

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

1. Introducción	1
1.1. Motivación y objetivos del trabajo	1
1.2. Contexto y antecedentes del trabajo	1
1.2.1. Redes neuronales	1
1.2.2. Importancia de la detección y prevención de ataques	1
1.2.3. Evolución de las amenazas ciberneticas	1
1.2.4. Avances en el aprendizaje automático para ciberseguridad	1
1.3. Metodología	2
1.4. Estructura de la memoria	2
2. Fundamentos de las redes neuronales	3
2.1. Introducción	3
2.2. Aprendizaje Automático	4
2.3. Aprendizaje profundo	5
2.3.1. Perceptrón	6
2.3.2. El Perceptrón Multicapa y la Retropropagación	7
2.4. Optimización del modelo	8
2.4.1. Descenso de gradiente	9
2.4.2. Descenso de gradiente estocástico	10
2.4.3. Descenso de gradiente por mini-lotes	10
2.4.4. Propagación de la raíz Media Cuadrada	11
2.4.5. Estimación del Momento Adaptativo	11

ÍNDICE GENERAL

2.5. Función de activación.	12
2.6. Función de perdida	16
2.6.1. Clasificación	16
2.6.2. Regresión	17
2.7. Sobreajuste del modelo.	18
2.7.1. Regularización l1 y l2	19
2.7.2. Dropout	20
2.7.3. Parada temprana	21
2.8. Evaluación del modelo.	22
2.8.1. Matriz de Confusión	23
2.8.2. Métricas	24
2.8.3. Curva ROC y AUC	25
2.8.4. Curva de Precisión-Recall	25
2.9. Arquitecturas relevantes	26
2.9.1. Autoencoder	26
2.9.2. Deep Belief Networks	27
2.9.3. Red Neuronal Convolucional	27
2.9.4. Red Neuronal Recurrente	32
2.10. Bibliotecas utilizadas en Python	32
2.10.1. Principales frameworks. Keras	33
2.10.2. Librerías y herramientas esenciales.	34
3. Clasificación de Malware	37
3.1. Microsoft Malware Classification Challenge	37
3.1.1. Distribución del dataset	39
3.2. Red Neuronal Convolutinal	41
3.2.1. Visualizar el malware como imagen	41
3.2.2. Visualización del modelo	42
3.2.3. Aportaciones al modelo	43

ÍNDICE GENERAL

3.2.4. GPU (Unidad de Procesamiento Gráfico)	44
3.2.5. CPU (Unidad Central de Procesamiento)	44
3.3. Autoencoder	45
3.3.1. CAE	45
3.3.2. DAE	45
3.3.3. AE	45
3.4. Resultados	45
4. Detección de intrusiones	47
4.1. KDD Cup 1999	47
4.2. Autoencoder	47
4.3. Red Neuronal Convolutacional	47
4.4. Red Neuronal Profunda	48
4.5. Red Neuronal Recurrente	48
4.6. Restricted Boltzmann Machine	48
4.7. Resultados	48
5. Conclusiones y Trabajo Futuro	49
5.1. Conclusiones	49
5.2. Trabajo futuro	49
Bibliografía	51

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 ciberneticos en entornos empresariales y gubernamentales, así como en la protección de datos sensibles y la infraestructura crítica.

1.2.3. Evolución de las amenazas ciberneticas

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

2.1. 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” [52]. 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) [12].

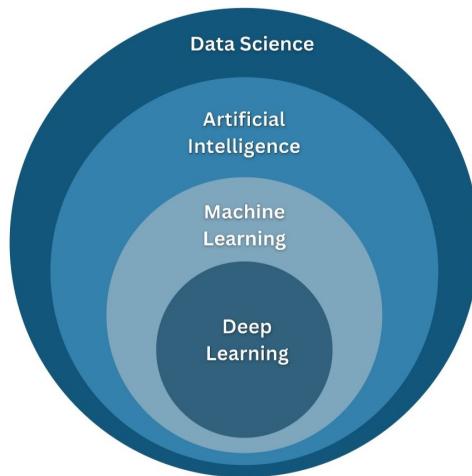


Figura 2.1: Relación entre Ciencia de Datos, Inteligencia Artificial, Machine Learning y Deep Learning.

2.2. Aprendizaje Automático

Dentro de la inteligencia artificial 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 ” [23]. 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 [23]. 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 [23].

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 [59].

En cuanto a la **clasificación de los sistemas de aprendizaje** automático, se distinguen cuatro tipos principales: el aprendizaje supervisado, no supervisado, semisupervisado y por refuerzo.

Un aprendizaje se dice que es **supervisado** si todos los datos tienen asociadas las salidas deseadas (también llamadas etiquetas). Es decir, para todo dato de entrada x se conoce su salida y , luego tenemos información necesaria para aprender de la función que los relaciona $y = f(x)$. El aprendizaje supervisado busca estimar la función \hat{f} tal que \hat{f} sea una buena aproximación de la función f que relaciona las entradas x con las salidas Y .

Un aprendizaje se dice que es **supervisado** si todos los datos tienen asociadas las salidas deseadas

das (también llamadas etiquetas). Es decir, para todo dato de entrada x se conoce su salida y , de tal forma que para una función desconocida f que relaciona ambas, $y = f(x)$. El objetivo del aprendizaje supervisado es estimar la función f con una función \hat{f} de tal forma que \hat{f} sea una buena aproximación de la función f que relaciona las entradas x con las salidas y . Un ejemplo 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 [23].

Un aprendizaje se dice que es **no supervisado** si, al contrario que el aprendizaje supervisado, los datos de entrenamiento no están etiquetados. De esta forma, el modelo tiene que aprender sin supervisión (también llamado sin “profesor”). El objetivo principal de este tipo de algoritmos se basa en el agrupamiento, detección de anomalías o reducción de dimensionalidad [23].

En el caso de encontrarnos ante un problema en el que tengamos datos tanto etiquetados como sin etiquetar, nos encontramos ante 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 [23].

Por último, está el **aprendizaje por refuerzo**, 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 [23].

¿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.

2.3. Aprendizaje profundo

Dentro del Machine Learning, se encuentra el Deep Learning, cuya base son las redes neuronales artificiales (ANN). Las ANN son unas estructuras versátiles, potentes y escalables, lo que las hace ideales para abordar tareas grandes y complicadas para el aprendizaje profundo. Según [59], se definen como un modelo matemático inspirado en la estructura de una red neuronal biológica. Consiste en una red de neuronas interconectadas organizadas por capas, con una capa de entrada, una o más capas ocultas y una capa de salida [18]. 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.

2.3.1. Perceptrón

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

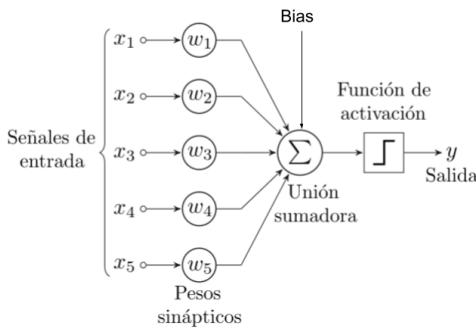


Figura 2.2: Modelo del perceptrón simple

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

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

Los coeficientes w_i , $i = 1, 2, \dots, n$ 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: $f(x) = \sum_{i=1}^n (w_i \cdot x_i) + b = w^t \cdot x + b$ donde $w^t = (w_1, \dots, w_n)$ y $x^t = (x_1, \dots, x_n)$

La segunda capa consiste en transformar la salida de la primera etapa mediante una función de activación. Se podría generalizar de la siguiente manera:

$$y = \phi(w^t \cdot x + b) \quad (2.2)$$

Pero, ¿cómo ajusta los parámetros el perceptrón? El algoritmo de entrenamiento del perceptrón se basa en la regla de aprendizaje de Hebb y tiene en cuenta el error cometido por la red al hacer una predicción. El perceptrón se alimenta con una entrada de entrenamiento a la vez, y para cada una, realiza sus predicciones. Para cada neurona de salida que produce una predicción errónea, se refuerzan los pesos de conexión desde las entradas que habrían contribuido a la predicción correcta [23]. La regla se muestra en la Ecuación 2.3.

$$w_{i,j}^{(t+1)} = w_{i,j}^{(t)} + \eta(y_j - \hat{y}_j)x_i \quad (2.3)$$

En esta ecuación, $w_{i,j}^{(t+1)}$ es el peso de conexión actualizado entre la i -ésima neurona de entrada y la j -ésima neurona de salida, $w_{i,j}^{(t)}$ es el peso de conexión actual entre la i -ésima neurona de entrada y la j -ésima neurona de salida, x_i es el valor de la i -ésima entrada de la instancia de entrenamiento actual, \hat{y}_j es la salida de la j -ésima neurona de salida, y_j es la salida objetivo de la j -ésima neurona de salida y η es la tasa de aprendizaje [23].

El límite de decisión de cada neurona de salida es lineal, por lo que los perceptrones son incapaces de aprender patrones complejos. Esto significa que solo pueden separar los datos si hay una línea recta (o un hiperplano en dimensiones superiores) que divida las distintas clases de datos. Sin embargo, si las instancias de entrenamiento son linealmente separables, Rosenblatt demostró que este algoritmo convergería a una solución, es decir, encontraría un conjunto de pesos que clasifique correctamente todas las instancias de entrenamiento [23]. Esto se conoce como el Teorema de Convergencia del Perceptrón.

2.3.2. El Perceptrón Multicapa y la Retropropagación

Un Perceptrón Multicapa (MLP) está compuesto por una capa de entrada, una o más capas ocultas, y una capa final llamada capa de salida. Todas las capas, excepto la capa de salida, incluyen una neurona de sesgo y están completamente conectadas a la siguiente capa [23].

Cuando una red neuronal artificial (ANN) contiene un conjunto profundo de capas ocultas¹, se llama red neuronal profunda (DNN). El campo del Aprendizaje Profundo estudia las DNNs, y más generalmente modelos que contienen conjuntos profundos de cálculos. Sin embargo, muchas personas hablan de Aprendizaje Profundo siempre que están involucradas redes neuronales.

David Rumelhart, Geoffrey Hinton y Ronald Williams publicaron un artículo revolucionario que introdujo el algoritmo de entrenamiento de retropropagación, que todavía se usa hoy en día. Consiste en un Descenso de Gradiente que utiliza una técnica eficiente para calcular los gradientes automáticamente. En solo dos etapas (una hacia adelante, una hacia atrás), el algoritmo de retropropagación puede determinar cómo se deben ajustar cada peso de conexión y cada término de sesgo para reducir el error. Para ello, calcula los gradientes del error y a continuación, simplemente realiza un paso regular de Descenso de Gradiente, y todo el proceso se repite hasta que la red converge a la solución.

El algoritmo maneja un mini-lote a la vez (batch size), y pasa por el conjunto completo de entrenamiento múltiples veces. Cada pasada se llama época (epoch). Cada mini-lote se pasa a la capa de entrada de la red, que lo envía a la primera capa oculta. El algoritmo luego calcula la salida de todas las neuronas en esta capa (para cada instancia en el mini-lote). El resultado se pasa a la siguiente capa, su salida se calcula y se pasa a la siguiente capa, y así sucesivamente hasta obtener la salida de la última capa, la capa de salida. Esta es la pasada hacia adelante: es exactamente como hacer predicciones, excepto que todos los resultados intermedios se preservan

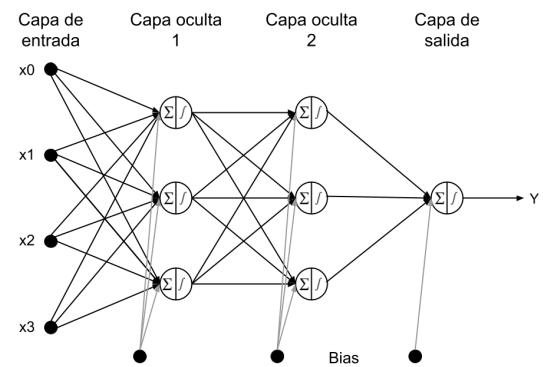


Figura 2.3: Arquitectura de un Perceptrón Multicapa con tres entradas, dos capas ocultas de tres neuronas cada una y una neurona de salida (las neuronas de sesgo se muestran aquí, pero normalmente son implícitas).

¹Se suele considerar más de dos capas ocultas como un conjunto profundo de capas ocultas

ya que son necesarios para la pasada hacia atrás. A continuación, el algoritmo mide el error de salida de la red (es decir, utiliza una función de pérdida que compara la salida deseada y la salida real de la red, y devuelve alguna medida del error). Luego, calcula cuánto contribuyó cada conexión de salida al error. Esto se hace analíticamente aplicando la regla de la cadena, lo que hace que este paso sea rápido y preciso. El algoritmo luego mide cuánto de estas contribuciones de error provienen de cada conexión en la capa inferior, nuevamente utilizando la regla de la cadena, trabajando hacia atrás hasta que el algoritmo llega a la capa de entrada. Como se explicó anteriormente, esta pasada inversa mide eficientemente el gradiente de error en todos los pesos de conexión en la red propagando el gradiente de error hacia atrás a través de la red (de ahí el nombre del algoritmo). Finalmente, el algoritmo realiza un paso de Descenso de Gradiente para ajustar todos los pesos de conexión en la red, utilizando los gradientes de error que acaba de calcular [23].

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

Es importante inicializar aleatoriamente todos los pesos de conexión de las capas ocultas, de lo contrario, el entrenamiento fallará. Por ejemplo, si inicializas todos los pesos y sesgos a cero, entonces todas las neuronas en una capa dada serán perfectamente idénticas, y por lo tanto la retropropagación las afectará de la misma manera, por lo que permanecerán idénticas. En otras palabras, a pesar de tener cientos de neuronas por capa, tu modelo actuará como si tuviera solo una neurona por capa: no será muy inteligente. Si en cambio inicializas aleatoriamente los pesos, rompes la simetría y permites que la retropropagación entrene un equipo diverso de neuronas [23].

2.4. Optimización del modelo

MEJOR CONEXIÓN DE INTRODUCCIÓN

La optimización consiste en minimizar o maximizar una función $f(\mathbf{x})$ modificando \mathbf{x} . En general, la optimización hace referencia a una minimización, ya que la maximización es equivalente a la minimización de $-f(\mathbf{x})$. La función a minimizar se conoce como función objetivo, función de coste, función de pérdida o función de error. La función de pérdida calcula la discrepancia entre la salida de la red neuronal (predicción) y la etiqueta verdadera. Su objetivo es minimizar este error a lo largo del entrenamiento. A la función o grupo de funciones que se están minimizando se denominan *loss functions*, y el valor mínimo se identifica mediante el símbolo $x^* = \arg \min f(\mathbf{x})$ [59].

Se sabe de Cálculo Diferencial que la derivada de una función en un punto x calcula la pendiente de la función en dicho punto. Sea δ un valor, entonces se tiene que $f(x + \delta) \approx f(x) + f'(x)\delta$, es decir, un pequeño cambio en la entrada permite aproximar su valor en la salida gracias a la derivada de la función. Para poder utilizar esta aproximación, es necesario que la función sea diferenciable². Este método se conoce como descenso de gradiente (*gradient descent*). Durante este método, se propaga el error hacia atrás a través de la estructura del modelo, de forma que

²En la red neuronal se realizan cálculos en cada capa y si estos cálculos son diferenciables, la función de pérdida también lo será. Sin embargo, a veces se usan funciones de activación como ReLU, que no son diferenciables en todos los puntos. En estos casos, se debe aplicar una aproximación de la derivada para poder utilizar la función en el método de descenso de gradiente.

los pesos en las redes neuronales vayan ajustándose a lo largo del entrenamiento. Esto cierra el ciclo de aprendizaje entre el envío de los datos hacia adelante, la generación de predicciones y la mejora durante la propagación hacia atrás. Al adaptar los pesos, el modelo probablemente mejore (a veces mucho, a veces ligeramente) y, por lo tanto, se dice que se ha realizado el aprendizaje [59].

2.4.1. Descenso de gradiente

Sea $f : S \in \mathbb{R}^n \rightarrow \mathbb{R}^m$, donde n y m son enteros. Cuando $f'(x) = 0$, la derivada no proporciona información sobre la dirección en la que hay que moverse. De esta forma, los puntos con $f'(x) = 0$ se conocen como puntos críticos o puntos estacionarios. Un punto x_0 es un **mínimo local** si $\exists \epsilon$ tal que $f(x_0) \leq f(x)$ para todo $x \in B(x_0, \epsilon)$ ³. Por otra parte, un punto x_0 es un **punto de inflexión** de la función f si f es continua en x_0 y existe un entorno $B(x_0, \epsilon)$ tal que f'' cambia de signo en x_0 . Esto implica que en un lado de x_0 , $f''(x)$ es positiva y en el otro lado es negativa, o viceversa. Si un punto x_0 cumple que $\exists \epsilon$ tal que $f(x_0) \leq f(x)$ para todo $x \in S$, entonces x_0 es un **mínimo global**. Puede haber solo un mínimo global o múltiples mínimos globales de la función. También es posible que haya mínimos locales que no sean globalmente óptimos. En el contexto del aprendizaje automático, se optimizan funciones que pueden tener muchos mínimos locales que no son óptimos y muchos puntos de inflexión rodeados de regiones muy planas. Todo esto hace que la optimización sea difícil, especialmente cuando la entrada a la función es multidimensional. Por lo tanto, se suele conformar con encontrar un valor que sea muy bajo pero no necesariamente mínimo en ningún sentido formal [59].

A menudo es necesario minimizar funciones que tienen múltiples (n) entradas: $f : \mathbb{R}^n \rightarrow \mathbb{R}$, con una única salida. Con múltiples entradas, se debe utilizar el concepto de derivadas parciales $\frac{\partial f}{\partial x_i}$, que mide cómo cambia f según aumenta la variable x_i , en el punto \mathbf{x} . El gradiente de f es el vector que contiene todas las derivadas parciales, denotado por $\nabla f(\mathbf{x})$. En múltiples dimensiones, los puntos críticos son puntos en los que cada elemento del gradiente es igual a cero. La derivada direccional en la dirección \mathbf{u} (vector unitario) es la pendiente de la función f en esa dirección \mathbf{u} . En otras palabras, la derivada direccional es la derivada de la función $f(\mathbf{x} + \alpha\mathbf{u})$ con respecto a α , evaluada en $\alpha = 0$. Usando la regla de la cadena, se determina que $f(\mathbf{x} + \alpha\mathbf{u})$ evalúa $\mathbf{u} \cdot \nabla f(\mathbf{x})$ cuando $\alpha = 0$ [59].

Para minimizar f , es deseable encontrar la dirección en la que f disminuye de forma más rápida, lo que se puede hacer usando la derivada direccional:

$$\min_{\mathbf{u}} D_{\mathbf{u}} f(\mathbf{x}) = \min_{\mathbf{u}} \nabla f(\mathbf{x}) \cdot \mathbf{u} = -\|\nabla f(\mathbf{x})\|, \quad (2.4)$$

donde θ es el ángulo entre \mathbf{u} y el gradiente. Sustituyendo $\|\mathbf{u}\|_2 = 1$ e ignorando factores que no dependen de u , se concreta en minimizar $\min_u \cos \theta$. Se puede hacer decrecer f moviéndose en la dirección opuesta del gradiente, lo que se conoce como método del gradiente descendente por lotes (*steepest descent* o *gradient descent*), que propone un nuevo punto:

$$\mathbf{x}' = \mathbf{x} - \eta \nabla f(\mathbf{x}), \quad (2.5)$$

donde η es la razón de aprendizaje (o *learning rate*), que determina el tamaño del paso. En caso de aplicar esta ecuación a cada uno de los datos de entrada, se llamará **descenso de gradiente por lotes**. El learning rate se puede elegir de varias formas, una de ellas es fijarlo a un valor constante pequeño. Algunas veces se puede determinar la dimensión del paso que hace desaparecer la derivada direccional. Otra aproximación consiste en evaluar $f(\mathbf{x} - \eta \nabla f(\mathbf{x}))$

³ $B(a, b)$ hace referencia a la bola o disco de centro a y radio b

para varios valores de η y escoger el que proporciona el valor de la función objetivo menor. Esta estrategia se denomina búsqueda en línea. El gradiente descendente converge cuando cada elemento del gradiente es cero o muy próximo a él [59]. Escoger valores de η muy pequeños hace que la convergencia sea muy lenta. Sin embargo, valores muy grandes pueden hacer que el algoritmo diverja y se aleje en cada paso más del valor óptimo.

También esta la posibilidad de añadirle momentum al SGD, que es un hiperparámetro mayor o igual a 0 que acelera el descenso del gradiente en la dirección relevante y amortigua las oscilaciones. Su valor por defecto es 0.

2.4.2. Descenso de gradiente estocástico

El Descenso de Gradiente Estocástico (SGD, por sus siglas en inglés) es un algoritmo utilizado para minimizar una función objetivo en problemas de aprendizaje automático.

$$w^{(t+1)} = w^{(t)} - \eta \nabla J_i(w^{(t)})$$

donde w son los pesos que desean estimarse en cada iteración y J_i es la i -ésima observación del conjunto de datos. El SGD selecciona aleatoriamente una elemento del conjunto de entrenamiento en cada iteración y calcula los gradientes basándose únicamente en esa instancia individual [59]. Esto hace que el algoritmo sea mucho más rápido que calculándolo para todos los datos, debido a que en cada iteración solo se necesita manipular una pequeña cantidad de datos. Además, permite entrenar con conjuntos de datos muy grandes. Sin embargo, debido a su naturaleza aleatoria, el SGD es mucho menos regular que el Descenso de Gradiente por Lote. En lugar de disminuir suavemente hasta alcanzar el mínimo, la función de perdida oscilará, disminuyendo solo en promedio. Con el tiempo, el algoritmo se acercará mucho al mínimo, pero continuará oscilando alrededor de él sin asentarse completamente [23].

Una solución a este problema es reducir gradualmente la tasa de aprendizaje. Al principio, los pasos son grandes, lo que ayuda a hacer un progreso rápido y a escapar de mínimos locales. Luego, los pasos se hacen más pequeños, permitiendo que el algoritmo se asiente en el mínimo global [23].

Los pasos de este método de optimización son: Primero, se elige un vector inicial de parámetros w (que puede ser seleccionado aleatoriamente) y una razón de aprendizaje η . Los siguientes pasos se repiten hasta que se consiga un mínimo aproximado. En cada iteración, se seleccionan aleatoriamente ejemplos del conjunto de entrenamiento. Para cada ejemplo $i = 1, 2, \dots, n$, se actualiza el vector de parámetros utilizando la fórmula $w^{(t+1)} = w^{(t)} - \eta \nabla J_i(w^{(t)})$ [59].

2.4.3. Descenso de gradiente por mini-lotes

El Descenso de Gradiente por Mini-lote (Mini-batch Gradient Descent) es un algoritmo que combina aspectos del Descenso de Gradiente por Lote y el SGD (Stochastic Gradient Descent). A diferencia de estos métodos, en lugar de calcular los gradientes basándose en todo el conjunto de entrenamiento o en una única instancia, el Descenso de Gradiente por Mini-lote calcula los gradientes basándose en pequeños subconjuntos aleatorios de instancias llamados mini-lotes (mini-batch). Estos subconjuntos definen el número de muestras utilizadas antes de actualizar los pesos de la red en el modelo. Si se utilizan todas las muestras de entrenamiento para crear un lote, se denomina **descenso del gradiente por lotes**. Si el lote tiene el tamaño de una muestra, se llama **descenso de gradiente estocástico**. Cuando el lote tiene más de una

muestra pero es menor que el tamaño del conjunto de datos de entrenamiento, se denomina **descenso de gradiente de mini-lote**. Este enfoque logra un progreso menos errático que el SGD, especialmente con mini-lotes relativamente grandes, y se acerca más al mínimo, aunque puede ser más difícil escapar de mínimos locales. No obstante, con un buen plan de aprendizaje, tanto el SGD como el Descenso de Gradiente por Mini-lote pueden llegar al mínimo [23].

Otro parámetro relevante es la **época (epoch)**, que define cuántas veces el conjunto total de muestras es procesado por el algoritmo. Cada muestra del conjunto N tiene una oportunidad de actualizar los pesos en cada epoch. Una epoch puede contener uno o más lotes. Para mayor claridad, se puede pensar en un bucle sobre el número de epochs, donde cada iteración opera sobre N . Dentro de esta iteración, existe otra iteración anidada que actúa sobre cada lote de muestras. El número de epochs varía según el tipo de datos y puede ser alto, como 10, 100, 1000 o más, garantizando una eficiencia suficiente, aunque también puede ser menor [59].

2.4.4. Propagación de la raíz Media Cuadrada

Además de estos métodos, existen optimizadores avanzados que mejoran la eficiencia del entrenamiento. Entre ellos se encuentra la **Propagación de la Raíz Media Cuadrática (RMS-Prop)**. Este método adapta la tasa de aprendizaje para cada parámetro manteniendo una media móvil de los cuadrados de los gradientes. La actualización del parámetro $w(t)$ se realiza de la siguiente manera [59]:

$$v(t) = \beta v(t-1) + (1 - \beta)(g(t))^2$$

$$w(t) = w(t-1) - \frac{\eta}{\sqrt{v(t)} + \epsilon} g(t)$$

donde β es el factor de decaimiento de la media, η es la tasa de aprendizaje y ϵ es una constante pequeña para evitar divisiones por cero [59].

2.4.5. Estimación del Momento Adaptativo

Otro optimizador relevante es **Adam (Adaptive Moment Estimation)**, que combina las ideas de RMSProp con el momento. Adam mantiene medias móviles de los gradientes y de los cuadrados de los gradientes. Las actualizaciones de los parámetros se realizan de la siguiente manera [59]:

$$m(t) = \beta_1 m(t-1) + (1 - \beta_1)g(t)$$

$$v(t) = \beta_2 v(t-1) + (1 - \beta_2)(g(t))^2$$

$$\hat{m}(t) = \frac{m(t)}{1 - \beta_1^t}$$

$$\hat{v}(t) = \frac{v(t)}{1 - \beta_2^t}$$

$$w(t) = w(t-1) - \frac{\eta \hat{m}(t)}{\sqrt{\hat{v}(t)} + \epsilon}$$

donde β_1 y β_2 son factores de decaimiento de las medias, η es la tasa de aprendizaje y ϵ es una constante pequeña para evitar divisiones por cero [59].

2.5. Función de activación.

Entre capa y capa de las redes neuronales, cada neurona suma ponderadamente las entradas provenientes de las neuronas de la capa anterior. Este resultado, se lo pasa a una función no lineal, llamada función de activación, que permiten que la red neuronal pueda aprender, representar patrones complejos y crear relaciones no lineales entre las características de entrada y la salida. El valor resultante de aplicarle la función es el valor que se le queda asociado a esta neurona, para que futuras capas puedan usarlo. En esta sección, describimos varias funciones de activación populares, sus propiedades y aplicaciones [59].

Softmax

La función **Softmax** se utiliza comúnmente en la capa de salida de redes neuronales para problemas de clasificación multiclas. 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:

$$\text{Softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^K e^{z_j}} \quad (2.6)$$

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 [59].

Sigmoide

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

$$\sigma(x) = \frac{1}{1 + e^{-x}} \quad (2.7)$$

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 tienen esperanza de la función de salida positiva, lo que puede llevar a una convergencia lenta [59].

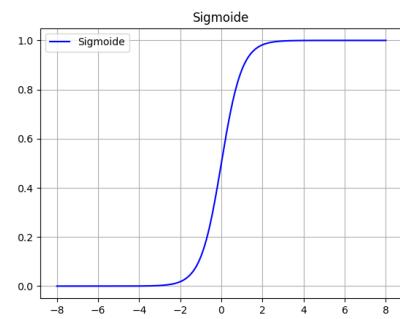


Figura 2.4: Gráfica de la función sigmoide.

Sigmoide dura

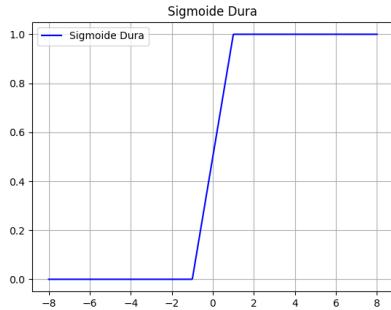


Figura 2.5: Gráfica de la función sigmoide dura.

Para solucionar el problema de la saturación del gradiente en los extremos, está la función **sigmoide dura** (*hard sigmoid*), que es una aproximación simplificada de la función sigmoide. Su fórmula es la siguiente:

$$f(x) = \max(0, \min(1, \frac{x+1}{2})) \quad (2.8)$$

Esta función es computacionalmente más eficiente y evita los problemas de saturación en los extremos [59].

Función Tangente hiperbólica

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

$$\tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}} \quad (2.9)$$

Aunque también sufre de saturación del gradiente, *tanh* centra las salidas alrededor de cero, lo que puede acelerar el entrenamiento [59].

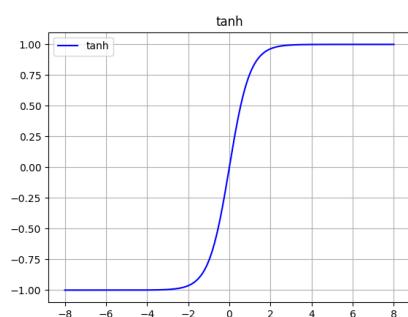


Figura 2.6: Gráfica de la función tangente hiperbólica.

Función ReLU

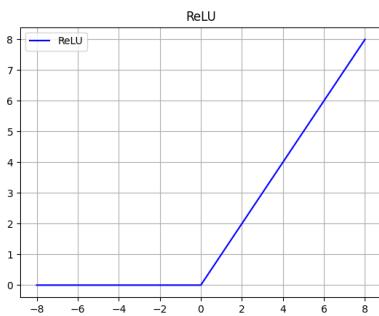


Figura 2.7: Gráfica de la función ReLU.

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

$$f(x) = \max(0, x) \quad (2.10)$$

Sin embargo, la función ReLU tiene algunos problemas, como el problema de la “ReLU muerta” [45] y la falta de diferenciabilidad en cero. Para este problema último, se suele asignar un valor arbitrario para este valor como 0, 0.5 o 1 citeppajares2021aprendizaje. Esto ocurre cuando se aprende un sesgo negativo grande, haciendo que la salida de la neurona sea siempre cero, sin importar la entrada [5]. Por eso, a continuación, discutimos algunas variantes de ReLU.

Función Leaky ReLU (LReLU)

Una de las primeras funciones de activación basadas en ReLU fue **LReLU** [45]. La función LReLU fue un intento de aliviar los problemas potenciales de ReLU mencionados anteriormente. Se define como:

$$f(x) = \max(0, 0.001x, x) \quad (2.11)$$

La función de activación Leaky ReLU permite que la neurona tenga un pequeño gradiente cuando su salida es cero o negativa (es decir, cuando $x \leq 0$). Sin embargo, se ha demostrado que Leaky ReLU funciona de manera casi idéntica a la ReLU original, por lo que no mejora significativamente el rendimiento de la red. Además, se propuso una versión aleatoria de ReLU, donde el valor de x se elige aleatoriamente de una distribución uniforme $U(l, u)$ con $0 \leq l < u < 1$ [5].

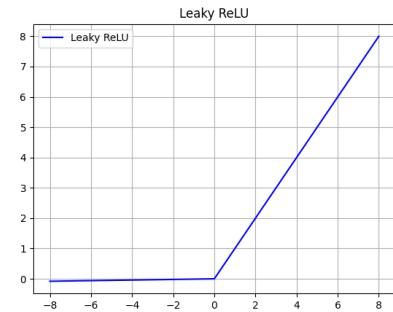


Figura 2.8: Gráfica de la función Leaky ReLU.

Función paramétrica ReLU(PReLU)

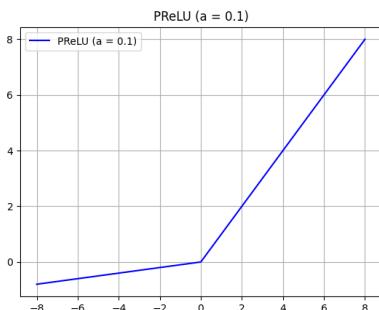


Figura 2.9: Gráfica de la función Paramétrica ReLU.

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:

$$f(x; \alpha) = \begin{cases} \alpha x & \text{si } x < 0 \\ x & \text{si } x \geq 0 \end{cases} \quad (2.12)$$

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

Función Softplus

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

$$\text{softplus}(a) = \log(1 + \exp(a)) \quad (2.13)$$

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

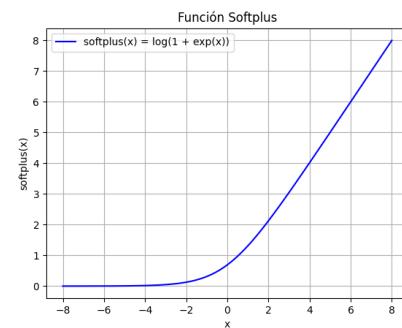


Figura 2.10: Gráfica de la función Softplus.

Función ELU

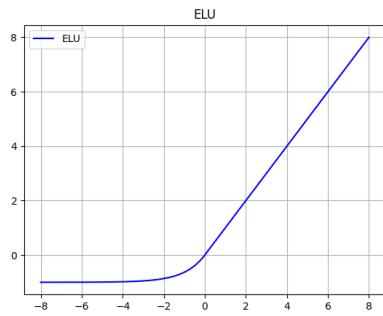


Figura 2.11: Gráfica de la función ELU.

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

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

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

Funciones Swish y SiLU

Swish y **SiLU** combinan la función sigmoidal con otros parámetros para proporcionar una activación suave y no lineal. Ambas funciones se han probado en tareas de aprendizaje por refuerzo y en otros contextos, demostrando su efectividad [5].

$$\text{SiLU}(a) = a \cdot \sigma(a) = \frac{a}{1 + e^{-a}} \quad (2.14)$$

La derivada de SiLU se define como:

$$d\text{SiLU}(a) = \sigma(a) (1 + a(1 - \sigma(a))) \quad (2.15)$$

Donde $\sigma(a)$ es la función sigmoidal.

La función Swish se define de manera similar, pero con un parámetro adicional β :

$$\text{Swish}(x) = x \cdot \sigma(\beta x) = \frac{x}{1 + e^{-\beta x}} \quad (2.16)$$

Para $\beta = 1$, Swish es equivalente a SiLU. La Figura 2.14 muestra las gráficas de las funciones Swish (con $\beta = 3$) y SiLU.

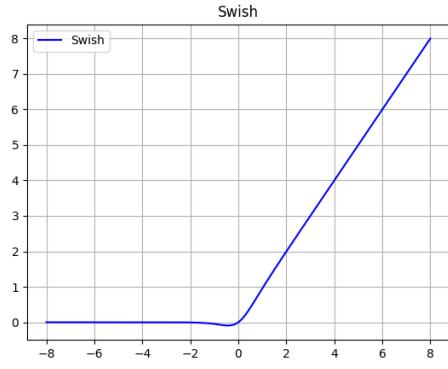


Figura 2.12: Gráfica de la función Swish con $\beta = 3$.

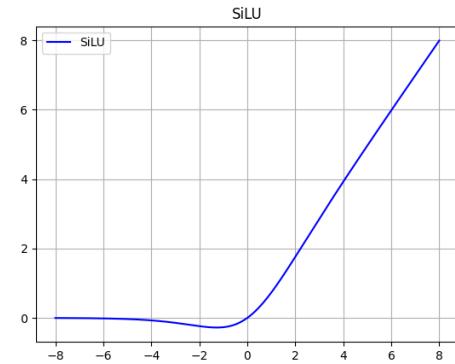


Figura 2.13: Gráfica de la función SiLU.

Figura 2.14: Comparación de las funciones de activación Swish y SiLU.

Las funciones de activación juegan un papel esencial en la formación de redes neuronales al permitir que las redes aprendan y generen relaciones complejas. Desde funciones clásicas como la sigmoide y tanh, hasta variantes más recientes como ReLU, LReLU y ELU, cada función tiene sus propios beneficios y limitaciones. La elección de la función de activación depende del tipo de problema al que nos enfrentemos y de lo que se busque con el modelo, pero tiene que ser una elección acertada, ya que puede influir significativamente en el rendimiento del modelo. Además, permiten resolver el desvanecimiento del gradiente durante el proceso de aprendizaje, evitando que el gradiente se acerque a cero, cumpliendo la hipótesis de derivabilidad [59].

2.6. 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 objetivo es cuantificar el grado de error en las predicciones realizadas por el modelo, lo cual es fundamental para el ajuste de sus parámetros durante el entrenamiento. En términos generales, estas funciones se dividen en dos grandes categorías: las utilizadas para problemas de clasificación y las utilizadas para problemas de regresión [59]. Ambas buscan minimizar el error, pero lo hacen con diferentes enfoques dependiendo de la naturaleza de la variable de salida.

2.6.1. 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:

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

$$H(p, q) = - \sum_x p(x) \log q(x) \quad (2.17)$$

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 [59].

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

$$H(p, q) = -[p \log q + (1 - p) \log(1 - q)] \quad (2.18)$$

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 [59].

Coeficiente de Dice

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

$$D = \frac{2|P \cap G|}{|P| + |G|} \quad (2.19)$$

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 [59].

2.6.2. Regresión

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

Error Cuadrático Medio (ECM)

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

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2 \quad (2.20)$$

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 [59].

Error Absoluto Medio (EAM)

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

$$\text{MAE} = \frac{1}{n} \sum_{i=1}^n |y_i - \hat{y}_i| \quad (2.21)$$

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 [59].

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:

$$L_\tau(y, \hat{y}) = \sum_{i=1}^n (\tau \max(y_i - \hat{y}_i, 0) + (1 - \tau) \max(\hat{y}_i - y_i, 0)) \quad (2.22)$$

donde τ es el cuantil elegido.

2.7. Sobreajuste del modelo.

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

El **sobreajuste** (también conocido como *overfitting*) se produce cuando un modelo se ajusta demasiado bien a los datos de entrenamiento, capturando incluso el ruido y las fluctuaciones aleatorias. Este ajuste excesivo provoca que el modelo tenga un rendimiento excelente en los

datos de entrenamiento pero un rendimiento deficiente en datos nuevos. El overfitting suele ocurrir cuando el modelo es demasiado complejo en comparación con la cantidad de datos disponibles.

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

En la primera gráfica de la Figura 2.15, 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.

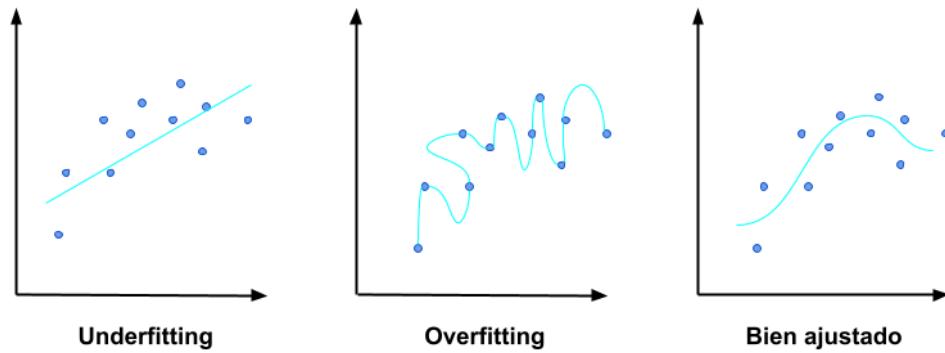


Figura 2.15: Ejemplo de sobreajuste, subajuste y buen ajuste.

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

2.7.1. Regularización l_1 y l_2

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 [59].

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 [23].

$$\text{loss}_{\text{regul}}(\theta) = \text{loss}(\theta) + \beta \cdot f(\theta) \quad (2.23)$$

Aquí, $\text{loss}_{\text{regul}}(\theta)$ es la función de pérdida final después de aplicarle una regularización. $\text{loss}(\theta)$ 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:

$$\text{Loss} = \text{Original Loss} + \lambda \sum_i w_i^2 \quad (2.24)$$

donde λ es el parámetro de regularización que controla la importancia de la penalización y w_i los pesos [23]. 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:

$$\text{Loss} = \text{Original Loss} + \lambda \sum_i |w_i| \quad (2.25)$$

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 $l2$ [23].

2.7.2. Dropout

Otro método para evitar el sobreajuste de nuestro modelo es el Dropout, que fue propuesto por Hinton et al. [30] como una forma de regularización para capas de redes neuronales completamente conectadas. El dropout consiste en que cada elemento de la salida de una capa se mantiene en cada iteración con una probabilidad p (“tasa de dropout”), de lo contrario se establece en 0, con una probabilidad $(1 - p)$. Según [23], este valor de p suele variar entre 0.1 y 0.5, siendo este último uno de los más típicos [64], aunque la red neuronal que se esté utilizando influye en el valor óptimo [23]. La elección de este parámetro es muy importante, ya que si tomamos un valor muy alto, puede llevarnos a underfitting.

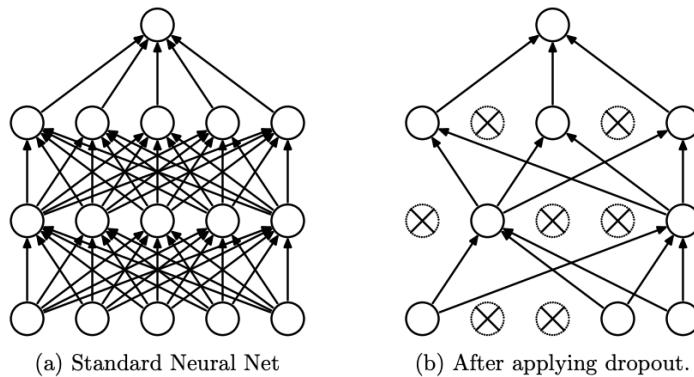


Figura 2.16: Esquema de la técnica dropout. Fuente [65]

Al eliminar una neurona, esta se elimina de la red con todas sus conexiones entrantes y salientes (ver Figura 2.16). 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 [64]. Puede interpretarse como entrenar una gran colección de redes neuronales diferentes y usar sus promedios para hacer predicciones.

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

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

```

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

```

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

2.7.3. Parada temprana

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

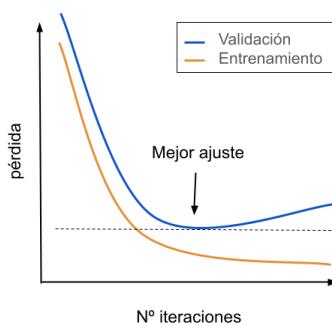


Figura 2.17: Curvas de error de entrenamiento y validación con Early Stopping.

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 [23]. La Figura 2.17 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 [68].

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

```

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

```

En este ejemplo, EarlyStopping detiene el entrenamiento después de un número definido de épocas sin mejora (`patience = 10`), restaurando los pesos que lograron el mejor desempeño [23].

Así, *Early Stopping* se complementa eficazmente con las técnicas de regularización l_1 , l_2 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 l_2 , 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 [23, 59].

2.8. Evaluación del modelo.

CYA HE ESTUDIADO SUPERVISADO, NO SUPERVISADO Y LOS MÉTODOS DE ML.

Una vez discutidos los fundamentos del aprendizaje supervisado y no supervisado, y explorado una variedad de algoritmos de machine learning, profundizaremos ahora en la evaluación de los diferentes modelos. Nos centraremos en los métodos supervisados, tanto en regresión como en clasificación, ya que la evaluación y selección de modelos en aprendizaje no supervisado suele ser un proceso más cualitativo [53].

Para evaluar los modelos supervisados, primero se divide el conjunto de datos en un conjunto de entrenamiento y un conjunto de prueba utilizando la función `train_test_split`, que divide los datos de manera aleatoria. Algunos modelos como la CNN o los autoencoders también necesitan un conjunto de validación durante su entrenamiento para ir evaluando el rendimiento del modelo en cada iteración. Además, proporciona información muy valiosa para detectar si un modelo empieza a sobreaprender de los datos de entrenamiento. Los porcentajes de estos subconjuntos suelen encontrarse entre el 70 - 80 %, 10 - 20 % y 10 - 20 % para entrenamiento, test y validación respectivamente. El conjunto de entrenamiento sirve para entrenar la red neuronal. Por otro lado, el conjunto de prueba sirve para medir definitivamente el rendimiento del modelo con nuevas muestras de datos que no han sido utilizadas durante del entrenamiento para evaluar el modelo [53].

El problema de esta técnica radica en que las distribuciones de los datos pueden no ser uniformes en los tres subconjuntos y que no sea una división muy óptima. Para evitar este inconveniente, se puede usar una técnica de evaluación del modelo más avanzada llamada **validación cruzada**

(*cross-validation*), donde los datos se dividen en varios subconjuntos, y el modelo se entrena y valida múltiples veces, garantizando una evaluación más robusta del rendimiento. La variante más comúnmente utilizada es la validación cruzada *k-fold*. Aquí, el conjunto de datos se divide en k partes aproximadamente iguales, llamadas pliegues. Cada uno de los k pliegues se utiliza una vez como conjunto de prueba, mientras que los $k - 1$ pliegues restantes se combinan para formar el conjunto de entrenamiento. Este proceso se repite k veces, generando k modelos y k evaluaciones. La Figura ?? ilustra este procedimiento [53].

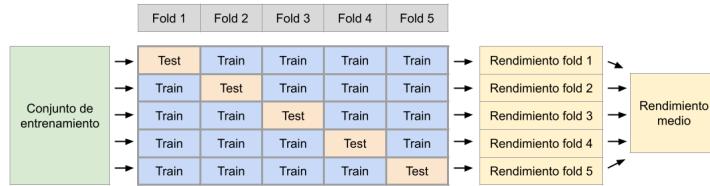


Figura 2.18: Validación cruzada *k-fold*.

Al final de la validación cruzada *k-fold*, se recopilan k valores de desempeño (por ejemplo, precisión en clasificación) que se pueden promediar para obtener una estimación más robusta del rendimiento del modelo. La validación cruzada no solo proporciona una evaluación más precisa del modelo, sino que también ayuda a detectar variaciones en el desempeño debido a diferentes particiones de datos, ofreciendo así una visión más completa de la capacidad del modelo para generalizar [53]. Aunque la validación cruzada ofrece bastantes beneficios, su uso es menos frecuente en redes neuronales debido a los altos costos computacionales asociados [23]. En su defecto, es más frecuente usar `train_test_split()`.

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

2.8.1. Matriz de Confusión

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

La matriz se define como:

	Predictión Positiva	Predictión Negativa
Real Positiva	TP	FN
Real Negativa	FP	TN

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

2.8.2. Métricas

La matriz de confusión ofrece una gran cantidad de información, pero a veces te interesa una métrica más concreta para evaluar un modelo de clasificación, como la sensibilidad, la tasa de falsos negativos o su sensibilidad. A continuación vamos a definir y explicar las principales métricas a tener en cuenta según [20] 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:

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

2.8.3. Curva ROC y AUC

La curva ROC (Receiver Operating Characteristic) es otra herramienta utilizada en los modelos de clasificación [23]. 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 2.19 indica la curva ROC de un clasificador puramente aleatorio [23].

Otro término que está relacionado es el AUC (*area under the curve*). El AUC es el área que se encuentra encerrada entre la curva ROC, la recta $y = 0$ y $x = 1$. Cuantifica la capacidad que tiene el modelo para distinguir entre las diferentes clases. Cuanto mayor valor, mejor rendimiento. En la Figura 2.19 se representa con rallas de intensidad baja.

La interpretación de estos términos consiste en que una curva ROC más cercana al vértice superior izquierdo, o un AUC cercano a 1, indica un mejor rendimiento del modelo [23].

2.8.4. Curva de Precisión-Recall

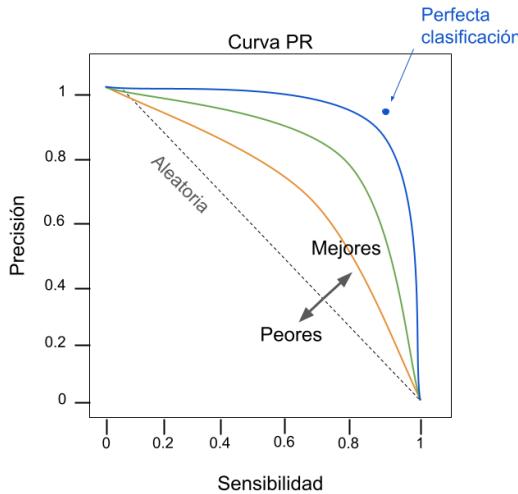


Figura 2.20: Curva de Precisión-Recall.

ta 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

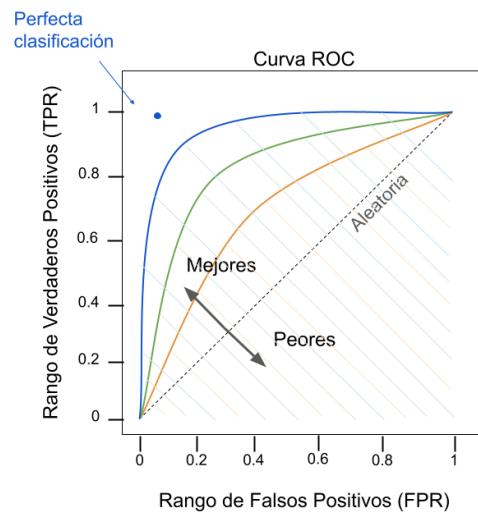


Figura 2.19: Interpretación de la curva ROC con el área bajo la curva (AUC).

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 deseé, pero a costa de la sensibilidad. Por ejemplo, si se desea una precisión del 90 %, se puede aumentar el umbral has-

proporciona la precisión deseada. La Figura 2.21 ilustra cómo se comportan la precisión y la sensibilidad en función del umbral de decisión [23].

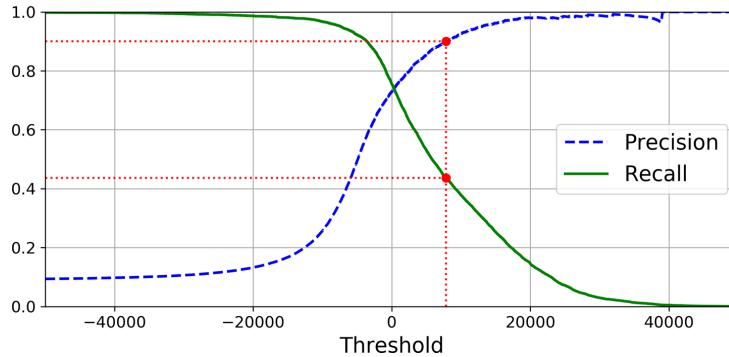


Figura 2.21: Precisión y Sensibilidad vs Umbral de decisión. Fuente [23]

2.9. 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

2.9.1. 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 [43]. La arquitectura básica de un autoencoder consta de tres partes: el encoder, el cuello de botella y el decoder (Figura 2.22).

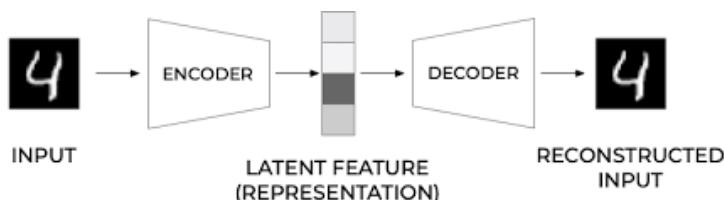


Figura 2.22: Arquitectura de un autoencoder. Fuente:[35].

el autoencoder ilustrado en la figura ?? puede modelarse mediante la ecuación (2.26).

$$\begin{cases} z = f(x; w_e, b_e) \\ r = g(z; w_d, b_d) \end{cases} \quad (2.26)$$

donde $f(\cdot)$ y $g(\cdot)$ son funciones del codificador y del decodificador que generalmente se implementan mediante redes neuronales. Los parámetros w_e y b_e son del codificador, y w_d y b_d son del decodificador. Si $f(\cdot)$ y $g(\cdot)$ son redes neuronales, entonces w_i y b_i son las matrices de pesos y los vectores de sesgo con respecto a la red neuronal del codificador y la red neuronal del

decodificador ($\forall i \in e, d$). El codificador (encoder) mapea el conjunto de datos x al espacio de características z que normalmente suene tener una menor dimensión. Por otra parte, el decodificador (decoder) reconstruye los datos iniciales x a través de la representación comprimida oculta z [76]. Durante el entrenamiento, los parámetros del autoencoder se optimizan para minimizar la diferencia entre la entrada y la salida reconstruida, utilizando una función de pérdida que mide esta discrepancia, como por ejemplo la `binary_crossentropy` o el `mse`. La expresión matemática puede expresarse como:

$$\arg \min_{A,B} \mathbb{E}[\Delta(x, B \circ A(x))] \quad (2.27)$$

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 [6]. Esto concluye el proceso de entrenamiento de un autoencoder.

Las ecuaciones para obtener la salida de un autoencoder serían:

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

donde x es el input, $b^{(1)}$ y $b^{(2)}$ son los sesgos, $W^{(1)}$ y $W^{(2)}$ son los pesos, $z^{(1)}$ es la salida lineal de la primera capa, $a^{(2)}$ es la activación de la segunda capa, $z^{(2)}$ es la salida lineal de la segunda capa e y es la salida final del modelo [48].

Tradicionalmente, los autoencoders se empleaban principalmente para reducir la dimensionalidad o para extraer características. No obstante, con el auge de diversos modelos de aprendizaje profundo, especialmente los modelos de redes generativas adversarias, los autoencoders han cobrado relevancia en la modelización generativa. Esto ha llevado a numerosos investigadores a proponer una variedad de modelos extendidos de autoencoders, como las convolutional autoencoders (que veremos en más profundidad en la sección 2.9.3) o los recurrentes [76].

Las principales capas que se utilizan en esta red neuronal son las capas densas y las de aplanoamiento, aunque también se pueden utilizar capas convolucionales traspuestas y capas unpooling (sección 2.9.3) para autocodificadores convolucionales⁴ o LSTM en el caso de autocodificadores recurrentes⁵ [23].

2.9.2. Deep Belief Networks

Red Neuronal Profunda

2.9.3. 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 [61]. La arquitectura de una

⁴Autocodificador para imágenes de gran tamaño

⁵Autocodificador específico para series temporales o secuencias.

CNN (Figura 2.23) consta de tres tipos de capas: capas de convolución, capas de pooling y la capa de clasificación.

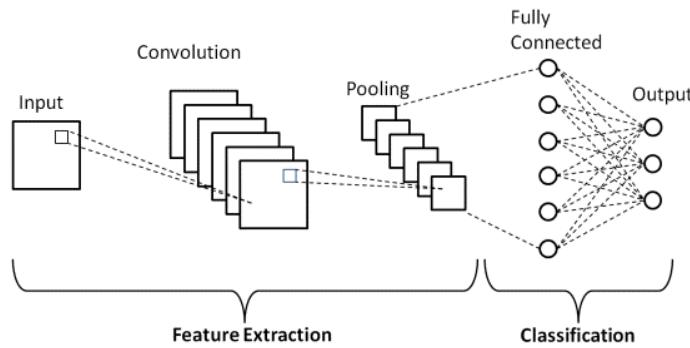
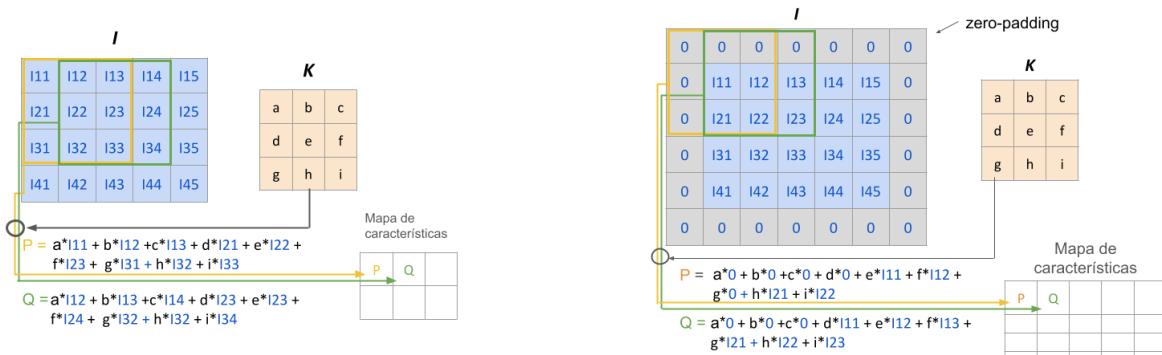


Figura 2.23: Arquitectura de una CNN con capas de convolución, pooling y clasificación. Fuente:[60].

Capa convolucional

La capa convolucional es la capa más importante de una CNN. En ella se extraen las características más significativas de la imagen de entrada, como los bordes, el color o la forma. Para ello se aplica una convolución a la imagen con un filtro. Esta operación matemática se representa como $f(x * w)(t)$ donde x representa la entrada y w el núcleo de convolución [59].

En una CNN, la entrada de la convolución es una matriz multidimensional junto con una matriz de parámetros w que se ajusta durante el aprendizaje (llamada núcleo o filtro). 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 receptivo⁶ [23]. Esta convolución se realiza con un solapamiento total del filtro, lo que resulta en una imagen de menor dimensión (Figura 2.24a). Si se desea mantener la misma dimensión, se puede aplicar zero-padding, que consiste en llenar con ceros la matriz para obtener las dimensiones deseadas (Figura 2.24b). En la figura 2.24 se muestran sendos campos receptivos de la imagen I que contribuyen a las salidas P y Q generadas por el filtro K . La matriz de respuesta al aplicarle el kernel se llama mapa de características (O).



(a) Convolución reduciendo tamaño.

(b) Convolución con zero-padding.

Figura 2.24: Convolución en 2D.

Se observa que el filtro se desplaza por la matriz I con paso unitario en vertical y horizontal.

⁶Región de entrada que contribuye a la salida generada por el filtro

Este parámetro se llama stride y su valor depende de el objetivo que se quiera lograr con esta capa convolucional.

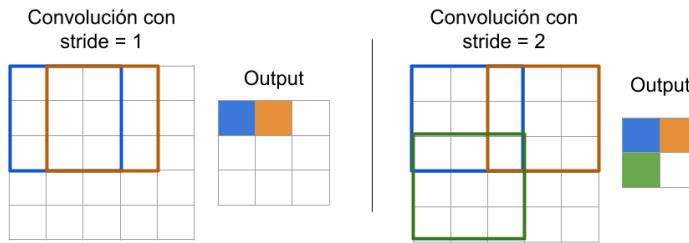


Figura 2.25: Campos receptivos en una convolución. Adaptación de imagen de [74]

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 [74].

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 [23].

Toda la información anterior se resume en la siguiente ecuación [59]:

$$z_{i,j,k} = b_k + \sum_{u=0}^{f_h-1} \sum_{v=0}^{f_w-1} \sum_{k'=0}^{f'_n-1} x_{i',j',k'} \cdot w_{u,v,k',k} \quad \text{con } \begin{cases} i' = i \cdot s_h + u \\ j' = j \cdot s_w + v \end{cases} \quad (2.28)$$

donde:

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

Capa de pooling

El siguiente tipo de capa de las CNN son las pooling, cuyo objetivo es reducir la imagen de entrada para disminuir la carga computacional, el uso de memoria y el número de parámetros,

limitando así el riesgo de sobreajuste y proporcionando robustez contra el ruido y las distorsiones. Esta capa se suele colocar entre las capas de convolución, permitiendo reducir el tamaño de las imágenes mientras se preservan las características más importantes [61]. 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 agregación⁷. Las tres funciones más comunes son el promedio, la suma y el máximo. La Figura 2.26 muestra una capa de max pooling, que es el tipo más común [23].

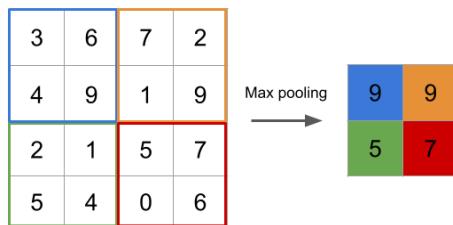


Figura 2.26: Capa de max pooling con un kernel de 2×2 , stride 2 y sin padding.

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 2.27. Esto significa que pequeñas variaciones en la posición de las características dentro de la imagen no afectan significativamente la salida.

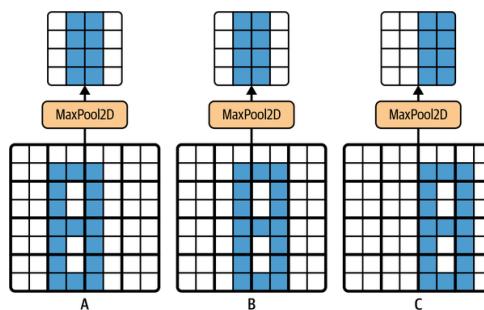


Figura 2.27: Invarianza a translaciones pequeñas mediante una capa de max pooling. Fuente [23]

Una capa típica de convolución se compone de tres etapas. En la primera etapa, se realizan múltiples convoluciones para producir un conjunto de activaciones lineales. En la segunda etapa, cada activación lineal se procesa a través de una función de activación no lineal, como la función de activación rectificada lineal (ReLU). Esta etapa a menudo se denomina etapa de detección. En la tercera etapa, se aplica una función de pooling para modificar la salida de la capa. Este tipo de capa de convolución compuesta será de utilidad en los experimentos realizados en capítulos posteriores, como por ejemplo en la creación de una M-CNN para el problema de clasificación de malware del Capítulo 3 [59].

Capa de Unpooling

En la figura 2.26 se presenta un ejemplo gráfico relacionado con la operación de max pooling. Durante este proceso, es posible mantener un mapa de localizaciones, que señala la posición de origen del valor máximo que ha generado la capa. Este mapa permite ejecutar la operación inversa conocida como reconstrucción (Unpooling), colocando los valores obtenidos en las posiciones

⁷Las funciones de agregación devuelven un valor único de un conjunto de registros.

indicadas por el mapa de localizaciones. Una vez completada la reconstrucción, se puede realizar una interpolación lineal usando el método del vecino más próximo para obtener el mapa de características reconstruido [59]. Este tipo de capa hace el proceso inverso de la capa de pooling. En vez de reducir la dimensión de entrada, la aumenta utilizando o bien interpolación lineal o bilineal o bien rellenando con ceros el resto de casillas [59]. La operación de Unpooling se suele emplear junto con la operación de convolución transpuesta, como se describe más adelante, y se utiliza, por ejemplo, en el modelo de autoencoder utilizado en la sección 3.3, en un autoencoder convolucional.

En Keras, esta capa se denomina `UpSampling2D`.

Convolución Transpuesta

La necesidad de realizar convoluciones transpuestas surge generalmente de la necesidad de aplicar una transformación en la dirección inversa a una convolución directa o normal, como la proyección de mapas de características a un espacio de mayor dimensión. Aunque a veces se denomina deconvolución, esta es en realidad una operación distinta. En las redes neuronales convolucionales (CNN), esta operación es más compleja que en las redes totalmente conectadas, donde solo se requiere el uso de una matriz de pesos transpuesta [59].

Considere la convolución mostrada en la figura 2.24a. Sean I y O la entrada y la salida respectivamente expandidas en forma vectorial de izquierda a derecha y de arriba a abajo. Entonces la operación de convolución 2.9.3 puede expresarse ahora como:

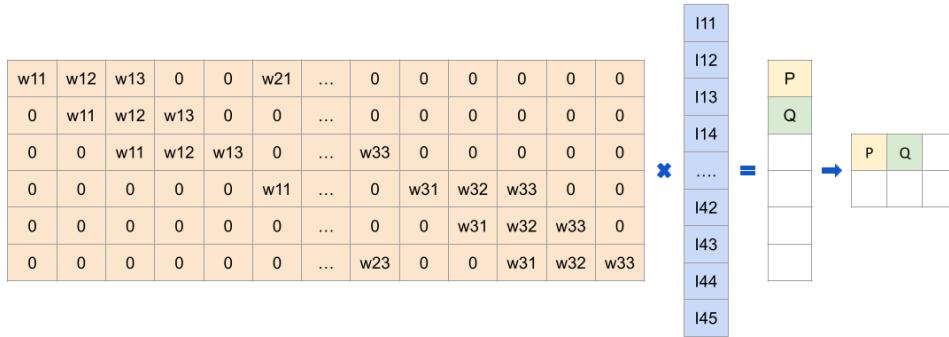


Figura 2.28: Convolución y representación matricial. $C \cdot I = O$

La convolución puede ser expresada como una matriz dispersa C , caracterizada por la presencia de múltiples ceros, donde los elementos no nulos corresponden a los elementos w_{ij} del núcleo K . Aquí, i y j denotan la fila y la columna, respectivamente. Esta operación lineal transforma la matriz de entrada en un vector de 20 dimensiones y produce un vector de 6 dimensiones, el cual se reestructura en una matriz de salida de 2×3 . Usando esta formulación, el paso hacia atrás se obtiene mediante la transposición de C [59].

Con la operación mencionada, se tiene un vector de 6 dimensiones como entrada y se obtiene un vector de 20 dimensiones como salida. En ambos casos, los valores w_{ij} del núcleo definen las matrices C y C^t utilizadas en la propagación hacia delante y hacia atrás. Si se realiza la multiplicación matricial $C^t \cdot O$, cuyas dimensiones son 20×6 y 6×1 , se obtiene la matriz esperada I de dimensión 20×1 , que finalmente habrá que redimensionar [59].

Al igual que en las capas convolucionales originales, también puede añadirse zero-padding a la matriz y con desplazamientos (strides) superiores a la unidad [59].

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

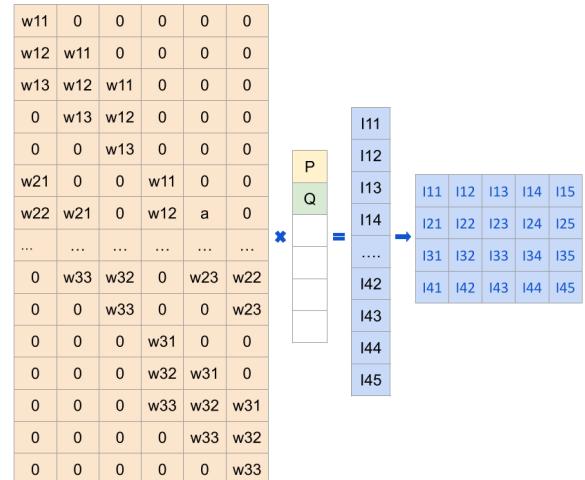


Figura 2.29: Convolución traspuesta. $C^t \cdot O = I$.

Capas Totalmente Conectadas (Fully Connected)

Por último están las capas totalmente conectadas (*fully connected*), que realizan la clasificación sobre la salida generada por las capas de convolución y pooling. Como el input de una capa densa debe ser un vector, primero se debe aplanar la salida de la última capa para poder utilizar después esta capa. Cada una de sus neuronas está conectada a todas las de la capa anterior, estableciendo una red densa de conexiones. Este tipo de neuronas suele ir seguido de una capa Dropout para mejorar la generalización del modelo. 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 [31].

2.9.4. Red Neuronal Recurrente

Restricted Boltzmann Machine

2.10. 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.

2.10.1. 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 [26] 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 [25] 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 [53]. 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 [21], 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 [46]. Esto, junto con su capacidad de auto-diferenciación en modo inverso⁸, 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 [46]. En él se hace un estudio de eficiencia, convergencia, tiempo de entrenamiento y uso de memoria de los diferentes frameworks con varios datasets. Entre sus resultados podemos observar como Keras destaca por encima de las demás en el entorno de la CPU. No solo logra el mejor accuracy en los tres datasets (MNIST, CIFAR-10, CIFAR-100), sino que además también tiene los tiempos de ejecución más bajos y una de las mejores tasas de convergencia. En cuanto al entorno de la GPU, las tres librerías obtienen unos resultados semejantes. En conclusión, podemos afirmar que estos resultados junto con su facilidad de uso, accesibilidad y documentación bien estructurada,

⁸Técnica en la que PyTorch calcula automáticamente las derivadas de las funciones de pérdida con respecto a los parámetros del modelo.

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 [50], 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.”

2.10.2. 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 [62] es una librería de código abierto con herramientas simples y eficientes para el análisis predictivo de datos. Contiene varios algoritmos de aprendizaje automático, desde clasificación y regresión hasta clustering y reducción de dimensionalidad, con la documentación completa sobre cada algoritmo. Está construida sobre otras librerías que veremos más adelante como Numpy, SciPy y matplotlib. Aunque no se aprovecharán todas estas funcionalidades de scikit-learn, si que se va a utilizar una de sus funciones más populares, `train\test\split()` [63]. 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.

```

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

```

Las variables `X_train`, `X_test` y `compañía` son numpy arrays. **NumPy** [15] 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 [10]. 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 [13], 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** [49]. 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 [61] para poder clasificar distintos tipos de ciberataques. Como el objetivo principal de este trabajo es el estudio y puesta en práctica de diferentes algoritmos de aprendizaje automático, se ha decidido tomar como base de datos Microsoft Malware Classification Challenge. La principal razón de esta decisión ha sido que con este dataset tenemos a nuestra disposición dos algoritmos diferentes de machine learning que están referenciados en este review y que se aborda el problema usando cada uno su propio enfoque.

3.1. Microsoft Malware Classification Challenge

El conjunto de datos utilizado en este estudio proviene del Microsoft Malware Classification Challenge (BIG 2015) [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 [61].

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

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

Cada programa malicioso tiene un identificador, un valor hash de 20 caracteres que identifica de forma única el archivo, y una etiqueta de clase, que es un número entero que representa una de las 9 familias de malware al que puede pertenecer. Por ejemplo, el programa *0ACDbR5M3ZhBJajygTuf* tiene como etiqueta el valor 7. Esta información se puede consultar en el archivo “trainLabels.csv”. Cada programa tiene dos archivos, uno asm con el código extraído por la herramienta de desensamblado IDA y otro bytes¹ con la representación hexadecimal del contenido binario del programa pero sin los encabezados ejecutables (para garantizar esterilidad). Para nuestro estudio vamos a utilizar únicamente este ultimo archivo.

	DIRECCIÓN MEMORIA	REPRESENTACIÓN HEXADECIMAL
Nº LÍNEA		BYTE
28232	0046F470	E0 01 EC 10 4C 01 62 00 EC 00 82 11 06 11 84 01
28233	0046F480	A8 11 00 10 EE 00 AE 01 42 10 20 11 C2 00 A0 10
28234	0046F490	CC 10 4A 01 42 00 EE 01 AA 00 44 00 84 10 0C 01
28235	0046F4A0	24 11 A8 10 AC 01 AE 11 0E 01 80 10 6A 11 6A 10
28236	0046F4B0	4E 01 82 01 00 01 AE 01 0E 11 E2 11 0A 10 2A 01
28237	0046F4C0	60 01 C8 00 E8 10 28 01 04 00 82 00 62 10 E4 01
28238	0046F4D0	EA 00 CE 01 A6 01 46 11 0C 00 00 00 ?? ?? ?? ?? ??
28239	0046F4E0	?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??

VALORES DESCONOCIDOS

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 [11].

¹Realmente 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: Ramnit, Lollipop, Kelihos_ver3, Vundo, Simda, Tracur, Kelihos_ver1, Obfuscator y Gatak. Según [32], podemos definirlas como:

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

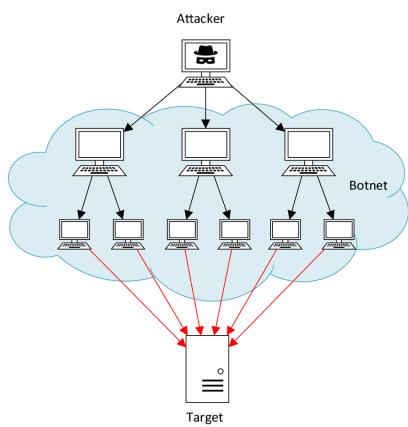


Figura 3.2: Estructura de un botnet. Imagen sacada de [54].

2. **Lollipop** es un tipo de programa adware² 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 backdoor³ que infecta ordenadores descargando y ejecutando archivos arbitrarios que pueden incluir malware adicional. Los ordenadores infectados pasar a ser parte de una botnet, lo que les permite cometer acciones criminales como robo de contraseñas, credenciales bancarias o descargar otros tipos de malware.

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

avanzadas para evitar su detección y eliminación.

5. **Kelihos_ver3** es un troyano tipo backdoor que distribuye correos electrónicos que pueden contener enlaces falsos a instaladores de malware. Consta de tres tipos de bots [37]: 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.

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

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

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

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

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

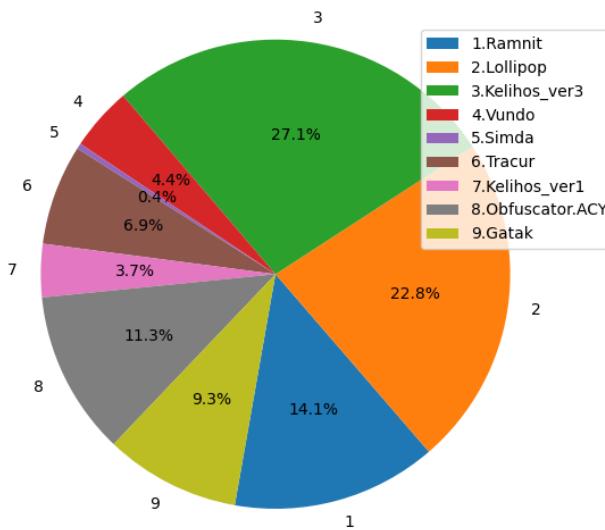


Figura 3.3: Distribución del BIG 2015 training dataset.

Analizando la Figura 3.3, podemos observar como la distribución entre las clases no es uniforme. Mientras que de la clase Simbda hay 42 muestras, de la clase Kelihos_ ver3 hay 2.942, es decir, 70 veces más de muestras. En [36] deciden prescindir de esta clase, pero nosotros hemos decidido hacer el análisis con las 9 clases.

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

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

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

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

Para abordar este problema de clasificación, vamos a realizar dos modelos de aprendizaje automático diferentes para luego comparar sus resultados. El primer método que vamos a utilizar es una 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 2.9.3, 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 [56], pero si lo representamos de forma matricial, estos pequeños cambios pueden ser detectados fácilmente [34].

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 [57], 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 [23], se plantea eliminar estos caracteres y tratar el resto de bytes. Otra solución la proponen Narayanan et al. [55], 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.

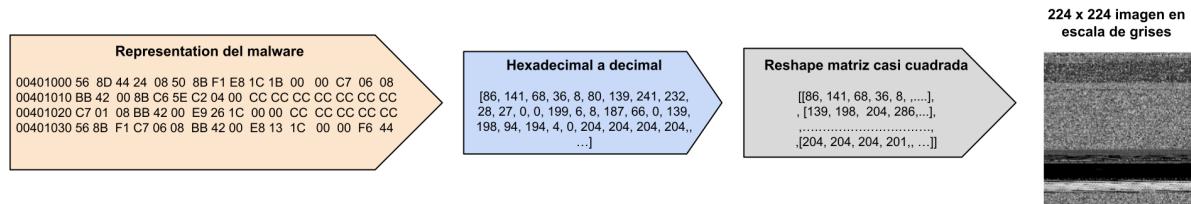


Figura 3.4: Proceso de visualización del malware. Adaptación de [57]

Después de tener todos los bytes agrupados en un vector con sus elementos en formato decimal, hacemos un reshape a una matriz 2D de forma que se consiga una matriz lo más cuadrada posible. Como las dimensiones de cada archivo son diferentes, las dimensiones después de hacer el reshape también lo serán, luego tenemos que fijar un tamaño fijo para poder entrenar nuestra CNN [41]. Para ello, siguiendo el ejemplo de [34], 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 bilineal [29] en base a los resultados obtenidos. En la figura 3.5, se puede observar cuales son los patrones que sigue cada tipo de malware en su forma matricial.

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

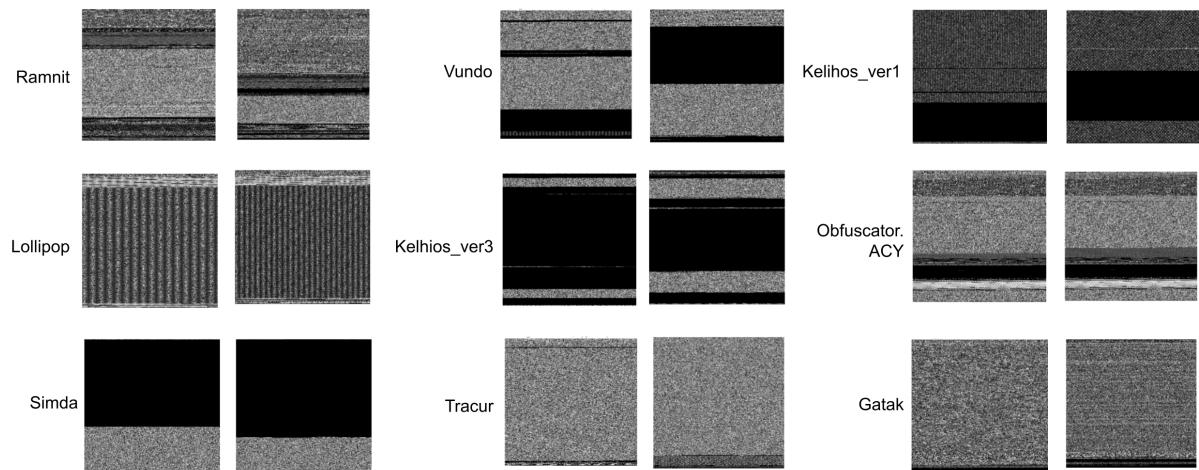


Figura 3.5: Visualización familias malware.

3.2.2. Visualización del modelo

Una vez ya hemos explorado y preparado los datos, el siguiente paso es crear nuestra red neuronal convolucional. Para ello hemos seguido el artículo [34], en el que se crea una M-CNN (malware CNN) con múltiples capas. Vemos su arquitectura en la Figura 3.6.

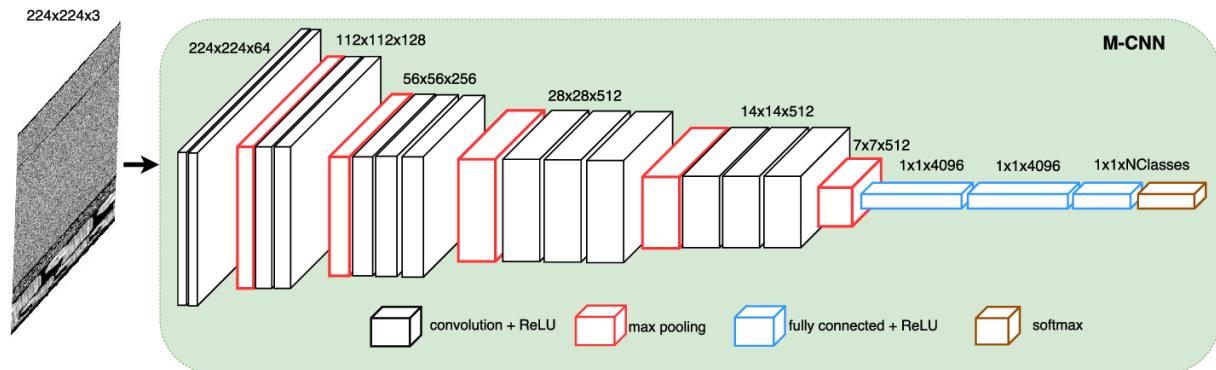


Figura 3.6: Arquitectura de la CNN. Fuente [34].

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 [23]. Después de cada operación de convolución, se aplica la función de activación ReLU para introducir no linealidad en el modelo.

Las capas de max pooling reducen la dimensionalidad de los mapas de características seleccionando el valor máximo en subregiones de 2×2 , lo que no solo reduce la carga computacional sino que también ayuda a hacer las características más robustas a pequeñas variaciones y traslaciones en la imagen de entrada [23].

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, hemos seguido el artículo [34] para la elección de las funciones de pérdida y de optimización, con los parámetros adecuados. 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. La función de pérdida utilizada es `categorical_crossentropy` ya que nuestro problema involucra múltiples clases de etiquetas [14].

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 (Early Stopping), y registrar los resultados de las métricas utilizadas durante el entrenamiento. Además, se baraja el conjunto de datos de entrenamiento antes de cada epoch.

3.2.3. Aportaciones al 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.

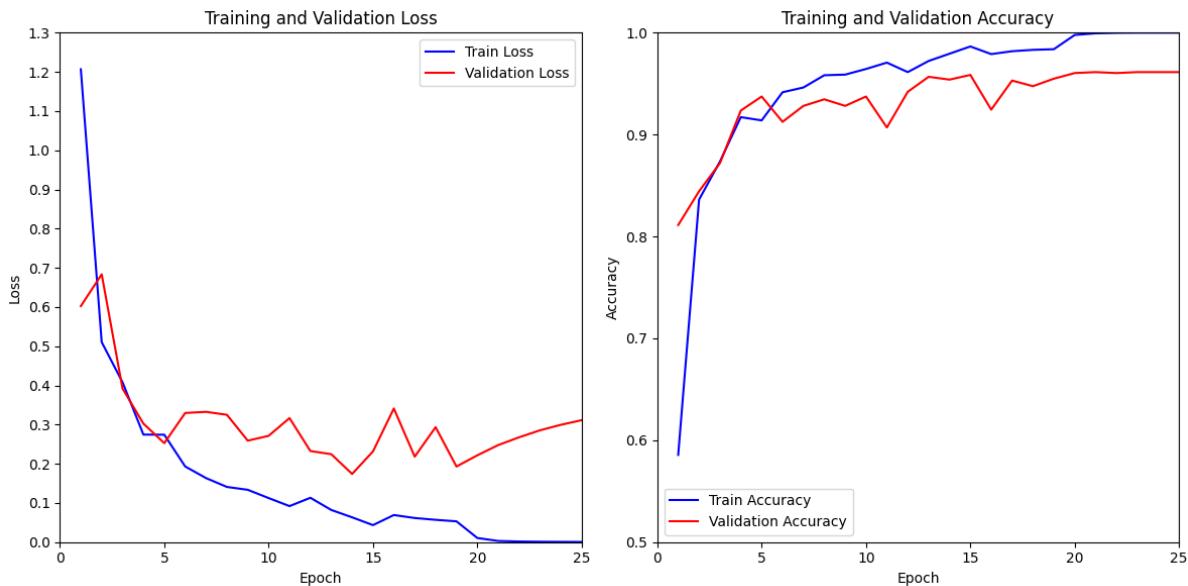


Figura 3.7: Evolución del modelo en términos de accuracy y loss.

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

Debido a la alta cantidad de parametros del deep ae y la alta carga computacional, se decide añadir capas de dropout.

3.3.1. CAE

parametros de optimizador y loss: [27] En AE se decide escoger MSE porque queremos medir regresión, no clasificación

3.3.2. DAE

Para el resto de modelos hemos usado las mismas funciones para poder comparar parámetros

3.3.3. AE

Convolutional autoencoder modelo [72]

3.4. Resultados

Discusión de resultados con recapitulación de todos los resultados discutiendolos, resumiendo un poco todos los resultados obtenidos

[24] 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 multiclassificación, tenemos dos opciones:

- Usamos autoencoder para comprimir la información de entrada y despues esa informacion la usamos para clasificarla usando una DNN [43]
- Usamos una cadena de autoencoders en el cual la salida de h es la entrada del autoencoder $h+1$. Utilizo el articulo [22] donde se desarrolla todo el modelo y explicacion y ademas se hace referencia al artículo [8] 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 [7] 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 artículos [38, 73, 58, 40]. Prácticamente todo el cuerpo del experimento se encuentra en el artículo [38], pero en el artículo [40] 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 [47]. Además, hay buena explicacion del experimento en [70]. Por último, en el articulo [19] 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 [19] 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 [19] 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

Resaltar los experimentos propios que he hecho

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] Andrea Apicella, Francesco Donnarumma, Francesco Isgrò, and Roberto Prevete. A survey on modern trainable activation functions. *Neural Networks*, 138:14–32, 2021.
- [6] Dor Bank, Noam Koenigstein, and Raja Giryes. Autoencoders. *Machine learning for data science handbook: data mining and knowledge discovery handbook*, pages 353–374, 2023.
- [7] 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.
- [8] Yoshua Bengio, Pascal Lamblin, Dan Popovici, and Hugo Larochelle. Greedy layer-wise training of deep networks. *Advances in neural information processing systems*, 19, 2006.
- [9] 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.
- [10] 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.
- [11] 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.
- [12] 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.
- [13] Keras Developers. Model training apis. Consultado el 25-04-2024.
- [14] Keras developers. Probabilistic losses. Consultado el 05-06-2024.
- [15] NumPy Developers. Numpy documentation. Consultado el 25-04-2024.
- [16] 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.

- [17] P Dileep, Dibyayoti 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.
- [18] Oscar R Dolling and Eduardo A Varas. Artificial neural networks for streamflow prediction. *Journal of hydraulic research*, 40(5):547–554, 2002.
- [19] Wisam Elmasry, Akhan Akbulut, and Abdul Halim Zaim. Empirical study on multiclass classification-based network intrusion detection. *Computational Intelligence*, 35(4):919–954, 2019.
- [20] Bradley J Erickson and Felipe Kitamura. Magician’s corner: 9. performance metrics for machine learning models, 2021.
- [21] Facebook AI Research. Pytorch, 2017.
- [22] 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.
- [23] Aurélien Géron. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow*. O'Reilly Media, Inc., 2022.
- [24] 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.
- [25] Google AI Team. Keras, 2015.
- [26] Google Brain Team. Tensorflow, 2015.
- [27] Xifeng Guo, Xinwang Liu, En Zhu, and Jianping Yin. Deep clustering with convolutional autoencoders. In *Neural Information Processing: 24th International Conference, ICONIP 2017, Guangzhou, China, November 14–18, 2017, Proceedings, Part II 24*, pages 373–382. Springer, 2017.
- [28] 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.
- [29] Ke He and Dong-Seong Kim. Malware detection with malware images using deep learning techniques. In *2019 18th IEEE international conference on trust, security and privacy in computing and communications/13th IEEE international conference on big data science and engineering (TrustCom/BigDataSE)*, pages 95–102. IEEE, 2019.
- [30] 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.
- [31] 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.
- [32] 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.

- [33] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.
- [34] 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.
- [35] Kavya. Auto encoder — implementation. Consultado el 25-05-2024.
- [36] 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.
- [37] 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.
- [38] 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.
- [39] Sera Kim and Seok-Pil Lee. A bilstm-transformer and 2d cnn architecture for emotion recognition from speech. *Electronics*, 12(19):4034, 2023.
- [40] 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.
- [41] 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.
- [42] 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.
- [43] 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.
- [44] Mika Luoma-aho. Analysis of modern malware: obfuscation techniques. 2023.
- [45] Andrew L Maas, Awni Y Hannun, Andrew Y Ng, et al. Rectifier nonlinearities improve neural network acoustic models. In *Proc. icml*, volume 30, page 3. Atlanta, GA, 2013.
- [46] 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.
- [47] 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.

- [48] Rosa Martínez Álvarez-Castellanos et al. Análisis de las máquinas sparse autoencoders como extractores de características. 2017.
- [49] matplotlib Developers. Matplotlib: Visualization with python. Consultado el 25-04-2024.
- [50] Matthew Carrigan. Keras Documentation. <https://keras.io/>. Consultado el 06-05-2024.
- [51] Montreal University. Theano, 2010.
- [52] James Moor. The dartmouth college artificial intelligence conference: The next fifty years. *Ai Magazine*, 27(4):87–87, 2006.
- [53] Andreas C Müller and Sarah Guido. *Introduction to machine learning with Python: a guide for data scientists*. .O'Reilly Media, Inc.", 2016.
- [54] Mohammad Najafimehr, Sajjad Zarifzadeh, and Seyedakbar Mostafavi. A hybrid machine learning approach for detecting unprecedented ddos attacks. *The Journal of Supercomputing*, 78, 04 2022.
- [55] 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.
- [56] Lakshmanan Nataraj, Shanmugavadiel Karthikeyan, and BS Manjunath. Sattva: Sparsity inspired classification of malware variants. In *Proceedings of the 3rd ACM Workshop on Information Hiding and Multimedia Security*, pages 135–140, 2015.
- [57] 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.
- [58] 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.
- [59] Gonzalo Pajares Martinsanz et al. Aprendizaje profundo. 2021.
- [60] 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.
- [61] 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.
- [62] scikit-learn Developers. ssikit-llarn documentation. Consultado el 25-04-2024.
- [63] scikit-learn Developers. Train test split de scikit-learn. Consultado el 25-04-2024.
- [64] Nitish Srivastava. Improving neural networks with dropout. *University of Toronto*, 182(566):7, 2013.
- [65] 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.

- [66] Sajedul Talukder. Tools and techniques for malware detection and analysis. *arXiv preprint arXiv:2002.06819*, 2020.
- [67] Mingdong Tang and Quan Qian. Dynamic api call sequence visualisation for malware classification. *IET Information Security*, 13(4):367–377, 2019.
- [68] Yingjie Tian and Yuqi Zhang. A comprehensive survey on regularization strategies in machine learning. *Information Fusion*, 80:146–166, 2022.
- [69] Tokyo University. Chainer, 2015.
- [70] 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.
- [71] 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.
- [72] 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.
- [73] 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.
- [74] 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.
- [75] 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.
- [76] Junhai Zhai, Sufang Zhang, Junfen Chen, and Qiang He. Autoencoder and its various variants. In *2018 IEEE international conference on systems, man, and cybernetics (SMC)*, pages 415–419. IEEE, 2018.
- [77] 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.

