UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE CIENCIAS MATEMÁTICAS

DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

TRABAJO DE FIN DE GRADO

Algoritmos de Aprendizaje Automático aplicados a problemas de Ciberseguridad

Presentado por: Pablo Jiménez Poyatos

Dirigido por: Luis Fernando Llana Diaz

Grado en Matemáticas

Curso académico 2023-24

Agradecimientos

Resumen

Palabras clave:

Abstract

Keywords:

Índice general

Capítulo 1

Introducción

1.1. Motivación y objetivos del trabajo

1.2. Contexto y antecedentes del trabajo

1.2.1. Redes neuronales

1.2.2. Importancia de la detección y prevención de ataques

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

1.2.3. Evolución de las amenazas cibernéticas

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

1.2.4. Avances en el aprendizaje automático para ciberseguridad

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

1.3. Metodología

1.4. Estructura de la memoria

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

Capítulo 2

Fundamentos de las redes neuronales

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

en la pagina 458 de hands aparece la arquitectura mía

2.1. Revisión teórica

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

2.2. Introducción

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

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

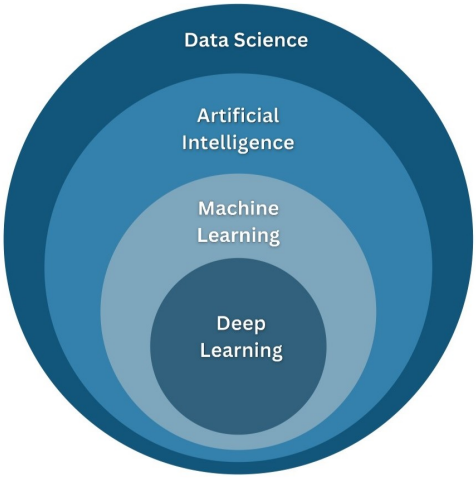


Figura 2.1: Relación entre Ciencia de Datos, Inteligencia Artificial, Machine Learning y Deep Learning.

2.3. Aprendizaje Automático

Por otro lado, el Aprendizaje Automático (ML, por sus siglas en inglés) es la ciencia o el arte de programar ordenadores para que puedan aprender a partir de datos. Arthur Samuel lo definió en 1959 como "el campo de estudio que otorga a las computadoras la capacidad de aprender sin ser explícitamente programadas". Más formalmente, según Tom Mitchell (1997), "se dice que un programa de computadora aprende de la experiencia E con respecto a alguna tarea T y alguna medida de rendimiento P, si su rendimiento en T, medido por P, mejora con la experiencia E" [17]. 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 [17]. 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 [17].

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 [48].

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

El **aprendizaje supervisado**, es un tipo de entrenamiento en el que los datos tienen asociados las salidas deseadas, también llamadas etiquetas. Un ejemplo de aprendizaje supervisado puede ser el del filtro de spam ya que se entrena el modelo con los correos y con la etiqueta de si son spam o no. Otros ejemplos incluyen regresión lineal, regresión logística, árboles de decisión y redes neuronales.

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

En el caso de encontrarnos ante un problema en el que tengamos datos tanto etiquetados como sin etiquetar, nos encontramos antes un tipo de **aprendizaje semisupervisado**, que se encuentra entre el supervisado y el no supervisado. Este tipo de aprendizaje suele darse en situaciones en las que obtener etiquetas de los datos puede ser muy costoso pero sin embargo obtener datos sin etiquetar no tanto. Un ejemplo podría ser la agrupación de fotos donde sale la misma persona en Google Photos. La parte no supervisada sería la de agrupación y la supervisada la de dar una etiqueta a cada grupo.

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

2.4. Aprendizaje profundo

Dentro del Machine Learning, se encuentra el Deep Learning, cuya base son las redes neuronales artificiales (ANN). Una red neuronal artificial es un modelo matemático inspirado en la estructura de una red neuronal biológica. Consiste en una red de neuronas interconectadas organizadas por capas, con una capa de entrada, una o más capas ocultas y una capa de salida [**?**]. En cada neurona se aplica una suma ponderada de las señales recibidas a las que se le aplica una función de activación o conexión no lineal. La capa de entrada recibe la información del exterior y la agrupa en la capa de entrada, mandando una salida a la siguiente capa a través de sus neuronas con pesos asociados en cada conexión. Las capas ocultas reciben información de otras neuronas

artificiales y cuyas señales de entrada y salida permanecen dentro de la red. Por último, la capa de salida recibe la información procesada y la devuelve al exterior con la salida predicha por nuestro modelo. Además de los pesos que se van ajustando durante el entrenamiento, también está el sesgo o bias, que es un valor que se asigna a cada neurona de cada capa para añadir características adicionales a la red neuronal que antes no tenía.

2.4.1. Perceptrón

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

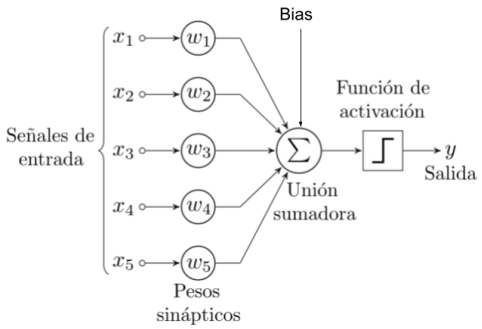


Figura 2.2: Modelo del perceptrón simple

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

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

Los coeficientes w_i, $i = 1, 2, \ldots, n$ llamados pesos, dan un valor determinado a cada una de las entradas en función de la importancia para obtener la salida. Además, el coeficiente b es el sesgo o bias que se añade a la función. Otra forma práctica de escribir esta ecuación sería: $f(x) = \sum_{i=1}^{n+1} w_i \cdot z_i = w^t \cdot z dondew = (w_1, \ldots, w_n, b)^t y z^t = (x_1, \ldots, x_n, 1)^t$

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

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

Veamos ahora el funcionamiento del aprendizaje del perceptrón.

Network Architecture and Learning

ANN architectures vary significantly, differing in network topology, learning processes, and training strategies. The complexity of the model influences the choice of architecture. Single-layer networks can represent linear models, while multi-layer networks are used for non-linear models. This work focuses on three-layer networks, which are effective for both linear and non-linear outputs.

Model Types

Different types of ANN models include:

- **Feedforward Networks (BP1)**: Use backpropagation with momentum and hyperbolic activation functions in hidden layers and linear functions in output layers.
- **BP2 Networks**: Similar to BP1 but with sigmoidal activation functions in both hidden and output layers.
- **Stochastic Networks (SM)**: Same as BP2 but employ stochastic learning processes.
- **Elman Recurrent Networks (ELM)**: Utilize recurrent connections and backpropagation with momentum.
- **Radial Basis Function Networks (RBN)**: Differ in their approach to activation functions and training.

Each model type is tailored to specific tasks and performance requirements. A detailed exploration of these models can be found in the works of Dibike et al. (1999), Zell et al. (1995), Mitchell (1997), Hassoun (1995), and Freeman and Skapura (1992).

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.

Dentro del AA se encuadra el conocido como Aprendizaje Profundo (AP). En la literatura especializada a nivel internacional es muy común referirse al AP por sus términos en inglés, esto es, Deep Learning, que es el núcleo central del presente libro. Por otra parte, y al hilo de esta cuestión, conviene reseñar que son muchos y diversos los términos en inglés utilizados para definir y describir los conceptos involucrados bajo el paradigma del AP. Muchos de los cuales no poseen una clara traducción conceptual al español, razón por la cual los considerados bajo esta situación se mantienen a lo largo del libro con el fin de que el lector pueda fácilmente identificarlos en la literatura especializada escrita en inglés. Solo se han traducido aquellos conceptos que no admiten discusión, manteniendo en todo caso su expresión original en inglés.

Bien es cierto que desde los años 50 del siglo pasado, la IA a veces se ha sobrevalorado y se ha considerado como muy prometedora en diversas ocasiones, eso pese a que no se han llegado a

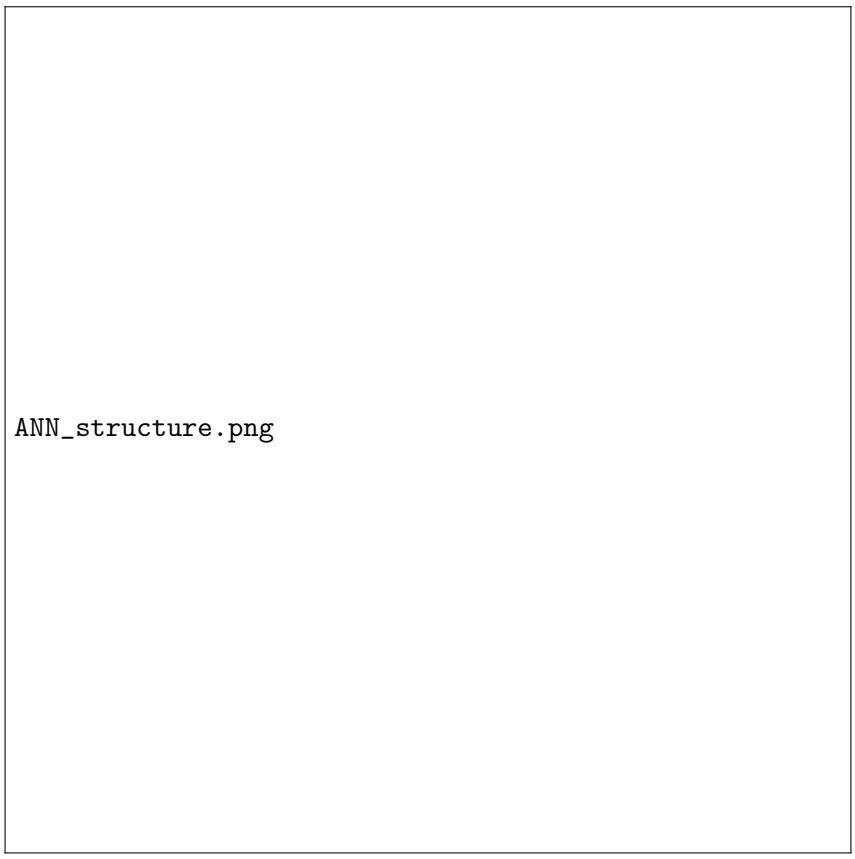


Figura 2.3: Artificial Neural Network fully connected Multi Input Multi Output structure (MIMO) ANN(30,20,7).

alcanzar las perspectivas iniciales. Por otra parte, no es menos cierto que en los últimos años se están viendo avances importantes gracias al AP. Ello a pesar de que todavía es relativamente frágil de cara a su generalización y adaptación a entornos o escenarios cambiantes, principalmente por falta de datos suficientes que capturen los cambios de dicho entorno, pudiendo aparecer ciertos sesgos por falta de la información necesaria extraíble de los datos. Algunos autores como Marcus y Davis (2019), expresan, no sin cierta razón, algunos aspectos relativos a las ventajas e inconvenientes de los procesos de AP, achacándoles que muchas arquitecturas basadas en redes neuronales hacen cosas increíbles, sin ser conscientes por parte de quien las aplica del conocimiento real sobre lo que están haciendo, y por ello, no son sistemas totalmente inteligentes.

Aunque en parte, esto último puede verse desde esta perspectiva, no es menos cierto que los desarrollos basados en AP son capaces de conseguir resultados importantes, siendo esta la perspectiva desde la que se abordan y plantean los temas del presente libro.

Como sostienen Goodfellow y col. (2016), en los comienzos de la IA, se abordaron rápidamente problemas intelectualmente difíciles para los seres humanos, pero relativamente sencillos para las computadoras, todo ello mediante una lista de reglas matemáticas formales. A partir de ahí, el verdadero desafío para la IA se tornó en resolver tareas fáciles de realizar para las personas, pero difíciles de describir formalmente. Aquí se incluyen tareas tales como el reconocimiento de objetos en imágenes, palabras o acciones en los movimientos. No cabe duda de que en este aspecto el AP ha conseguido ya logros muy relevantes. Es en este rasgo, y más concretamente en la exposición de una serie de técnicas orientadas a tal fin, donde se centra el presente libro.

En definitiva, se trata de exponer una serie de técnicas para resolver problemas, por decirlo de alguna manera, intuitivos para el ser humano, con el uso de las computadoras, que aprenden me-

diante los métodos y algoritmos diseñados a partir de los datos suministrados y sin necesidad de que los humanos especifiquen formalmente todo el conocimiento requerido por la computadora. En cualquier caso, y siguiendo también la teoría expuesta en Goodfellow y col. (2016), la jerarquía de conceptos permite que la computadora aprenda conceptos complicados al construirlos a partir de otros más simples, todos ellos estructurados en múltiples capas, razón por la cual a este enfoque se le denomina con el término ya indicado de AP. En cualquier caso, como una técnica específica del AA, estos procedimientos se encaminan a extraer patrones determinados a partir de los datos.

Existe una diferencia fundamental en lo que respecta a la extracción de las mencionadas características entre las técnicas clásicas, por llamarlas de alguna manera, de AA y las específicas del AP. Por ejemplo, considérese un ejemplo sencillo biclase, en el que se trata de separar el cielo y la hierba en una imagen de color de un paisaje de campo. Los datos disponibles en este caso son valores de color de los píxeles, de forma que en el caso del cielo predominan las tonalidades azules, mientras que en la hierba son las verdes. Un método simple tal como naive Bayes puede separar los patrones en dos clases diferentes teniendo en cuenta que los mismos están definidos, por lo que se conoce como características. La extracción de características en este caso es esencial. Por otro lado, y siguiendo en el ámbito de las imágenes, estas se caracterizan por poseer información espacial, y en vídeos, también temporal. Las redes neuronales profundas pueden captar perfectamente ambos tipos de información. En el primer caso, los filtros de convolución son responsables de la captura espacial, a diferencia de lo que ocurre con otros modelos de red, tal como las de retropropagación, en las que las características de las imágenes se transforman en vectores que se suministran a la entrada, perdiendo las relaciones espaciales. Por ejemplo, una imagen de dimensión 3x3 se transforma en un vector con 9 componentes. En el caso de las características temporales las redes recurrentes tienen la habilidad de realizar tal captura.

No obstante, para muchas tareas no resulta fácil extraer características. Por ejemplo, supóngase que a partir de una imagen se quieren identificar peatones cuando un vehículo autónomo navega en un entorno urbano. Una persona puede identificarse por poseer cabeza, tronco y extremidades. Se podría pensar en detectar la presencia de extremidades o del cuerpo y la cabeza o todas, lo cual no resulta trivial debido a que no es fácil establecer las características de dichas partes. Los brazos y las piernas son alargados, el tronco tiene una forma más rectangular, pero en todos los casos nunca están exentos de elementos perturbadores, tales como el uso de distintos tipos de ropa, sombras, oclusiones totales o parciales, entre otros.

Una solución a este problema consiste en utilizar el AA para descubrir no solo la proyección de la representación a la salida, sino también la representación misma. Este enfoque se conoce como aprendizaje de representación según Goodfellow y col. (2016). Las representaciones aprendidas a menudo resultan en un rendimiento mucho mejor que el que se puede obtener con representaciones diseñadas a mano. También permiten que los sistemas de IA se adapten rápidamente a nuevas tareas, con una mínima intervención humana. Un algoritmo de aprendizaje de representación puede descubrir un buen conjunto de características para una tarea simple en minutos o para una tarea compleja en horas y meses. El diseño manual de características para una tarea compleja requiere una gran cantidad de tiempo y esfuerzo humanos, pudiendo llevar incluso décadas. Un ejemplo por excelencia de un algoritmo de aprendizaje de representación es el *autoencoder* (autocodificador), que convierte los datos de entrada en una representación diferente para luego poder devolverla a la representación original mediante el correspondiente *decoder* (decodificador).

Cuando se diseñan algoritmos para aprender las características, el objetivo consiste en separar los factores de variación que expliquen los datos observados. El concepto factor se refiere a abstracciones que ayudan a distinguir entre la alta variabilidad de los datos observados; así,

en el reconocimiento de los peatones los factores de variación hacen referencia a la posición de las extremidades con respecto al tronco, la posición con respecto a la cámara, la ropa con la que van vestidos, las oclusiones de las extremidades, las posibles sombras proyectadas sobre sus cuerpos o la intensidad de la luz con la que se ha obtenido la imagen, entre otros. La mayoría de las aplicaciones exigen separar los factores de variación descartando aquellos que no interesan. A la vista de lo cual, resulta francamente difícil obtener una representación tal que permita resolver el problema. Es precisamente aquí donde entra en acción el aprendizaje profundo, ya que permite introducir representaciones que se expresan en términos de otras representaciones a distintos niveles, que van estructurando convenientemente la información. Por ejemplo, la figura 2.4 muestra un sistema basado en AP, concretamente una Red Neuronal Convolucional (RNC) o en terminología inglesa *Convolutional Neural Network* (CNN), donde se representa el concepto de una imagen de una taza combinando conceptos más simples, tales como bordes, contornos o partes de los objetos hasta llegar a su clasificación, en este caso como taza. La idea de representación en múltiples capas es lo que determina una de las perspectivas del AP. De esta forma, se puede decir con carácter general, que las primeras capas de las redes profundas extraen características de bajo nivel, de modo que estas se van tornando en más complejas con características de mayor nivel hasta llegar a las capas superiores, en las que las características extraídas de la imagen son del más alto nivel. Esta información concatenada permite identificar un objeto (taza), a pesar de que pueda presentar diferentes características tales como, por ejemplo, color, forma o tamaño, lo que permite claramente diferenciar el AP del AA.

Figura 2.4: Modelo de aprendizaje profundo

Otra idea para determinar el concepto de profundidad es la también establecida por Goodfellow y col. (2016), en el sentido de que la profundidad se determina como el estado del computador para aprender un programa computacional multipaso, de forma que cada capa de la representación puede verse como el estado de la memoria del computador después de ejecutar otro conjunto de instrucciones en paralelo. Las redes con mayor profundidad pueden ejecutar más instrucciones en secuencia. Las instrucciones secuenciales ofrecen un gran poder porque las instrucciones posteriores pueden referirse a los resultados de instrucciones anteriores. Según esta visión del AP, no toda la información en las activaciones de una capa codifica necesariamente factores de variación que explican la entrada. La representación también almacena información de estado que ayuda a ejecutar un programa que puede dar sentido a la entrada. Esta información de estado podría ser análoga a un contador o puntero en un programa de computación tradicional. No tiene nada que ver con el contenido de la entrada específicamente, pero ayuda al modelo a organizar su procesamiento.

Existen dos formas principales de medir la profundidad de un modelo. La primera se basa en el número de instrucciones secuenciales que deben ejecutarse para evaluar la arquitectura. Se puede pensar en esto como la longitud de la ruta más larga a través del diagrama de flujo que describe cómo calcular cada una de las salidas del modelo dadas sus entradas. Otro enfoque, utilizado por modelos probabilísticos profundos, considera que la profundidad de un modelo no es la profundidad del gráfico computacional sino la profundidad del gráfico que describe cómo se relacionan los conceptos entre sí. En este caso, la profundidad del diagrama de flujo de los cálculos necesarios para computar la representación de cada concepto puede ser mucho más profunda que la gráfica de los conceptos en sí mismos. Esto se debe a que la comprensión del sistema de los conceptos más simples puede refinarse dando información sobre los conceptos más complejos. Por ejemplo, siguiendo también a Goodfellow y col. (2016), un sistema inteligente que observa una imagen de una cara con un ojo en la sombra puede ver inicialmente únicamente un ojo. Después de detectar la presencia de una cara, el sistema puede inferir que probablemente también esté presente un segundo ojo. En este caso, la gráfica de conceptos incluye sólo dos capas, una capa para ojos y una capa para caras, pero la gráfica de cálculos incluye dos capas si

se refina la estimación de cada concepto dadas las otras n veces. Debido a que no siempre está claro cuál de estos dos modelos (la profundidad del gráfico computacional o la profundidad del gráfico de modelado probabilístico) es más relevante, y debido a que distintas personas eligen diferentes conjuntos de elementos más pequeños a partir de los cuales construir sus gráficos, no existe un único valor correcto para la profundidad de una arquitectura, así como tampoco hay un valor correcto único para la duración de un programa computacional. Tampoco existe un consenso acerca de la profundidad que un modelo requiere para calificarlo como "profundo". Sin embargo, el aprendizaje profundo puede considerarse, con seguridad, como el estudio de modelos que implican una mayor cantidad de composición de funciones aprendidas o conceptos aprendidos que el aprendizaje automático tradicional.

En definitiva, el aprendizaje profundo es un tipo particular de aprendizaje automático que consigue un gran poder y flexibilidad al representar al mundo como una jerarquía anidada de conceptos, donde cada concepto se define en relación con conceptos más simples y representaciones más abstractas calculadas en términos de conceptos menos abstractos.

2.5. Funciones de Unidades Lineales de Activación No Lineales

Una función de activación clásica es la función sigmoide sigmoidal definida como sigue:

$$f(a, x, c) = \frac{1}{1 + e^{-a(x-c)}} \tag{2.3}$$

Dependiendo del signo del parámetro a, la función sigmoide se abre hacia la izquierda o hacia la derecha, siendo apropiada para representar conceptos tales como "muy grande.º "muy negativo". La figura 2-4 muestra la representación de sendas funciones sigmoide: en (a) con los siguientes parámetros $a = 2$ y $c = 4$; y en (b) con $a = -2$ y $c = 4$.

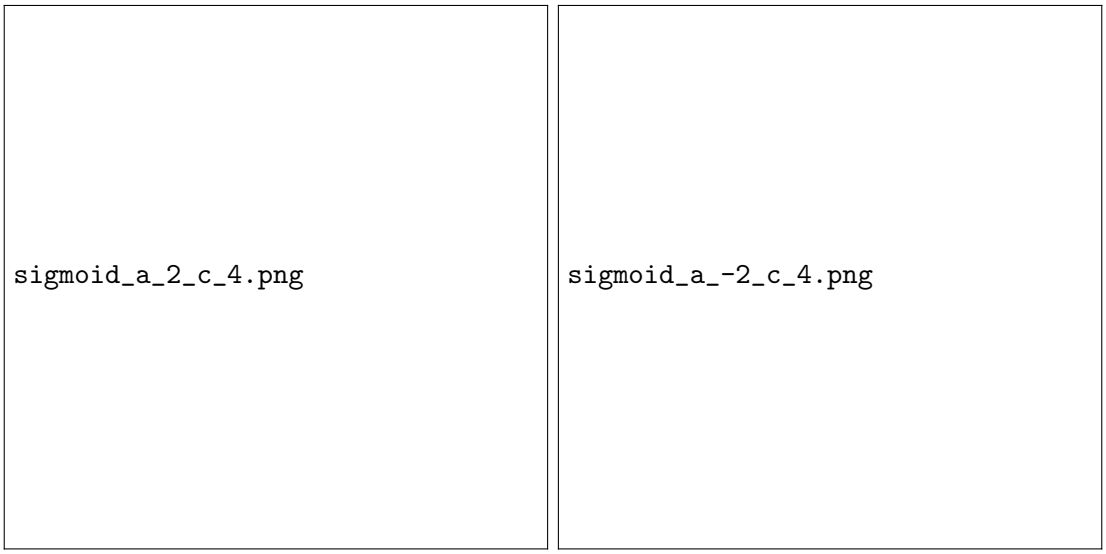


Figura 2.5: Funciones sigmoide: (a) con $a = 2$ y $c = 4$; (b) con $a = -2$ y $c = 4$.

La función sigmoide que proyecta salidas de números reales de entrada al intervalo [0, 1] posee dos problemas:

1. Saturación del gradiente. Cuando el valor de la función de activación se aproxima a los

extremos 0 o 1, el gradiente de la función tiende a 0, lo que repercute en el ajuste de los pesos de las redes.

2. Pesos positivos de forma continua. El valor medio de la función de salida no es 0, lo que origina que los pesos tiendan a ser positivos.

Estas dos cuestiones provocan una convergencia lenta de los parámetros afectando a la eficiencia del entrenamiento.

2.6. Capítulo 2: Computación Numérica

En Courbariaux y col. (2015) se define lo que denominan función sigmoide dura (hard-sigmoid) como sigue:

$$f(x) = \text{máx}(0, \text{mín}(1, 0{,}5(x+1))) \tag{2.4}$$

La función tangente hiperbólica tanh proporciona salidas reales en el rango definido, siendo una variante de la función sigmoide, definida exactamente como

$$\tanh(x) = 2 \cdot \text{sigmoid}(2x) - 1 \tag{2.5}$$

presentando el mismo problema de la saturación del gradiente. La figura 2-5 muestra la representación de la función tanh.

Figura 2.6: Función tanh

La función Unidad Lineal Rectificada (ReLU, Rectified Linear Unit),

$$f(x) = \text{máx}(0, x) \tag{2.6}$$

representada en la figura 2-6(a) tiene las siguientes características:

1. Gradiente no saturado. Por el hecho de que $x > 0$, el problema de la dispersión del gradiente en el proceso de propagación inversa se ve aliviado, y los parámetros en la primera capa de la red neuronal pueden actualizarse rápidamente. En $x = 0$ no es derivable, por lo que es habitual asignar un valor arbitrario en este caso, por ejemplo 0, 0.5 o bien 1.0.

2. Baja complejidad computacional. Dada su propia definición.

No obstante, posee la desventaja de que la neurona ReLU puede morir cuando recibe un gradiente negativo alto durante la retropropagación que le permite aprender más porque su derivada es cero cuando su entrada es menor que cero, por lo que el gradiente será finalmente cero. Esto se puede evitar al inicializar cuidadosamente los pesos o utilizar ReLU con "fugas", similar a ReLU, pero donde su salida es lineal multiplicada por un valor pequeño (aproximadamente 0.001) cuando la entrada es negativa, esto es

$$f(x) = \text{máx}(0{,}01x, x) \tag{2.7}$$

tal y como se muestra en la figura 2-6(b), conocida en ocasiones como ReLU con fugas (LReLU, Leaky ReLU).

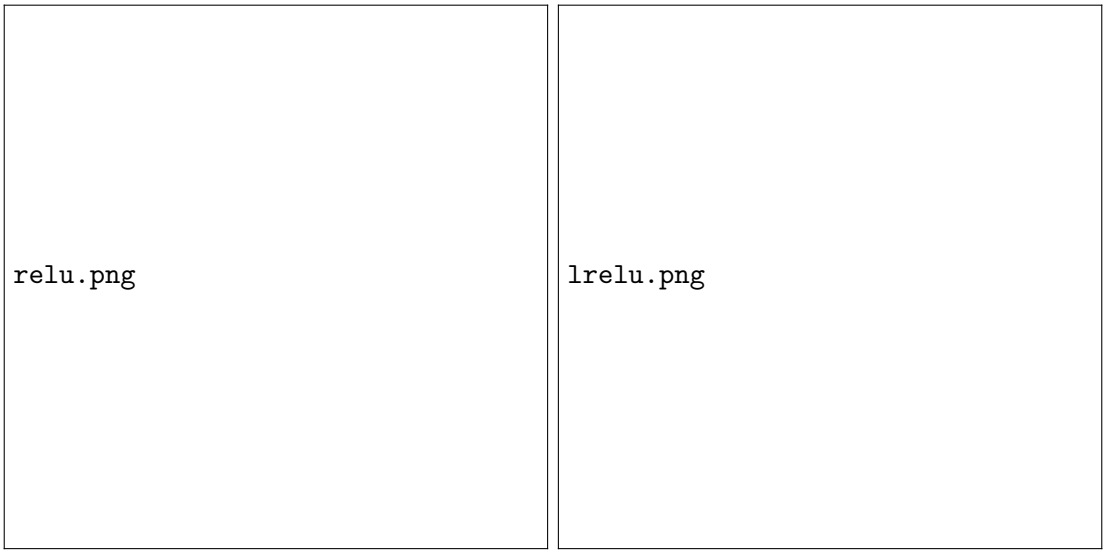


Figura 2.7: Funciones: (a) ReLU; (b) LReLU

En algunos tipos de redes como Mobile Net, que se estudiarán posteriormente, se define una variante de ReLU como sigue (es la función ReLU6), y cuya representación se muestra en la figura 2-7(a). A partir de ella se define hard-swish o h-swish (Hs) representada en la figura 2-7(b), aunque a veces en esta función se utiliza:

$$\text{ReLU6}(x) = \text{mín}(\text{máx}(x, 0), 6) \tag{2.8}$$

$$\text{HS}(x) = x \cdot \frac{\text{ReLU6}(x + 3)}{6} \tag{2.9}$$

La función Paramétrica ReLU (PReLU, Parametric ReLU, He y col., 2015b) se define según:

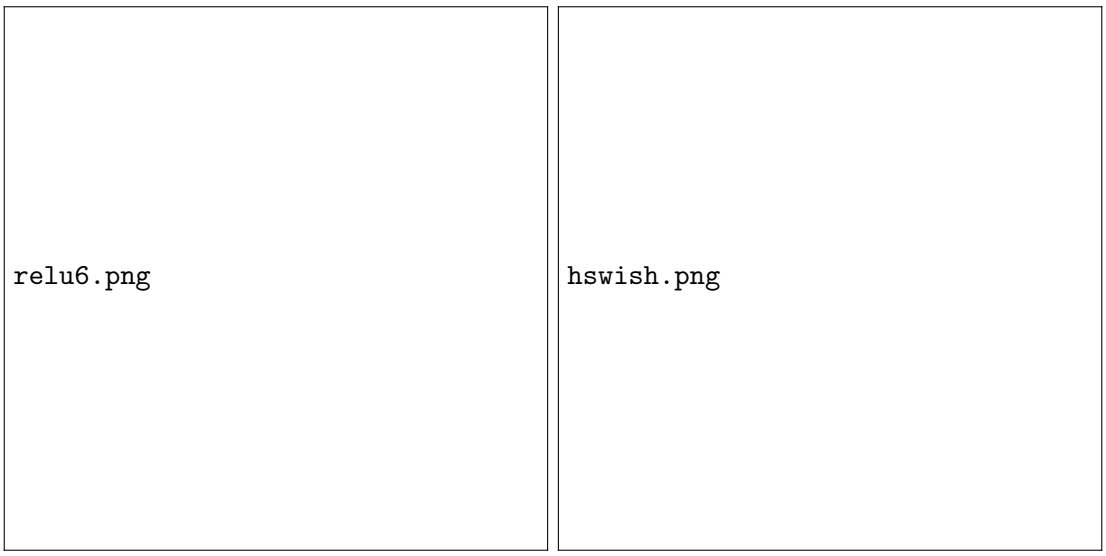


Figura 2.8: Funciones: (a) ReLU6; (b) Hard-Swish

$$f(x,\alpha) = \begin{cases} x & \text{si } x > 0 \\ \alpha x & \text{si } x \leq 0 \end{cases} \tag{2.10}$$

De forma que si el parámetro $\alpha = 0$ la función es exactamente ReLU; si $\alpha > 0$ se trata de la función LReLU, es cuando el parámetro α se incluye como un parámetro a aprender durante el proceso de entrenamiento, cuando la función toma su verdadero significado, de ahí su nombre.

Por otra parte, existe la Unidad lineal exponencial (ELU, Exponential Linear Unit) definida en Clevert y col. (2016) como sigue con $\alpha > 0$:

$$f(x,\alpha) = \begin{cases} x & \text{si } x > 0 \\ \alpha(e^x - 1) & \text{si } x \leq 0 \end{cases} \tag{2.11}$$

El parámetro α controla el valor para el cual se produce la saturación para valores de x negativos. En la figura 2-8 se muestran sendas funciones ELU con valores de $\alpha = 0{,}1$ en (a) y 1.0 en (b), respectivamente.

La función SELU hace referencia a Unidad lineal exponencial escalada (Scaled Exponential Linear Unit), siendo una versión ligeramente modificada de ELU por Klambauer y col. (2017) y definida como sigue:

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

En la figura 2-9 se muestra la representación gráfica de la función SELU con valores $\lambda = 1{,}0$ y $\alpha = 2{,}0$.

Las funciones ReLU tienen la ventaja de acelerar el entrenamiento, ya que el cálculo del gradiente es simple (0 o 1 dependiendo del signo de la entrada), y no existe una constante en la parte positiva del dominio.

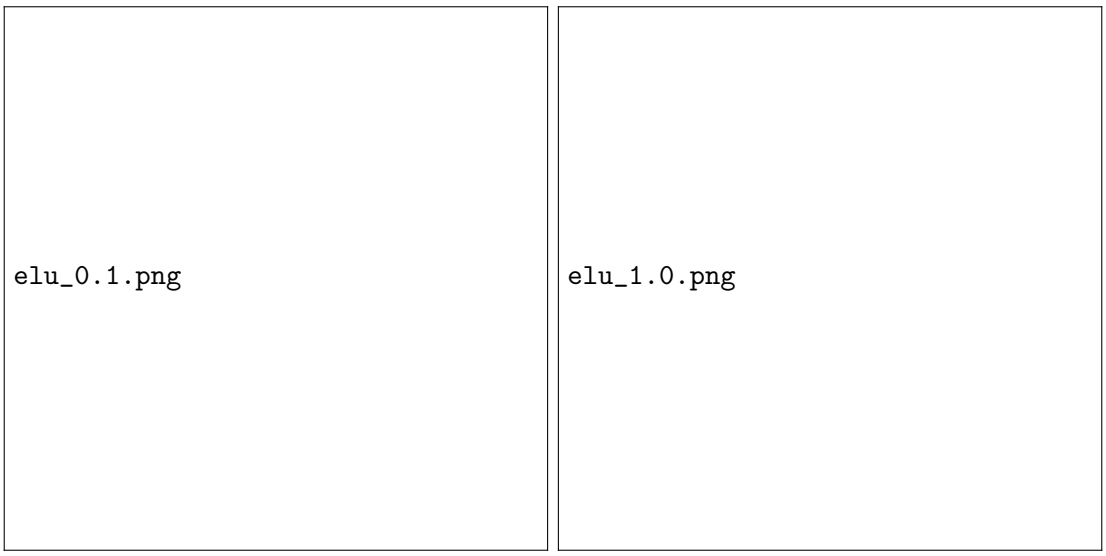


Figura 2.9: Funciones ELU: (a) con $\alpha = 0{,}1$; (b) con $\alpha = 1{,}0$

Figura 2.10: Función SELU

2.7. Introducción

En este capítulo se inicia la presentación, junto con las características más relevantes de las redes neuronales profundas, que, desde sus primeros desarrollos a mediados del siglo XX, han recibido diferentes nombres entre los que destacan: redes neuronales, computadores neuronales, sistemas distribuidos paralelos, modelos conexionistas, entre otros.

Se inicia el capítulo con los fundamentos de este tipo de redes, para abordar a continuación el modelo del perceptrón, como unidad básica, y posteriormente la red de retropropagación. Se finaliza con una pincelada de los modelos conocidos como redes de creencia o bayesianas. En todos los casos con la vista puesta en el concepto de profundidad dentro del aprendizaje profundo.

2.8. Fundamentos Generales

El interés por las redes neuronales data de los años 40, a partir del trabajo de McCulloch y Pitts (1943), donde se proponen modelos de neuronas en la forma de dispositivos binarios basados en un umbral y algoritmos estocásticos que implicaban cambios binarios 0-1 y 1-0 en los estados de las neuronas como la base para el desarrollo de sistemas neuronales. El trabajo posterior de Hebb (1949) estaba basado en modelos matemáticos que intentaban capturar el concepto de aprendizaje por refuerzo o asociación. Durante la primera mitad de los años 50 y principios de los 60, las denominadas máquinas de aprendizaje propuestas por Rosenblatt (1962) supusieron una revolución entre los investigadores en la teoría de reconocimiento de patrones. La razón del gran interés de dichas máquinas llamadas perceptrones fue el desarrollo de las correspondientes demostraciones matemáticas llegando a la conclusión de que los perceptrones, cuando son entrenados con conjuntos de entrenamientos linealmente separables, convergen a una solución en un número finito de iteraciones. La solución tomó la forma de coeficientes de hiperplanos capaces de separar correctamente las clases representadas por patrones del conjunto de entrenamiento.

Desafortunadamente, las expectativas que siguieron a este descubrimiento perdieron fuerza porque el modelo anterior era inapropiado para muchas tareas de reconocimiento de patrones. Intentos posteriores para extender la potencia del perceptrón considerando múltiples capas de perceptrones fracasaron también. Un estado del arte sobre las máquinas de aprendizaje a mitad de los años 60 fue recopilado por Nilsson (1965). Unos pocos años más tarde, Minsky y Papert (1969) presentaron un estudio desalentador sobre las limitaciones de las máquinas de aprendizaje basadas en el perceptrón. Esta idea negativa se mantuvo hasta mitad de la década de los 80, llegándose incluso a rechazar el uso del perceptrón en algunos trabajos como el presentado por Simon (1986).

Los nuevos algoritmos de aprendizaje para perceptrones multicapa presentados por Rumelhart et al. (1986), conocidos como regla delta generalizada para aprendizaje por retropropagación, modificaron el interés por las redes neuronales. En los años sucesivos, nuevos algoritmos fueron presentados por diferentes autores. Aunque no se ha probado la convergencia de dichos algoritmos hacia una solución óptima, han sido aplicados con éxito a muchos problemas de interés práctico, lo que les otorgó una cierta validez en los años 90. A continuación, estas redes cayeron en popularidad en favor de otras técnicas diferentes. Los motivos fueron múltiples. Por ejemplo, no había un lenguaje estándar de facto para modelar redes neuronales. Tampoco se observaban mejoras significativas cuando se añadían muchas capas a la red, resultando a veces contraproducente. Además, el entrenamiento era computacionalmente costoso con los ordenadores disponibles en la época de referencia. Y en muchas condiciones no había suficientes datos para evitar el sobreajuste del modelo.

Para entender correctamente los fundamentos de las redes neuronales profundas conviene empezar estudiando el modelo más básico. Este consiste en especificar un conjunto de funciones indicador $A(x, w)$ que toman los valores {0,1}, considerando que son los parámetros que definen cada posible elemento que hay que clasificar. Además, se considera que se dispone de n muestras (x_i, y_i) de entrenamiento (con $i = 1, ..., n$) de las que se conoce su clase y_i. A continuación se minimiza el riesgo empírico de la función indicador sobre todo elemento del conjunto de entrenamiento. Para ello se suele minimizar el error de clasificación incorrecta $R_o(w) = \sum_i [f(x_i, w) - y_i]^2$.

Para entender mejor el proceso, considérese primero el siguiente caso especial de funciones indicador, donde $h()$ es una función que indica si el elemento pertenece a la clase (toma el valor 1) o no (toma el valor 0) en base a la combinación lineal, sopesada por los valores de los pesos del

modelo w, de los q parámetros de entrada en x.

$$f(x,w) = h\left(\sum_{j=1}^{q} w_j x_j\right)$$

En este caso, si se supone que el conjunto de datos de entrenamiento es linealmente separable, existe un simple procedimiento de optimización para encontrar $f(x,w^*)$, que es conocido como el algoritmo del perceptrón. Cuando los datos son no separables este algoritmo no proporciona una solución óptima, motivo por el que se han desarrollado otros procedimientos. Uno de los métodos considerados es el entrenamiento conocido como Widrow-Hoff o regla delta de mínimos cuadrados, que minimiza una función del tipo dado en la ecuación anterior. Una alternativa diferente consiste en utilizar un perceptrón multicapa (MLP, Multi-Layer Perceptron), que es capaz de manejar adecuadamente tanto las clases separables como las no separables y que utiliza el riesgo empírico funcional dado por:

$$R_{emp}(w,v) = \sum_{i=1}^{n} \left(f(x_i,w,v) - y_i\right)^2$$

que debe ser minimizado con respecto a los parámetros o pesos w y $v = [v_1, v_2, ..., v_m]$. En el caso del perceptrón multicapa, la función $f(x,w,v)$ se parametriza como:

$$f(x,w,v) = h\left(g(x,w,v)\right)$$

donde $g(x,w,v) = \sum_{j=1}^{m} v_j h\left(\sum_{i=1}^{q} w_{ji} x_i\right)$.

2.9. El Perceptrón

El modelo más básico que se puede utilizar en los modelos de redes neuronales recibe el nombre de perceptrón. Debido a su expresión matemática, introducida brevemente en la sección anterior, es capaz de aprender el modelo (obtener los pesos en w) que permite distinguir los elementos de dos clases linealmente separables. En esta sección se detallan los elementos y formas de representar este modelo, y se explican diferentes métodos de entrenamiento para ajustar los valores de sus parámetros.

2.9.1. Arquitectura del Perceptrón

En su forma más básica, el perceptrón aprende una función discriminante lineal $f_d(x)$ que se corresponde con la función $f(x,w^*)$ introducida en la sección anterior. Esta función establece una dicotomía entre dos conjuntos de entrenamiento linealmente separables. Dados dos conjuntos de puntos, se dice que son linealmente separables si existe un hiperplano en el espacio patrón que separa ambos conjuntos de datos. En el caso de un espacio de dos dimensiones, el hiperplano se puede representar gráficamente y a efectos pedagógicos como una recta, tal y como se muestra en la Figura 3-1.

La Figura 3-2(a) muestra el esquema del modelo del perceptrón para dos clases. La respuesta de este dispositivo está basada en dos etapas. En la primera, se calcula la suma promediada de sus entradas mediante una función de decisión lineal con respecto a las componentes de los vectores patrón:

$$f_d(x) = \sum_{i=1}^{n} w_i x_i + w_{n+1}$$

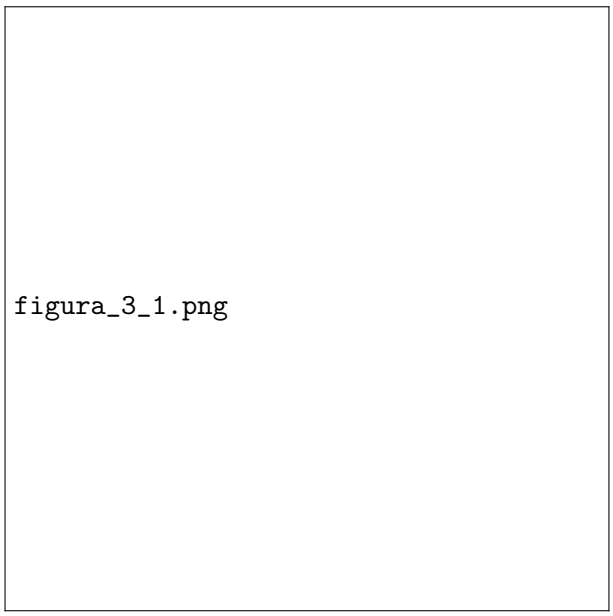
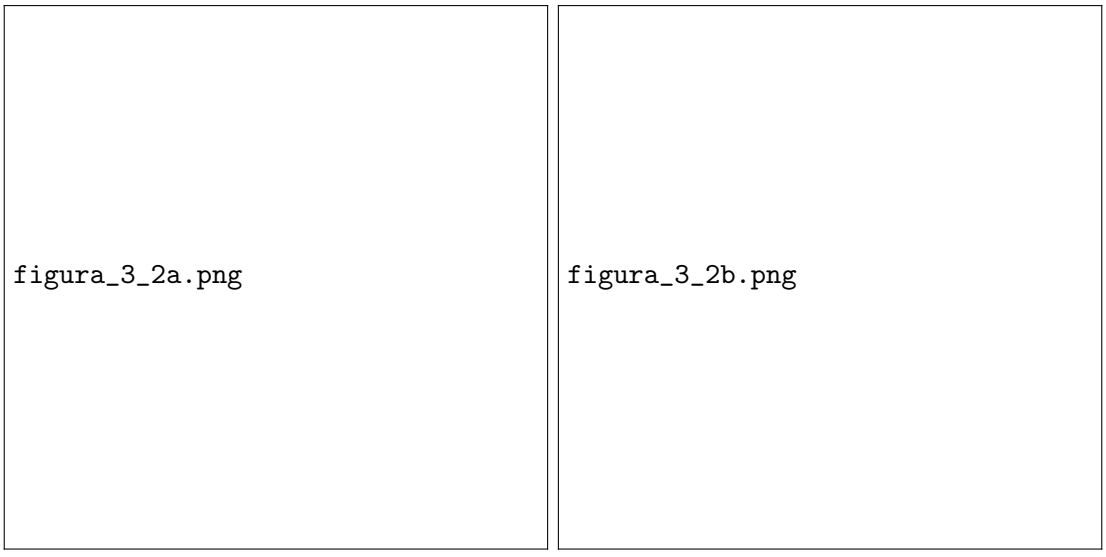


Figura 2.11: Hiperplano de separación para el caso bidimensional

Los coeficientes w_i, $i = 1, 2, ..., n, n+1$, llamados pesos, modifican las entradas antes de que sean sumadas y suministradas al elemento de umbral. Una de las entradas es el sesgo (bias) externo w_{n+1}. En este sentido, los pesos son similares a las sinapsis en el sistema neuronal humano.

En la segunda, se transforma la salida de la primera etapa mediante una función de activación (también conocida como función de transferencia). En su forma más simple, la función de activación es una función escalón (Figura 3-2(b)). Sin embargo, en otras arquitecturas de red también puede ser una función sigmoide o sigmoide binaria.



(a) (b)

Figura 2.12: Esquema del modelo del perceptrón para dos clases

A continuación, se presenta un conjunto de ejemplos de patrones de entrada con el objetivo de encontrar el hiperplano que permite separar dos clases de forma adecuada. El conjunto de patrones de entrada es denotado como $x_1, x_2, ..., x_n$, y cada patrón de entrada tiene una dimensión de m. La salida deseada es denotada como $d_1, d_2, ..., d_n$, donde cada d_i pertenece a

{0,1}.

2.10. 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

2.10.1. Autoencoder

Los autoencoders son una clase de redes neuronales artificiales utilizadas en aprendizaje no supervisado para aprender representaciones eficientes de los datos. Su funcionamiento consiste en codificar la entrada en una representación comprimida y significativa, y luego decodificarla de manera que la reconstrucción sea lo más similar posible a la entrada original [34]. La arquitectura básica de un autoencoder consta de tres partes: el encoder, el cuello de bottela y el decoder (Figura 2.13).

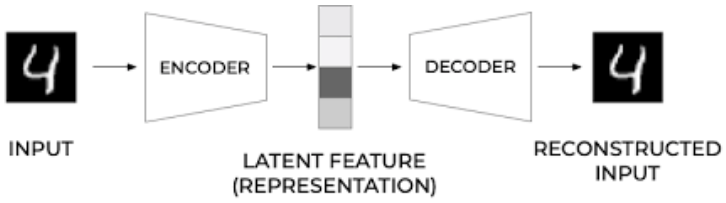


Figura 2.13: Arquitectura de un autoencoder. Fuente:[26].

El encoder mapea los datos de entrada a una representación oculta de menor dimensión utilizando funciones principalmente no lineales, mientras que el decoder reconstruye los datos de entrada a partir de esta representación oculta. Durante el entrenamiento, los parámetros del autoencoder se optimizan para minimizar la diferencia entre la entrada y la salida reconstruida, utilizando una función de pérdida que mide esta discrepancia, como por ejemplo la pérdida de entropía cruzada. Esto concluye el proceso de entrenamiento de un autoencoder.

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

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

y

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

que satisfacen

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

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

Las ecuaciones para obtener la salida de un autoencoder serían:

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

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

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

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

2.10.2. Deep Belief Networks

Red Neuronal Profunda

2.10.3. Red Neuronal Convolucional

Las redes neuronales convolucionales son una de las métodos de machine learning más importantes y utilizados en el campo de la ciberseguridad. Estas redes neuronales están diseñadas para procesar entradas almacenadas en matrices, como las imágenes. Son una parte de las redes profundas que procesa y analiza entradas de imágenes visuales, y están compuestas por neuronas con pesos y sesgos que aprenden a lo largo de su entrenamiento [50]. La arquitectura de una CNN (Figura 2.14) consta de tres tipos de capas: capas de convolución, capas de pooling y la capa de clasificación.

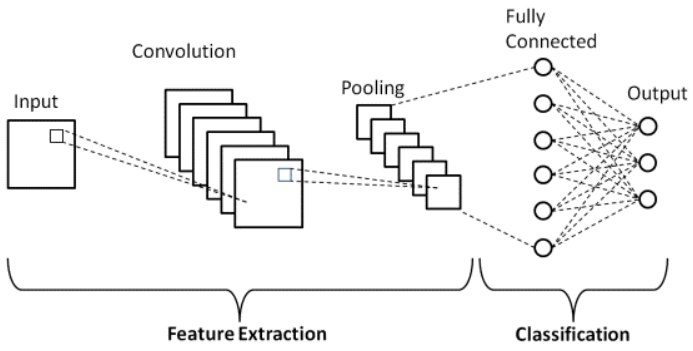


Figura 2.14: Arquitectura de una CNN con capas de convolución, pooling y clasificación. Fuente:[49].

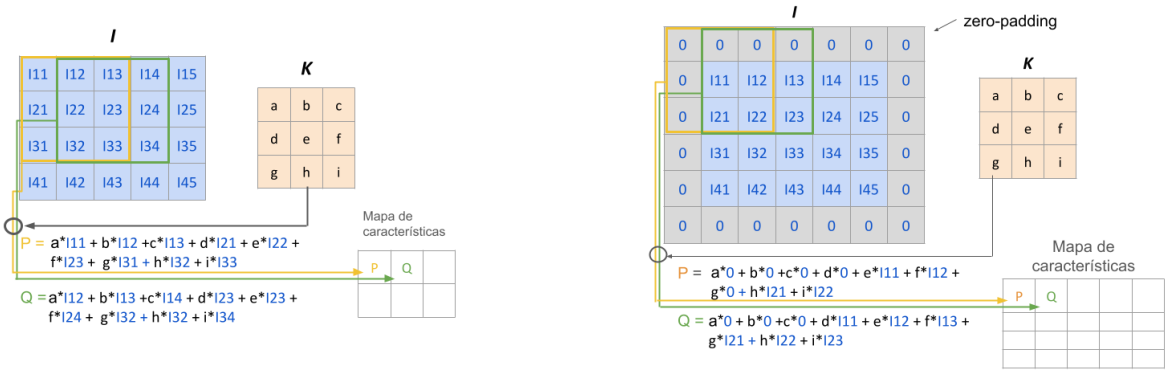
Capas convolucionales

La capa convolucional es la capa más importante de una CNN. En ella se extraen las características más significativas de la imagen de entrada, como los bordes, el color o la forma. Para ello se aplica una convolución a la imagen con un filtro. Esta operación matemática se representa como $\int(x \star w)(t)$ donde x representa la entrada y w el núcleo de convolución [48].

[1] Autocodificador para imágenes de gran tamaño

[2] Autocodificador específico para series temporales o secuencias.

En una CNN, la entrada de la convolución es una matriz multidimensional mientras que w es una matriz de parámetros, llamada núcleo o filtro, que se ajusta durante el aprendizaje. Cada píxel de la capa de convolución tiene una neurona, que se conecta con la capa anterior aplicando la convolución con las neuronas de su campo receptivo[3] [17]. Esta convolución se realiza con un solapamiento total del filtro, lo que resulta en una imagen de menor dimensión (Figura 2.15a). Si se desea mantener la misma dimensión, se puede aplicar zero-padding, que consiste en rellenar con ceros la matriz para obtener las dimensiones deseadas (Figura 2.15b). En la figura 2.15 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.



(a) Convolución reduciendo tamaño.

(b) Convolución con zero-padding.

Figura 2.15: Convolución en 2D.

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

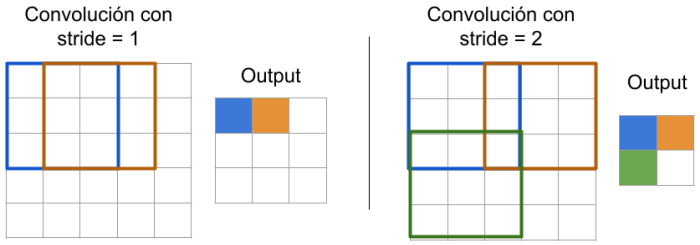


Figura 2.16: Campos receptivos en una convolución. Adaptación de imagen de [60]

Una longitud de paso de 1 se utiliza normalmente para extraer el máximo número de características, ya que proporciona el máximo solapamiento entre el núcleo y la entrada. Por otro lado, cuando la longitud de paso es mayor que 1, los campos receptivos se solapan menos y producen una salida más pequeña. Si la longitud de paso fuera 3, habría problemas con el espaciado, ya que el campo receptivo no encajaría alrededor de la entrada como un número entero [60].

Por simplicidad, se ha usado siempre un único kernel, pero se puede generalizar a varios filtros, creando un mapa de características por cada uno. En cada uno de estos mapas hay una neurona por pixel y todas ellas comparten los mismos parámetros, lo que reduce considerablemente el número de parámetros del modelo. El campo receptivo de una neurona ahora se extiende por los mapas de características de todas las capas anteriores [17].

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

[3]Región de entrada que contribuye a la salida generada por el filtro

$$z_{i,j,k} = b_k + \sum_{u=0}^{f_h-1} \sum_{v=0}^{f_w-1} \sum_{k'=0}^{f'_n-1} x_{i',j',k'} \cdot w_{u,v,k',k} \quad \text{con} \begin{cases} i' = i \cdot s_h + u \\ j' = j \cdot s_w + v \end{cases} \tag{2.13}$$

donde:

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

Capas de pooling

El siguiente tipo de capa de las CNN son las pooling, cuyo objetivo es reducir la imagen de entrada para disminuir la carga computacional, el uso de memoria y el número de parámetros, limitando así el riesgo de sobreajuste y proporcionando robustez contra el ruido y las distorsiones. Esta capa se suele colocar entre las capas de convolución, permitiendo reducir el tamaño de las imágenes mientras se preservan las características más importantes [50]. Al igual que en las capas convolucionales, sus neuronas están conectadas a un pequeño grupo de neuronas de la capa anterior a las que se le aplica una función de agregación[4]. Las tres funciones más comunes son el promedio, la suma y el máximo. La Figura 2.17 muestra una capa de max pooling, que es el tipo más común [17].

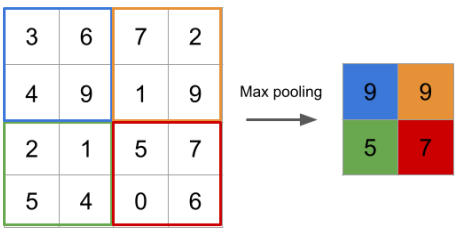


Figura 2.17: Capa de max pooling con un kernel de 2×2, stride 2 y sin padding.

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

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

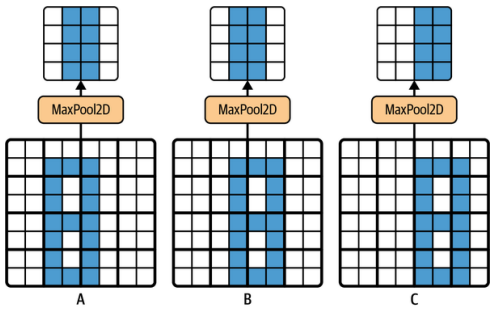


Figura 2.18: Invarianza a translaciones pequeñas mediante una capa de max pooling. Fuente [17]

Capas Totalmente Conectadas (Fully Connected)

Por último están las capas totalmente conectadas (*fully connected*), que realizan la clasificación sobre la salida generada por las capas de convolución y pooling. Como el input de una capa densa debe ser un vector, primero se debe aplanar la salida de la última capa para poder utilizar después esta capa. Cada una de sus neuronas está conectada a todas las de la capa anterior, estableciendo una red densa de conexiones. Este tipo de neuronas suele ir seguido de una capa Dropout para mejorar la generalización del modelo. Este diseño permite a las CNN manejar datos complejos y variados, aprovechando la jerarquía de características aprendidas durante el entrenamiento. Este tipo de capa suele ir seguido de una capa de Dropout que mejora la capacidad de generalización del modelo al prevenir el sobreajuste, un problema común en el ámbito del aprendizaje profundo [22].

2.10.4. Red Neuronal Recurrente

Restricted Boltzmann Machine

2.11. 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.

2.11.1. Principales frameworks. Keras

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

TensorFlow [20] es una biblioteca de código abierto desarrollada por el equipo de Google Brain para la computación numérica y el aprendizaje automático a gran escala. Diseñada para ser altamente flexible, TensorFlow soporta computación distribuida y permite la optimización de gráficos computacionales, lo que mejora significativamente la velocidad y el uso de memoria

de las operaciones. En su núcleo, TensorFlow es similar a NumPy pero con soporte para GPU, lo que acelera considerablemente los cálculos. Además, incluye herramientas avanzadas como TensorBoard para la visualización de modelos y TensorFlow Extended para la producción de modelos de aprendizaje automático. Gracias a estas capacidades, TensorFlow se ha convertido en una herramienta esencial en la industria y la investigación, siendo utilizada en aplicaciones que van desde la clasificación de imágenes y el procesamiento de lenguaje natural hasta los sistemas de recomendación y la previsión de series temporales.

Keras [19] es una API de alto nivel para redes neuronales que ahora es parte integral de TensorFlow. Fue desarrollada por François Chollet y ganó popularidad rápidamente gracias a su simplicidad y diseño elegante. Inicialmente, Keras soportaba múltiples backends, pero desde la versión 2.4, funciona exclusivamente con TensorFlow [42]. Keras permite a los usuarios construir, entrenar y evaluar modelos de aprendizaje profundo de manera rápida y eficiente. Su facilidad de uso y extensa documentación la convierten en una herramienta valiosa tanto para la investigación como para la implementación de aplicaciones de inteligencia artificial.

PyTorch [15], desarrollado por el equipo de investigación de IA de Facebook, es una biblioteca de aprendizaje profundo que destaca por su enfoque en la computación dinámica, lo que permite una mayor flexibilidad en la creación de modelos complejos. A diferencia de TensorFlow, que utiliza gráficos computacionales estáticos, PyTorch permite que la topología de la red neuronal cambie durante la ejecución del programa [36]. Esto, junto con su capacidad de auto-diferenciación en modo inverso[5], hace que PyTorch sea popular entre los investigadores y desarrolladores. Su facilidad de uso y robusta comunidad de apoyo han llevado a su adopción por parte de importantes organizaciones como Facebook, Twitter y NVIDIA.

Para escoger con cuál de estas librerías se realizará la parte práctica de este trabajo, vamos a utilizar, además de las características previamente vistas, los resultados de [36]. En él se hace un estudio de eficiencia, convergencia, tiempo de entrenamiento y uso de memoria de los diferentes frameworks con varios datasets. Entre sus resultados podemos observar como Keras destaca por encima de las demás en el entorno de la CPU. No solo logra el mejor accuracy en los tres datasets (MNIST, CIFAR-10, CIFAR-100), sino que además también tiene los tiempos de ejecución más bajos y una de las mejores tasas de convergencia. En cuento al entorno de la GPU, las tres librerías obtienen unos resultados semejantes. En conclusión, podemos afirmar que estos resultados junto con su facilidad de uso, accesibilidad y documentación bien estructurada, han sido determinantes para optar por usar Keras en vez de PyTorch o TensorFlow en nuestros estudios posteriores. Aakash Nain resume perfectamente las ventajas de Keras [2] al señalar que:

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

De manera similar, Matthew Carrigan destaca la intuitividad y facilidad de uso de Keras [40], afirmando:

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

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

2.11.2. Librerías y herramientas esenciales.

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

Scikit-Learn [51] es una librería de código abierto con herramientas simples y eficientes para el análisis predictivo de datos. Contiene varios algoritmos de aprendizaje automático, desde clasificación y regresión hasta clustering y reducción de dimensionalidad, con la documentación completa sobre cada algoritmo. Está construida sobre otras librerías que veremos más adelante como Numpy, SciPy y matplotlib. Aunque no se aprovecharán todas estas funcionalidades de scikit-learn, si que se va a utilizar una de sus funciones más populares, `train_test_split()` [52]. Esta función divide el dataset en dos subconjuntos de forma aleatoria, manteniendo la correspondencia en caso de que el dataset contenga dos o más partes. Usualmente, a estos subconjuntos se les llama conjunto de prueba y conjunto de entrenamiento, cuyo tamaño se indica con un valor entre 0 y 1 (`test_size`). Además, también se suele asignar una semilla a esa 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 codigo en Python
2 from sklearn.model_selection import train_test_split
3
4 X_train, X_test, y_train, y_test = train_test_split(data, labels,
5                                                    test_size=0.25, random_state=42)
```

Las variables X_train, X_ test y compañía son numpy arrays. **NumPy** [11] es el paquete fundamental de Python para la computación científica. Es una biblioteca general de estructuras de datos, álgebra lineal y manipulación de matrices para Python, cuya sintaxis y manejo de estructuras de datos y matrices es comparable al de MATLAB [8]. En NumPy, se pueden crear arrays y realizar operaciones rápidas y eficientes sobre ellos. Se utilizarán estas estructuras de datos para almacenar los datos y entrenar las redes neuronales con ellas. Aunque también se pueden utilizar tensores [10], se ha decidido utilizar numpy arrays por su alta eficiencia operacional y por su uso en la industria.

Otro paquete que se va a utilizar durante los experimentos y que Scikit-Learn utiliza es **matplotlib** [39]. Es la principal biblioteca de gráficos científicos en Python y proporciona funciones para crear visualizaciones de calidad como gráficos de barras, histogramas, gráficos de dispersión, etc. Se utilizará este paquete para representar gráficamente los datos de cada dataset para poder obtener bastante información con un simple vistazo.

Capítulo 3

Clasificación de Malware

Hoy en día, uno de los principales retos que enfrenta el software anti-malware es la enorme cantidad de datos y archivos que se requieren evaluar en busca de posibles amenazas maliciosas. Una de las razones principales de este volumen tan elevado de archivos diferentes es que los creadores de malware introducen variaciones en los componentes maliciosos para evadir la detección. Esto implica que los archivos maliciosos pertenecientes a la misma "familia" de malware (con patrones de comportamiento similares), se modifican constantemente utilizando diversas tácticas, lo que hace que parezcan ser múltiples archivos distintos [1].

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

Para enfrentar este tipo de problema, se va a escoger una de las bases de datos disponibles en [50] para poder clasificar distintos tipos de ciberataques. Como el objetivo principal de este trabajo es el estudio y puesta en práctica de diferentes algoritmos de aprendizaje automático, se ha decidido tomar como base de datos Microsoft Malware Classification Challenge. La principal razón de esta decisión ha sido que con este dataset tenemos a nuestra disposición dos algoritmos diferentes de machine learning que están referenciacios en este review y que se aborda el problema usando cada uno su propio enfoque.

3.1. Microsoft Malware Classification Challenge

El conjunto de datos utilizado en este estudio proviene del Microsoft Malware Classification Challenge (BIG 2015) [1], una competición dirigida a la comunidad científica con el objetivo de promover el desarrollo de técnicas efectivas para agrupar diferentes variantes de malware. Se decidió escoger este dataset porque el objetivo que tengo en este trabajo es el de aprender y desarrollar diferentes métodos de aprendizaje automático y este dataset nos permite utilizar tanto una CNN como un Autoencoder según [50].

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

Este dataset contiene 5 archivos:

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

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

Cada programa malicioso tiene un identificador, un valor hash de 20 caracteres que identifica de forma única el archivo, y una etiqueta de clase, que es un número entero que representa una de las 9 familias de malware al que puede pertenecer. Por ejemplo, el programa *0ACDbR5M3ZhBJajygTuf* tiene como etiqueta el valor 7. Esta información se puede consultar en el archivo "trainLabels.csv". Cada programa tiene dos archivos, uno asm con el código extraído por la herramienta de desensamblado IDA y otro bytes[1] con la representación hexadecimal del contenido binario del programa pero sin los encabezados ejecutables (para garantizar esterilidad). Para nuestro estudio vamos a utilizar únicamente este ultimo archivo.

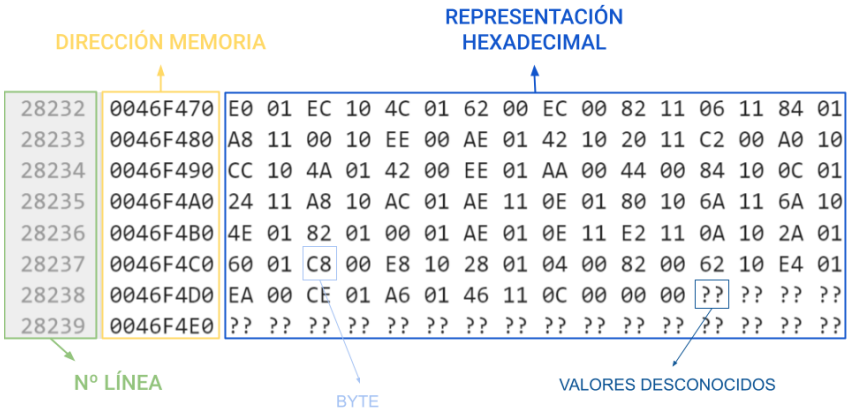


Figura 3.1: Explicación del contenido de "0ACDbR5M3ZhBJajygTuf.bytes".

Como aparece en la figura 3.1, los ocho primeros caracteres son direcciones de memoria, seguido de la representación hexadecimal del contenido binario del programa, que contiene 16 bytes (cada uno dos caracteres). A veces nos podemos encontrar con "??" en el lugar de un byte. Este símbolo se utiliza en estos archivos para representar que se desconoce su información porque su memoria no se puede leer [9].

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

3.1.1. Distribución del dataset

Hay un total de 21.741 programas de malware, pero solo 10.868 de ellos tienen etiquetas. Estos programas pertenecen a una de estas 9 familias de malware: Rammit, Lollipop, Kelihos_ ver3, Vundo, Simda, Tracur, Kelihos_ ver1, Obfuscator y Gatak. Según [23], podemos deinirlas como:

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

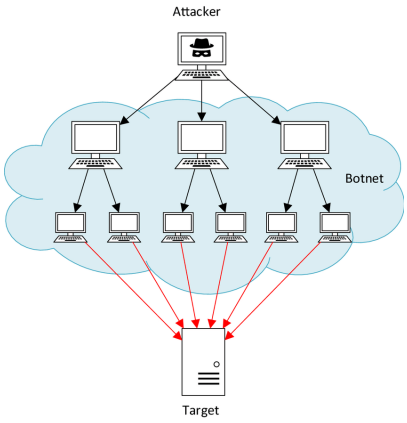


Figura 3.2: Estructura de un botnet. Imagen sacada de [43].

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

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

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

5. **Kelihos_ ver3** es un troyano tipo backdoor que distribuye correos electrónicos que pueden contener enlaces falsos a instaladores de malware. Consta de tres tipos de bots [28]: controladores (operados por los dueños y donde se crean las instrucciones), enrutadores (redistribuyen las instrucciones a otros bots) y trabajadores (ejecutan las instrucciones).

6. **Tracur** es un descargador troyano que agrega el proceso 'explorer.exe' a la lista de excepciones del Firewall de Windows para disminuir deliberadamente la seguridad del sistema y permitir la comunicación no autorizada a través del firewall. Además, esta familia también te puede redirigir a enlaces maliciosos para descargar e instalar otros tipos de malware.

7. **Kelihos_ ver1** es una versión más antigua del troyano Kelihos_ ver3, pero con las mismas funcionalidades.

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

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

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

8. **Obfuscator.ACY** es un tipo de malware sofisticado que oculta su propósito y podría sobrepasar las capas de seguridad del software. Se puede propagar mediante archivos adjuntos de correo electrónico, anuncios web y descargas de archivos.

9. **Gatak** es un troyano que abre una puerta trasera en el ordenador. Se propaga a través de sitios web falsos que ofrecen claves de licencias de productos. Una vez infectado el sistema, Gatak recopila información del ordenador.

Como ya mencionamos antes, solo hay 10.868 programas con etiquetas, luego vamos a hacer el análisis descriptivo de los datos solo con estos archivos. De estos programas, solo son válidos 10.860 porque en los 8 archivos restantes[5] (pertenecientes a la familia Ramnit), todo sus bytes son "??", es decir, información desconocida. Con estos datos, vamos a ver gráficamente como se distribuyen en las 9 clases de malware (Figura 3.3).

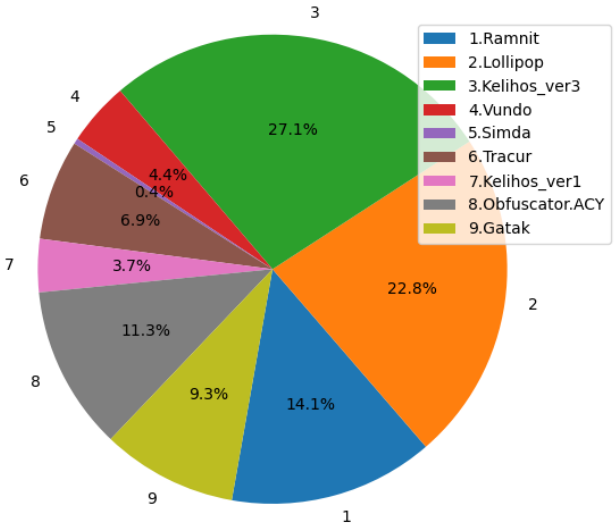


Figura 3.3: Distribución del BIG 2015 training dataset.

Analizando la Figura 3.3, podemos observar como la distribución entre las clases no es uniforme. Mientras que de la clase Simbda hay 42 muestras, de la clase Kelihos_ ver3 hay 2.942, es decir, 70 veces más de muestras. En [27] deciden prescindir de esta clase, pero nosotros hemos decidido hacer el análisis con las 9 clases.

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

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

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

[5]Los identificadores de estos archivos son 58kxhXouHzFd4g3rmInB, 6tfw0xSL2FNHOCJBdlaA, a9oIzfw03ED4lTBCt52Y, cf4nzsoCmudt1kwleOTI, d0iHC6ANYGon7myPFzBe, da3XhOZzQEbKVtLgMYWv, fRLS3aKkijp4GH0Ds6Pv, IidxQvXrlBkWPZAfcqKT.

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

3.2. Red Neuronal Convolucional

Para abordar este problema utilizando como modelo una CNN, solo podemos utilizar los datos etiquetados ya que este método es un método supervisado. De los 21.741 programas de malware con los que disponemos, solo podemos utilizar los 10.860 que están etiquetados y contienen información disponible. Como ya vimos en la sección 2.10.3, para entrenar estas redes neuronales es necesario tener los datos en forma matricial. Uno de los principales motivos para convertir el malware en imagen es porque los creadores de malware suelen modificar sus implementaciones para producir nuevo malware [45], pero si lo representamos de forma matricial, estos pequeños cambios pueden ser detectados fácilmente [25].

3.2.1. Visualizar el malware como imagen

Para visualizar los archivos .bytes como imagen en escala de grises, cada byte debe ser interpretado como un pixel en la imagen. Inspirado en [46], para pasar de código hexadecimal a imagen primero pasamos cada byte a su número decimal correspondiente que se encuentra en el rango [0,255][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 [17], se plantea eliminar estos caracteres y tratar el resto de bytes. Otra solución la proponen Narayanan et al. [44], que es sustituir estos bytes por el valor -1 (color blanco). Después de probar con ambas propuestas y además la de cambiando el "??" por el valor 0, finalmente hemos decidido sustituirlo por 0 (color negro) en base a los resultados obtenidos.

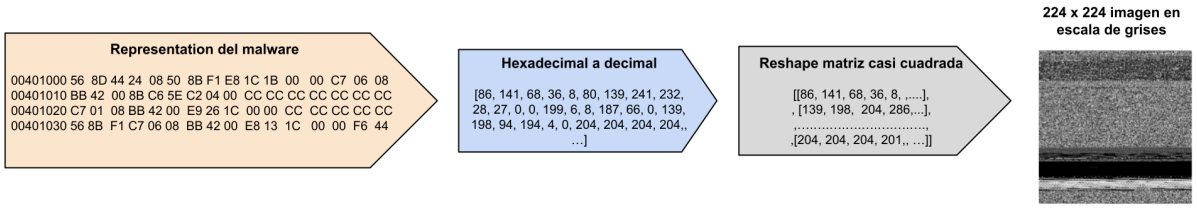


Figura 3.4: Proceso de visualización del malware. Adaptación de [46]

Después de tener todos los bytes en formato decimal dentro de un vector, hacemos un reshape a una matriz 2D de forma que se consiga una matriz lo más cuadrada posible. Como las dimensiones de cada archivo son diferentes, las dimensiones después de hacer el reshape también lo serán, luego tenemos que fijar un tamaño fijo para poder entrenar nuestra CNN [32]. Para ello, siguiendo el ejemplo de [25], decidimos escoger como tamaño 224×224. Para obtener estas dimensiones, Simonyan et al. [53] deciden recortar aleatoriamente un cuadrado de la imagen de tamaño 224×224, pero nosotros hemos decidido usar interpolación lineal. En la figura 3.5, se puede observar cuales son los patrones que sigue cada tipo de malware en su forma matricial.

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

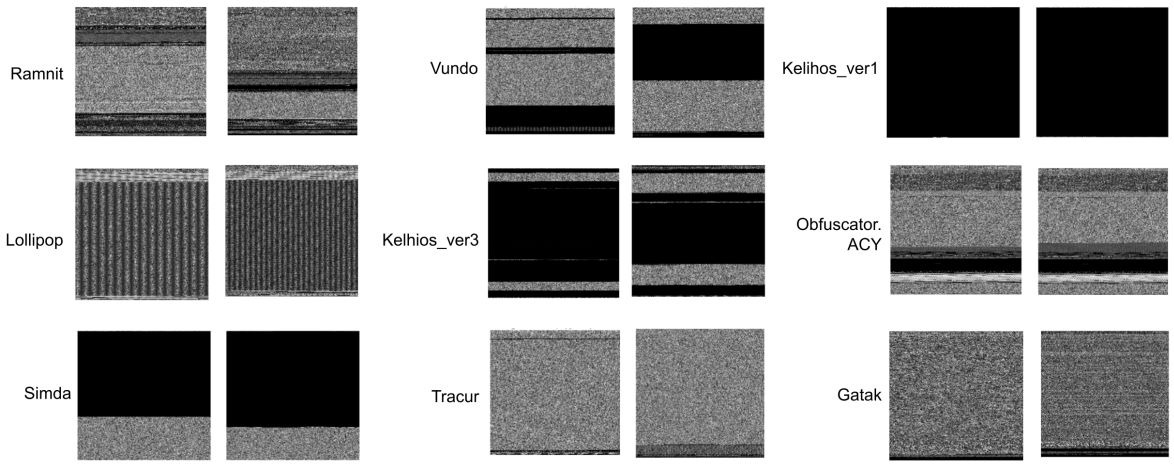


Figura 3.5: Visualización familias malware.

3.2.2. Visualización del modelo

Una vez ya hemos explorado y preparado los datos, el siguiente paso es crear nuestra red neuronal convolucional. Para ello hemos seguido el artículo [25], en el que se crea una M-CNN (malware CNN) con múltiples capas. Vemos su arquitectura en la Figura 3.6.

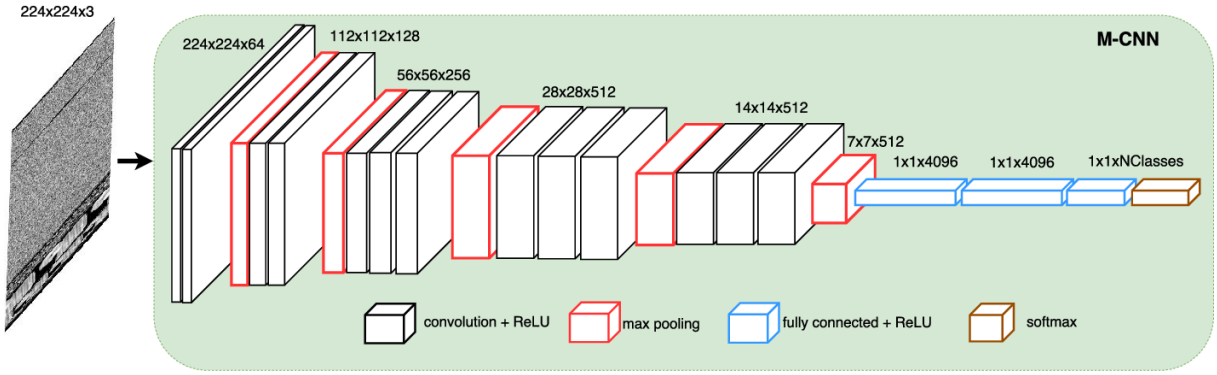


Figura 3.6: Arquitectura de la CNN. Fuente [25].

En las capas iniciales, se aplican filtros 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[17]. Después de cada operación de convolución, se aplica la función de activación ReLU para introducir no linealidades en el modelo.

Las capas de max pooling reducen la dimensionalidad de los mapas de características seleccionando el valor máximo en sub-regiones de 2×2, lo que no solo reduce la carga computacional sino que también ayuda a hacer las características más robustas a pequeñas variaciones y traslaciones en la imagen de entrada[17].

La secuencia de capas convolucionales y de pooling se repite varias veces hasta obtener un conjunto final de características que es una matriz de 7×7 con 512 mapas de características. Esta salida se aplana para convertirla en un vector unidimensional, que luego pasa a través de varias capas completamente conectadas (fully connected layers). Finalmente, la capa de salida

utiliza una función de activación softmax para generar las predicciones finales del modelo.

Una vez definida la arquitectura de la red, procedemos a compilar y entrenar el modelo. Cabe recordar que los datos fueron divididos en tres conjuntos: entrenamiento (75 %), validación (10 %) y prueba (15 %).

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

El entrenamiento se realizó con un batch_size de 8 y se ejecutó durante 25 epochs. Utilizamos callbacks para ajustar dinámicamente el learning rate, detener el entrenamiento temprano si no se observan mejoras en la pérdida de validación, y registrar el progreso del entrenamiento. Además, se barajea el conjunto de datos de entrenamiento antes de cada epoch.

3.2.3. Mejora del modelo

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

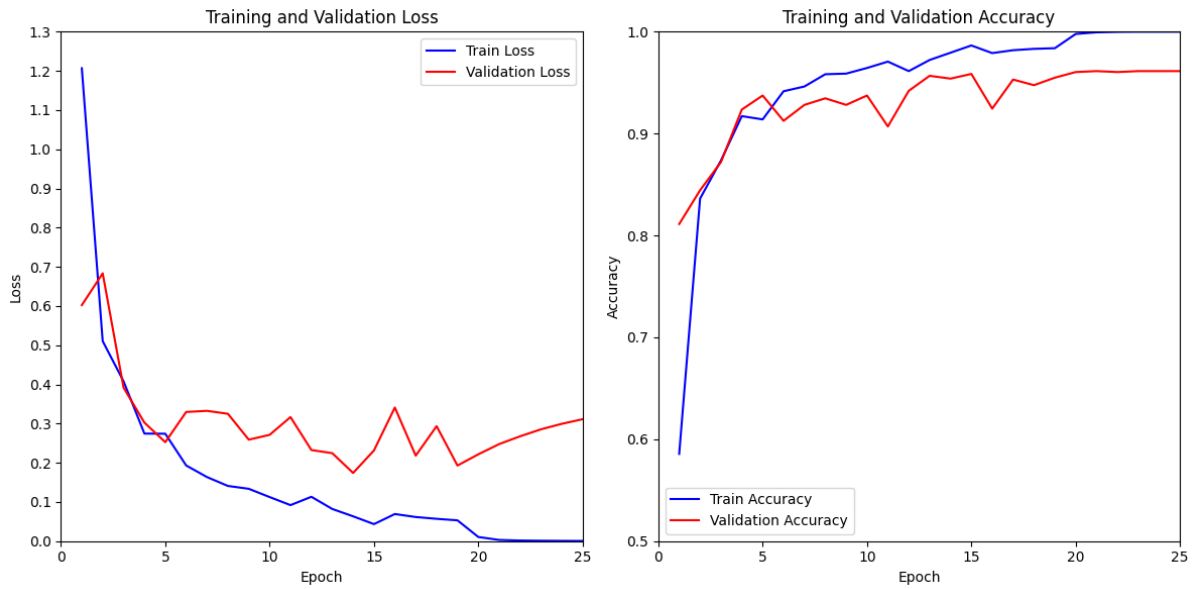


Figura 3.7: Evolución del modelo en términos de accuracy y loss.

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

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

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

3.2.4. GPU (Unidad de Procesamiento Gráfico)

Ventajas

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

Desventajas

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

3.2.5. CPU (Unidad Central de Procesamiento)

Ventajas

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

Desventajas

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

3.3. Autoencoder

Convolutional autoencoder modelo [**?**]

3.4. Resultados

[18] mete imagen de asm junto con bytes.

Capítulo 4

Detección de intrusiones

4.1. KDD Cup 1999

4.2. Autoencoder

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

- Usamos autoencoder para comprimir la información de entrada y despues esa informacion la usamos para clasificarla usando una DNN [34]
- Usamos una cadena de autoencoders en el cual la salida de h es la entrada del autoencoder h+1. Utilizo el articulo [16] donde se desarrolla todo el modelo y explicacion y ademas se hace referencia al artículo [6] 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 [5] explica bastante bien la diferencia entre capa autoencoder y un autoencoder.

4.3. Red Neuronal Convolucional

Para clasificar los datos del dataset KDD 1999 usando las Convolutional Neural Network (CNN) vamos a seguir los siguientes articulos [29, 59, 47, 31]. Prácticamente todo el cuerpo del experimento se encuentra en el artículo [29], pero en el artículo [31] aparece la parte de normalización de los datos y algunos 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 [37]. Además, hay buena explicacion del experimento en [57]. Por último, en el articulo [14] están los experimentos con DNN, RNN, RBM que puedo tomar también como referencia porque está muy bien explicado las capas e hiperparametros que utiliza.

4.5. Red Neuronal Recurrente

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

4.6. Restricted Boltzmann Machine

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

4.7. Resultados

Capítulo 5

Conclusiones y Trabajo Futuro

5.1. Conclusiones

5.2. Trabajo futuro

Bibliografía

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

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

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

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

[5] Wei Bao, Jun Yue, and Yulei Rao. A deep learning framework for financial time series using stacked autoencoders and long-short term memory. *PloS one*, 12(7):e0180944, 2017.

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

[7] Daniel S Berman, Anna L Buczak, Jeffrey S Chavis, and Cherita L Corbett. A survey of deep learning methods for cyber security. *Information*, 10(4):122, 2019.

[8] Marcus D Bloice and Andreas Holzinger. A tutorial on machine learning and data science tools with python. *Machine Learning for Health Informatics: State-of-the-Art and Future Challenges*, pages 435–480, 2016.

[9] Niken Dwi Wahyu Cahyani, Erwid M Jadied, Nurul Hidayah Ab Rahman, and Endro Ariyanto. The influence of virtual secure mode (vsm) on memory acquisition. *International Journal of Advanced Computer Science and Applications*, 13(11), 2022.

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

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

[12] Ali Diba, Mohsen Fayyaz, Vivek Sharma, Amir Hossein Karami, Mohammad Mahdi Arzani, Rahman Yousefzadeh, and Luc Van Gool. Temporal 3d convnets: New architecture and transfer learning for video classification. *arXiv preprint arXiv:1711.08200*, 2017.

[13] P Dileep, Dibyaiyoti Das, and Prabin Kumar Bora. Dense layer dropout based cnn architecture for automatic modulation classification. In *2020 national conference on communications (NCC)*, pages 1–5. IEEE, 2020.

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

[15] Facebook AI Research. Pytorch, 2017.

[16] Fahimeh Farahnakian and Jukka Heikkonen. A deep auto-encoder based approach for intrusion detection system. In *2018 20th International Conference on Advanced Communication Technology (ICACT)*, pages 178–183. IEEE, 2018.

[17] Aurélien Géron. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow*. .ºReilly Media, Inc.", 2022.

[18] Daniel Gibert, Jordi Planes, Carles Mateu, and Quan Le. Fusing feature engineering and deep learning: A case study for malware classification. *Expert Systems with Applications*, 207:117957, 2022.

[19] Google AI Team. Keras, 2015.

[20] Google Brain Team. Tensorflow, 2015.

[21] Sanchit Gupta, Harshit Sharma, and Sarvjeet Kaur. Malware characterization using windows api call sequences. In *Security, Privacy, and Applied Cryptography Engineering: 6th International Conference, SPACE 2016, Hyderabad, India, December 14-18, 2016, Proceedings 6*, pages 271–280. Springer, 2016.

[22] Md Anwar Hossain and Md Shahriar Alam Sajib. Classification of image using convolutional neural network (cnn). *Global Journal of Computer Science and Technology*, 19(2):13–14, 2019.

[23] Yen-Hung Frank Hu, Abdinur Ali, Chung-Chu George Hsieh, and Aurelia Williams. Machine learning techniques for classifying malicious api calls and n-grams in kaggle data-set. In *2019 SoutheastCon*, pages 1–8. IEEE, 2019.

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

[25] Mahmoud Kalash, Mrigank Rochan, Noman Mohammed, Neil DB Bruce, Yang Wang, and Farkhund Iqbal. Malware classification with deep convolutional neural networks. In *2018 9th IFIP international conference on new technologies, mobility and security (NTMS)*, pages 1–5. IEEE, 2018.

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

[27] Temesguen Messay Kebede, Ouboti Djaneye-Boundjou, Barath Narayanan Narayanan, Anca Ralescu, and David Kapp. Classification of malware programs using autoencoders based deep learning architecture and its application to the microsoft malware classification challenge (big 2015) dataset. In *2017 IEEE National Aerospace and Electronics Conference (NAECON)*, pages 70–75. IEEE, 2017.

[28] Max Kerkers, José Jair Santanna, and Anna Sperotto. Characterisation of the 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.

[29] Jiyeon Kim, Jiwon Kim, Hyunjung Kim, Minsun Shim, and Eunjung Choi. Cnn-based network intrusion detection against denial-of-service attacks. *Electronics*, 9(6):916, 2020.

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

[31] Taejoon Kim, Sang C Suh, 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.

[32] Sushil Kumar et al. Mcft-cnn: Malware classification with fine-tune convolution neural networks using traditional and transfer learning in internet of things. *Future Generation Computer Systems*, 125:334–351, 2021.

[33] Runze Lin. Analysis on the selection of the appropriate batch size in cnn neural network. In *2022 International Conference on Machine Learning and Knowledge Engineering (MLKE)*, pages 106–109. IEEE, 2022.

[34] Ivandro O Lopes, Deqing Zou, Ihsan H Abdulqadder, Francis A Ruambo, Bin Yuan, and Hai Jin. Effective network intrusion detection via representation learning: A denoising autoencoder approach. *Computer Communications*, 194:55–65, 2022.

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

[36] Nesma Mahmoud, Youssef Essam, Radwa 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.

[37] Mohammed Maithem and Ghadaa A Al-Sultany. Network intrusion detection system using deep neural networks. In *Journal of Physics: Conference Series*, volume 1804, page 012138. IOP Publishing, 2021.

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

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

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

[41] Montreal University. Theano, 2010.

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

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

[44] Barath Narayanan Narayanan, Ouboti Djaneye-Boundjou, and Temesguen M Kebede. Performance analysis of machine learning and pattern recognition algorithms for malware classification. In *2016 IEEE national aerospace and electronics conference (NAECON) and ohio innovation summit (OIS)*, pages 338–342. IEEE, 2016.

[45] Lakshmanan Nataraj, Shanmugavadivel Karthikeyan, and BS Manjunath. Sattva: Sparsity inspired classification of malware variants. In *Proceedings of the 3rd ACM Workshop on Information Hiding and Multimedia Security*, pages 135–140, 2015.

[46] Lakshmanan Nataraj, Sreejith Karthikeyan, Gregoire Jacob, and Bangalore S Manjunath. Malware images: visualization and automatic classification. In *Proceedings of the 8th international symposium on visualization for cyber security*, pages 1–7, 2011.

[47] Sinh-Ngoc Nguyen, Van-Quyet Nguyen, Jintae Choi, and Kyungbaek Kim. Design and implementation of intrusion detection system using convolutional neural network for dos detection. In *Proceedings of the 2nd international conference on machine learning and soft computing*, pages 34–38, 2018.

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

[49] Van Hiep Phung and Eun Joo Rhee. A deep learning approach for classification of cloud image patches on small datasets. *Journal of information and communication convergence engineering*, 16(3):173–178, 2018.

[50] Prajoy Podder, Subrato Bharati, M Mondal, Pinto Kumar Paul, and Utku Kose. Artificial neural network for cybersecurity: A comprehensive review. *arXiv preprint arXiv:2107.01185*, 2021.

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

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

[53] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

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

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

[56] Tokyo University. Chainer, 2015.

[57] Rahul K Vigneswaran, R Vinayakumar, KP Soman, and Prabaharan Poornachandran. Evaluating shallow and deep neural networks for network intrusion detection systems in cyber security. In *2018 9th International conference on computing, communication and networking technologies (ICCCNT)*, pages 1–6. IEEE, 2018.

[58] Jin Wang, Zhongyuan Wang, Dawei Zhang, and Jun Yan. Combining knowledge with deep convolutional neural networks for short text classification. In *IJCAI*, volume 350, pages 3172077–3172295, 2017.

[59] Zhongxue Yang and Adem Karahoca. An anomaly intrusion detection approach using cellular neural networks. In *Computer and Information Sciences–ISCIS 2006: 21th International Symposium, Istanbul, Turkey, November 1-3, 2006. Proceedings 21*, pages 908–917. Springer, 2006.

[60] Juan Yepez and Seok-Bum Ko. Stride 2 1-d, 2-d, and 3-d winograd for convolutional neural networks. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 28(4):853–863, 2020.

[61] Ilsun You and Kangbin Yim. Malware obfuscation techniques: A brief survey. In *2010 International conference on broadband, wireless computing, communication and applications*, pages 297–300. IEEE, 2010.

[62] Mohamad Fadli Zolkipli and Aman Jantan. Malware behavior analysis: Learning and understanding current malware threats. In *2010 Second International Conference on Network Applications, Protocols and Services*, pages 218–221. IEEE, 2010.

.1. Anexo A

Cuadro 1: Tabla de códigos hexadecimales y decimales

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