métricas a tener en cuenta según [19] 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:

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

2.11.3. Curva ROC y AUC

La curva ROC (Receiver Operating Characteristic) es otra herramienta utilizada en los modelos de clasificación [22]. 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. La linea discontinua de la figura 2.7 indica la curva ROC de un clasificador puramente aleatorio [22].

Otro término que está relacionado es el AUC (*area under the curve*). El AUC cuantifica la capacidad que tiene el modelo para distinguir entre las diferentes clases. Cuanto mayor valor, mejor rendimiento. En la Figura 2.7 se representa con rallas de intensidad baja.

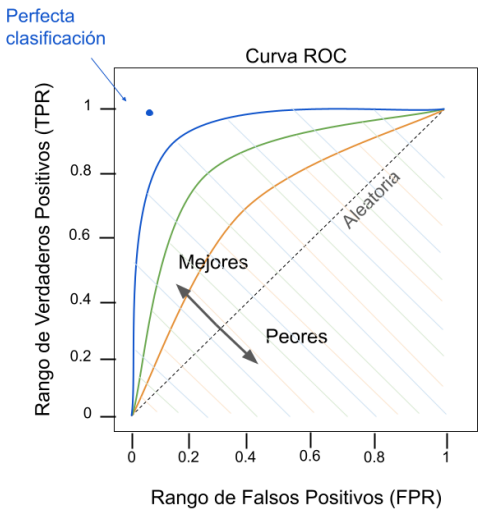


Figura 2.7: Interpretación de la curva ROC con el área bajo la curva (AUC).

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 [22].

2.11.4. Curva de Precisión-Recall

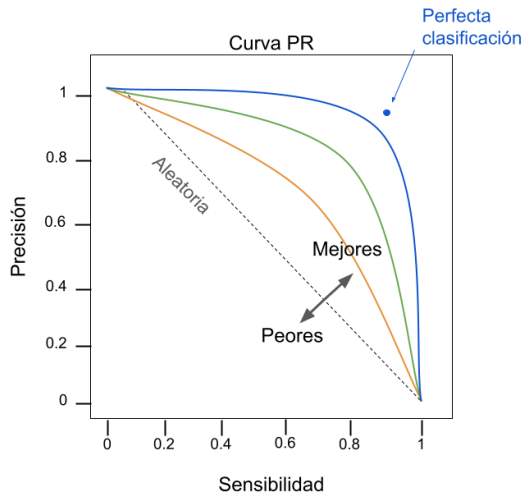


Figura 2.8: Curva de Precisión-Recall.

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 2.9 ilustra cómo se comportan la precisión y la sensibilidad en función del umbral [22].

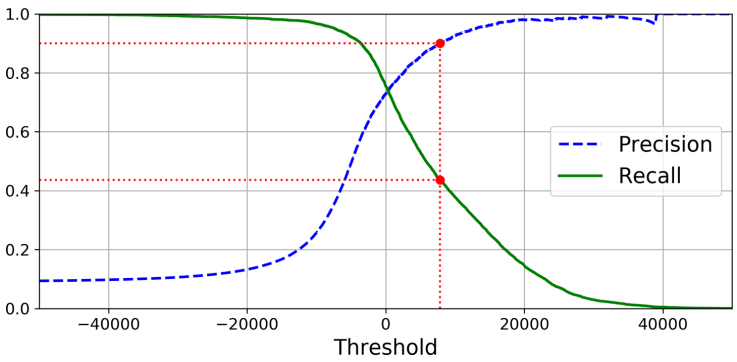


Figura 2.9: Precisión y Sensibilidad vs Umbral de decisión. Fuente [22]

2.12. 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

2.12.1. 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 [40]. La arquitectura básica de un autoencoder consta de tres partes: el encoder, el cuello de bottela y el decoder (Figura 2.10).

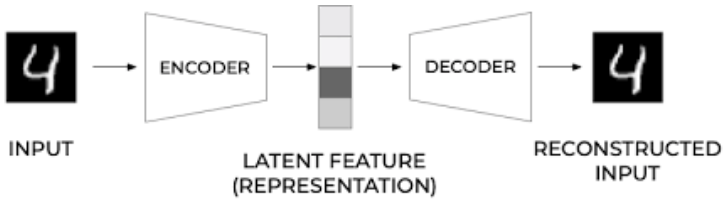


Figura 2.10: Arquitectura de un autoencoder. Fuente:[32].

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 [5], es aprender las funciones

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

y

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

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 Δ 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:

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

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 [44].

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[1] o LSTM en el caso de autocodificadores recurrentes[2][22].

2.12.2. Deep Belief Networks

Red Neuronal Profunda

2.12.3. 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 [57]. La arquitectura de una CNN (Figura 2.11) consta de tres tipos de capas: capas de convolución, capas de pooling y la capa de clasificación.

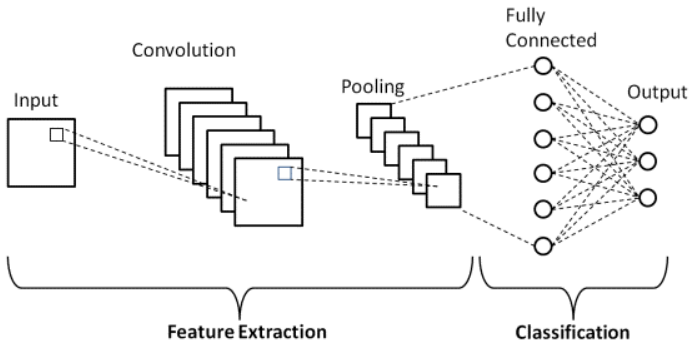


Figura 2.11: Arquitectura de una CNN con capas de convolución, pooling y clasificación. Fuente:[56].

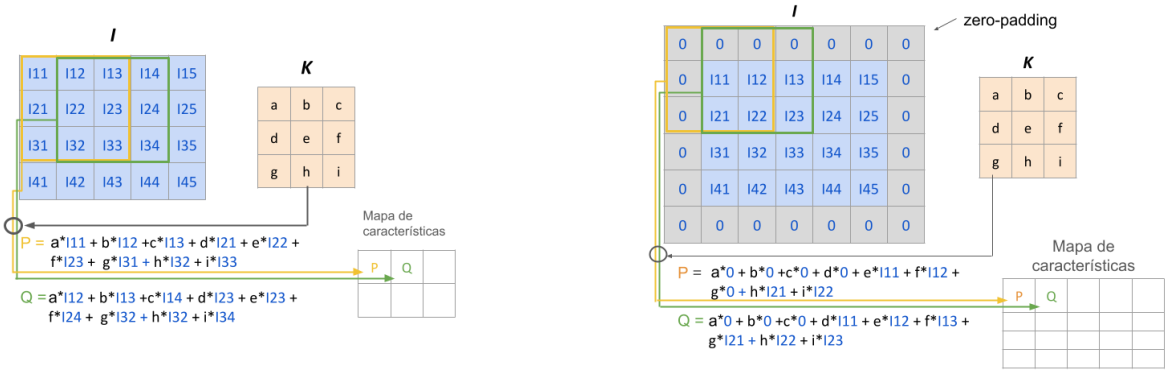
Capas convolucionales

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 [55].

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[3] [22]. Esta convolución se realiza con un solapamiento total del filtro, lo que resulta en una imagen de menor dimensión (Figura 2.12a). 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 2.12b). En la figura 2.12 se

[1] Autocodificador para imágenes de gran tamaño
[2] Autocodificador específico para series temporales o secuencias.
[3] Región de entrada que contribuye a la salida generada por el filtro

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.



(a) Convolución reduciendo tamaño. (b) Convolución con zero-padding.

Figura 2.12: 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.

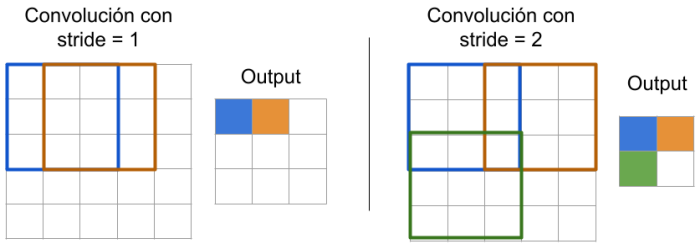


Figura 2.13: Campos receptivos en una convolución. Adaptación de imagen de [70]

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 [70].

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 [22].

Toda la información anterior se resume en la siguiente ecuación [55]:

$$z_{i,j,k} = b_k + \sum_{u=0}^{f_h-1} \sum_{v=0}^{f_w-1} \sum_{k'=0}^{f'_n-1} x_{i',j',k'} \cdot w_{u,v,k',k} \quad \text{con} \begin{cases} i' = i \cdot s_h + u \\ j' = j \cdot s_w + v \end{cases} \tag{2.27}$$

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 [57]. 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[4]. Las tres funciones más comunes son el promedio, la suma y el máximo. La Figura 2.14 muestra una capa de max pooling, que es el tipo más común [22].

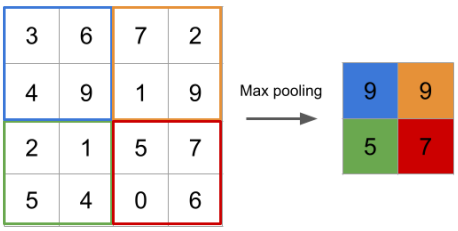


Figura 2.14: Capa de max pooling con un kernel de 2×2, stride 2 y sin padding.

Además de reducir el número de operaciones, el número de parámetros y ayudar con el overfitting, 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 2.15. Esto significa que pequeñas variaciones en la posición de las características dentro de la imagen no afectan significativamente la salida.

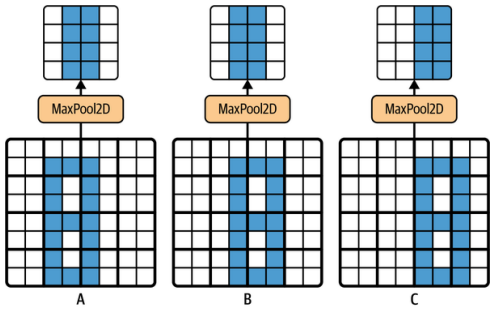


Figura 2.15: Invarianza a translaciones pequeñas mediante una capa de max pooling. Fuente [22]

[4]Las funciones de agregación devuelven un valor único de un conjunto de registros.

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. 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 [28].

2.12.4. Red Neuronal Recurrente

Restricted Boltzmann Machine

2.13. 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.

2.13.1. 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 [25] 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 [24] 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 [49]. 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 [20], 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 [42]. Esto, junto con su capacidad de auto-diferenciación en modo inverso[5], 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 [42]. 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 cuento 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 [3] 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 [46], 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."

2.13.2. 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 [58] 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

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

scikit-learn, si que se va a utilizar una de sus funciones más populares, `train_test_split()` [59]. 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 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.

```
# Ejemplo de codigo en Python
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(data, labels,
                                                    test_size=0.25, random_state=42)
```

Las variables X_train, X_ test y compañía son numpy arrays. **NumPy** [14] 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 [9]. 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 [12], 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** [45]. 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 [2].

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 [57] 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 referenciacios 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) [2], 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 [57].

Se puede descargar desde su página web [2]. 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 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 bytes[1] 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.

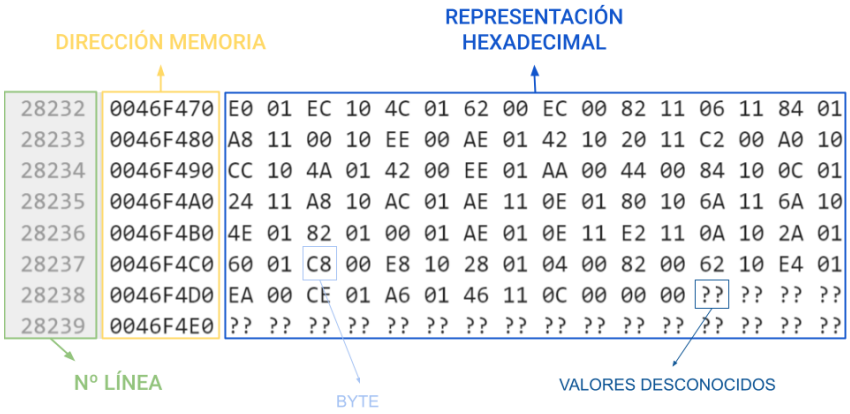


Figura 3.1: Explicación del contenido de "0ACDbR5M3ZhBJajygTuf.bytes".

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 [10].

[1]Realmente no es un archivo bytes, sino un fichero de texto con caractéres.

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: Rammit, Lollipop, Kelihos_ ver3, Vundo, Simda, Tracur, Kelihos_ ver1, Obfuscator y Gatak. Según [29], podemos deinirlas 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.

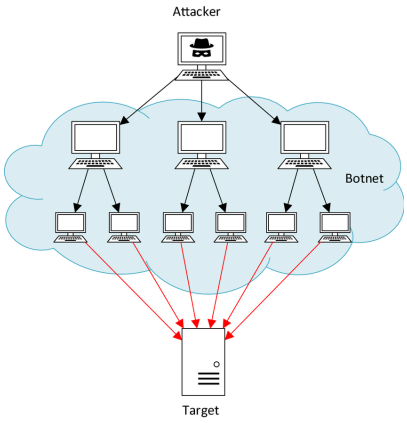


Figura 3.2: Estructura de un botnet. Imagen sacada de [50].

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 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[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 aciones 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 [34]: 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 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.

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

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

[4] 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[5] (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).

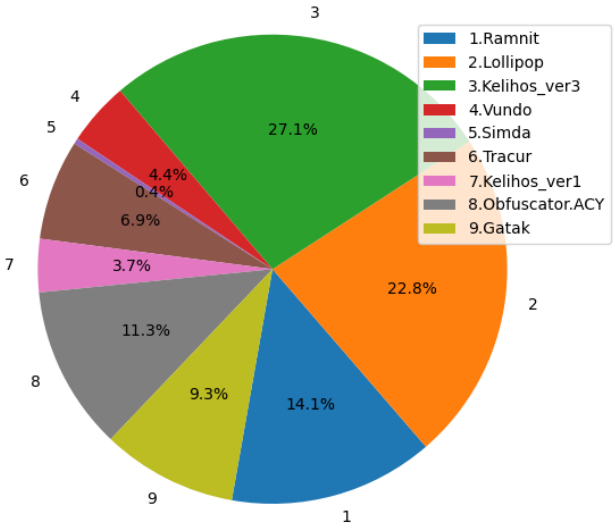


Figura 3.3: Distribución del BIG 2015 training dataset.

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 [33] 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

[5]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 Convolucional

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 2.12.3, 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 [52], pero si lo representamos de forma matricial, estos pequeños cambios pueden ser detectados fácilmente [31].

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 [53], 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 [22], se plantea eliminar estos caracteres y tratar el resto de bytes. Otra solución la proponen Narayanan et al. [51], 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.

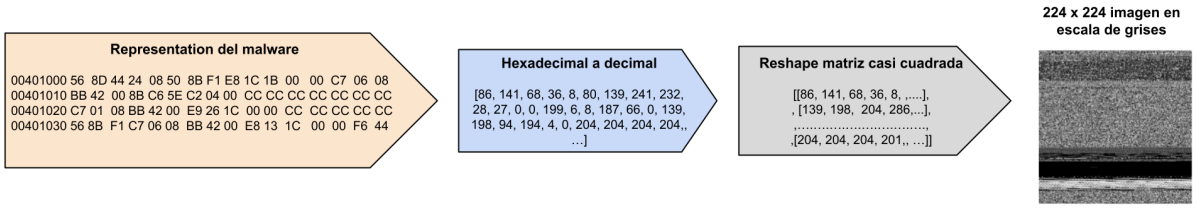


Figura 3.4: Proceso de visualización del malware. Adaptación de [53]

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 [38]. Para ello, siguiendo el ejemplo de [31], decidimos escoger como tamaño 224×224. Para obtener estas dimensiones, Simonyan et al. [1] deciden recortar aleatoriamente un cuadrado de la imagen de tamaño 224×224, 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.

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

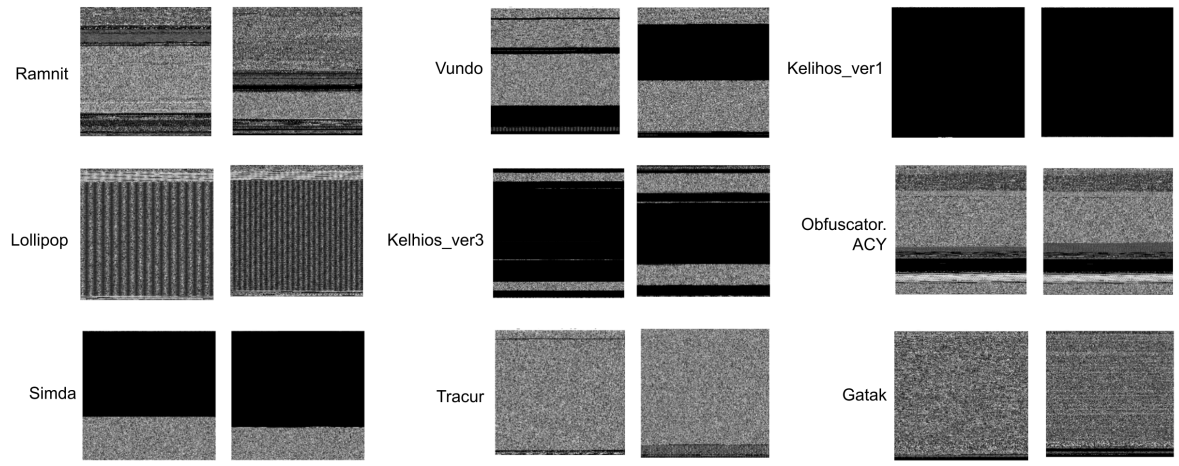


Figura 3.5: Visualización familias malware.

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

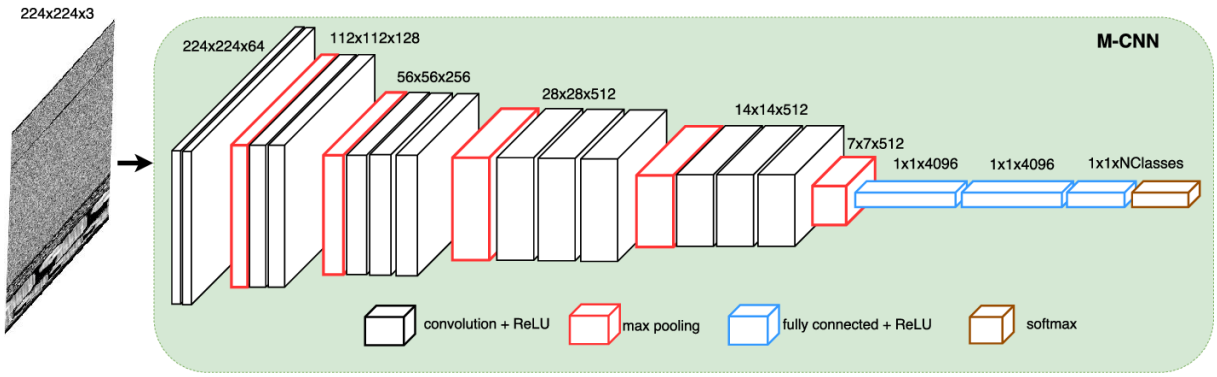


Figura 3.6: Arquitectura de la CNN. Fuente [31].

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[22]. 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 2×2, 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[22].

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. 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 [31]. La función de pérdida utilizada es categorical_crossentropy ya que nuestro problema involucra múltiples clases de etiquetas [13].

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.

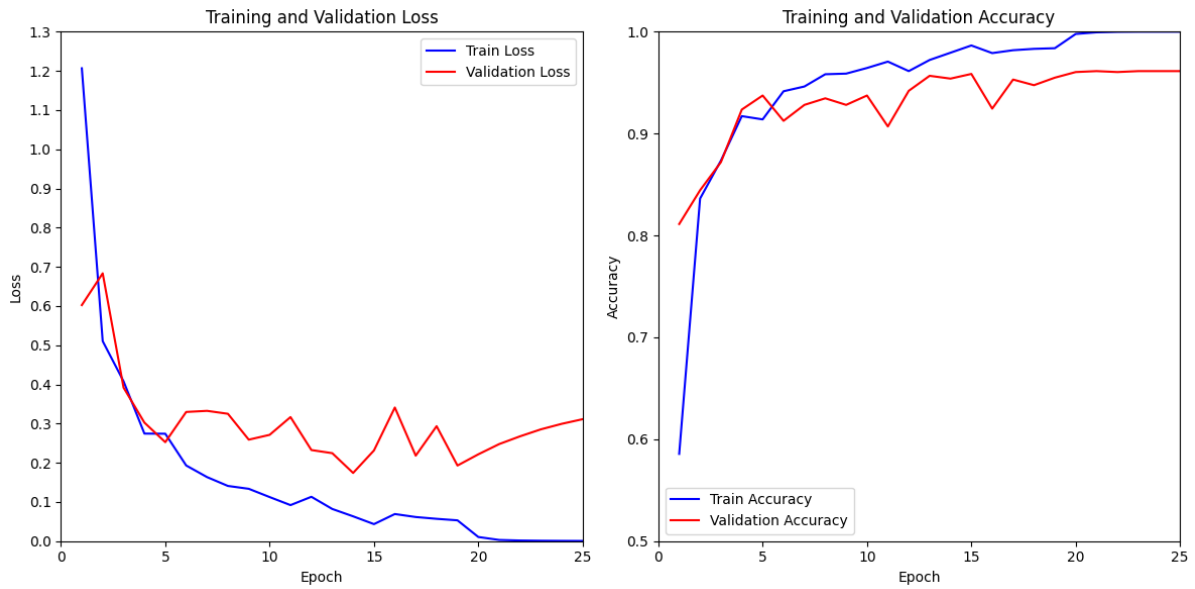


Figura 3.7: Evolución del modelo en términos de accuracy y loss.

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 numeros 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 caragr 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

Convolutional autoencoder modelo [68]

3.4. Resultados

[23] mete imagen de asm junto con bytes.

Capítulo 4

Detección de intrusiones

4.1. KDD Cup 1999

4.2. Autoencoder

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

- Usamos autoencoder para comprimir la información de entrada y despues esa informacion la usamos para clasificarla usando una DNN [40]
- Usamos una cadena de autoencoders en el cual la salida de h es la entrada del autoencoder h+1. Utilizo el articulo [21] donde se desarrolla todo el modelo y explicacion y ademas se hace referencia al artículo [7] porque se basa en él (lo de salida de h es la entrada de h+1). Ver tambien:
  - 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 [6] explica bastante bien la diferencia entre capa autoencoder y un autoencoder.

4.3. Red Neuronal Convolucional

Para clasificar los datos del dataset KDD 1999 usando las Convolutional Neural Network (CNN) vamos a seguir los siguientes articulos [35, 69, 54, 37]. Prácticamente todo el cuerpo del experimento se encuentra en el artículo [35], pero en el artículo [37] aparece la parte de normalización de los datos y algunos hiperparametros 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 [43]. Además, hay buena explicacion del experimento en [66]. Por último, en el articulo [18] 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 articulo [18] 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 articulo [18] 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] Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[2] Microsoft malware classification challenge (big 2015), 2015.

[3] Aakash Nain. Keras Documentation. `https://keras.io/`. Consultado el 06-05-2024.

[4] Apache Software Foundation. Apache mxnet, 2015.

[5] Dor Bank, Noam Koenigstein, and Raja Giryes. Autoencoders. *Machine learning for data science handbook: data mining and knowledge discovery handbook*, pages 353–374, 2023.

[6] 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.

[7] Yoshua Bengio, Pascal Lamblin, Dan Popovici, and Hugo Larochelle. Greedy layer-wise training of deep networks. *Advances in neural information processing systems*, 19, 2006.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] Keras Developers. Model training apis. Consultado el 25-04-2024.

[13] Keras developers. Probabilistic losses. Consultado el 05-06-2024.

[14] NumPy Developers. Numpy documentation. Consultado el 25-04-2024.

[15] 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.

[16] 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.

[17] Oscar R Dolling and Eduardo A Varas. Artificial neural networks for streamflow prediction. *Journal of hydraulic research*, 40(5):547–554, 2002.

[18] Wisam Elmasry, Akhan Akbulut, and Abdul Halim Zaim. Empirical study on multiclass classification-based network intrusion detection. *Computational Intelligence*, 35(4):919–954, 2019.

[19] Bradley J Erickson and Felipe Kitamura. Magician's corner: 9. performance metrics for machine learning models, 2021.

[20] Facebook AI Research. Pytorch, 2017.

[21] 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.

[22] Aurélien Géron. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow*. .º'Reilly Media, Inc.", 2022.

[23] 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.

[24] Google AI Team. Keras, 2015.

[25] Google Brain Team. Tensorflow, 2015.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[31] 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.

[32] Kavya. Auto encoder — implementation. Consultado el 25-05-2024.

[33] 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.

[34] 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.

[35] 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.

[36] Sera Kim and Seok-Pil Lee. A bilstm–transformer and 2d cnn architecture for emotion recognition from speech. *Electronics*, 12(19):4034, 2023.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] Mika Luoma-aho. Analysis of modern malware: obfuscation techniques. 2023.

[42] 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.

[43] 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.

[44] Rosa Martínez Álvarez-Castellanos et al. Análisis de las máquinas sparse autoencoders como extractores de características. 2017.

[45] matplotlib Developers. Matplotlib: Visualization with python. Consultado el 25-04-2024.

[46] Matthew Carrigan. Keras Documentation. `https://keras.io/`. Consultado el 06-05-2024.

[47] Montreal University. Theano, 2010.

[48] James Moor. The dartmouth college artificial intelligence conference: The next fifty years. *Ai Magazine*, 27(4):87–87, 2006.

[49] Andreas C Müller and Sarah Guido. *Introduction to machine learning with Python: a guide for data scientists*. .ºReilly Media, Inc.", 2016.

[50] Mohammad Najafimehr, Sajjad Zarifzadeh, and Seyedakbar Mostafavi. A hybrid machine learning approach for detecting unprecedented ddos attacks. *The Journal of Supercomputing*, 78, 04 2022.

[51] 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.

[52] 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.

[53] 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.

[54] 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.

[55] Gonzalo Pajares Martinsanz et al. Aprendizaje profundo. 2021.

[56] 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.

[57] 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.

[58] scikit-learn Developers. ssikit-llarn documentation. Consultado el 25-04-2024.

[59] scikit-learn Developers. Train test split de scikit-learn. Consultado el 25-04-2024.

[60] Nitish Srivastava. Improving neural networks with dropout. *University of Toronto*, 182(566):7, 2013.

[61] 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.

[62] Sajedul Talukder. Tools and techniques for malware detection and analysis. *arXiv preprint arXiv:2002.06819*, 2020.

[63] Mingdong Tang and Quan Qian. Dynamic api call sequence visualisation for malware classification. *IET Information Security*, 13(4):367–377, 2019.

[64] Yingjie Tian and Yuqi Zhang. A comprehensive survey on regularization strategies in machine learning. *Information Fusion*, 80:146–166, 2022.

[65] Tokyo University. Chainer, 2015.

[66] 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.

[67] 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.

[68] 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.

[69] 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.

[70] 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.

[71] 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.

[72] 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		