TRABAJOS FIN DE GRADO

Universidad de Jaén

Escuela Politécnica
Superior de Jaén

Deep Learning aplicado a la detección de amenazas de seguridad

José Manuel Martínez Ramírez

Grado en Ingeniería Informática

Directores: Manuel José Lucena López
Cristóbal José Carmona del Jesús

Departamento de Informática

Fecha: XX/06/2024

UNIVERSIDAD DE JAÉN

D. Manuel José Lucena López y D. Cristóbal José Carmona del Jesús, tutor(es) del Trabajo Fin de Grado titulado: ***Deep Learning*** **aplicado a la detección de amenazas de seguridad**, que presenta José Manuel Martínez Ramírez, autoriza(n) su presentación para defensa y evaluación en la Escuela Politécnica Superior de Jaén.

Jaén, junio de 2024

El estudiante	Los tutores	
MARTINEZ RAMIREZ JOSE MANUEL - 77374369T Firmado digitalmente por MARTINEZ RAMIREZ JOSE MANUEL - 77374369T Nombre de reconocimiento (DN): c=ES, serialNumber=IDCES-77374369T, givenName=JOSE MANUEL, sn=MARTINEZ RAMIREZ, cn=MARTINEZ RAMIREZ JOSE MANUEL - 77374369T Fecha: 2024.06.04 15:45:44 +02'00'	Firmado por LUCENA LOPEZ MANUEL JOSE - ***7735** el día 04/06/2024	CARMONA DEL JESUS CRISTOBAL JOSE - 75102603J Firmado digitalmente por CARMONA DEL JESUS CRISTOBAL JOSE - 75102603J Fecha: 2024.06.04 15:58:50 +02'00'
José Manuel Martínez Ramírez	Manuel José Lucena López	Cristóbal José Carmona del Jesús

Agradecimientos

Me gustaría comenzar esta memoria agradeciendo a mis tutores, Cristóbal J. Carmona y Manuel J. Lucena, por haberme brindado su apoyo para llevar a cabo este TFG, y por haber inculcado en mí no solo el gusto por sus respectivos campos de la ciberseguridad y la inteligencia artificial, sino también por animarme y guiarme para lograr llevarlo a cabo.

A continuación, me gustaría dar las gracias a mi familia, y particularmente a mis padres, José Manuel y María Jesús. Pese a que sus campos de conocimiento difieren considerablemente del de este trabajo, no por ello han mostrado menos interés en ayudarme con sus consejos e interesarse por el avance del mismo, por no mencionar los ánimos que me han dado desde siempre para continuar aprendiendo e investigando.

Para continuar, quiero dar las gracias a todos los amigos que me han acompañado a lo largo de la elaboración de este trayecto, compartiendo conmigo tiempo de espera mientras los modelos empleados se ejecutaban. Si bien están repartidos por toda España e incluso más allá en algunos casos, se sienten cercanos, y particularmente a Elegguá, Eduardo, Adela, Alberto y Luca. También me gustaría dar un agradecimiento especial a Andrea, por ser de quienes más apoyo me ha brindado y soportar mis quejas incesantes sobre los errores de código que he tenido que resolver.

Por último, agradecer a todos los profesores que me han enseñado durante la carrera, ayudándome a encontrar mi camino en el mundo de la informática a lo largo de cuatro años. Sin cada uno de ellos, no sabría lo que sé ahora ni hubiera sido capaz de llegar hasta este punto.

Tabla de contenidos

Lista de figuras

Lista de tablas

Lista de algoritmos

Lista de listados de código

Glosario de términos

En la memoria del TFG se emplearán a menudo diversos términos. A continuación se presenta un índice que contiene todos los acrónimos y términos utilizados.

- 0-day: Ataque de día cero. Ataque posible gracias a vulnerabilidades en un producto desconocidas para sus usuarios y creadores.
- ANN: Artificial Neural Network. Red Neuronal Artificial.
- CIC: Canadian Institute of Cybersecurity.
- CNN: Convolutional Neural Network.
- CSE: Communications Security Establishment.
- DL: Deep Learning o Aprendizaje Profundo.
- DBN: Deep Bayesian Network. Red bayesiana profunda.
- ELMo: Embedding from Language Models.
- GAN: Generative Adversarial Network. Red generativa adversarial.
- IA: Inteligencia Artificial.
- IAT: Inter-Arrival Time.
- KDD: Knowledge Discovery in Databases. Descubrimiento del conocimiento en bases de datos.
- LLM: Large Language Model. Modelo de lenguaje natural.
- LR: Learning Rate. Ratio de aprendizaje.
- LSTM: Long Short-Term Memory.
- MEBN: Multi-Entity Bayesian Network. Red Bayesiana Multi-Entidad.
- SDI: Sistema de Detección de Intrusiones.
- SVM: Support Vector Machine. Máquina de Soporte Vectorial.
- TFG: Trabajo Fin de Grado.
- WNN: Wavelet Neural Network.

Capítulo 1

INTRODUCCIÓN

En este primer capítulo se explicarán brevemente los contenidos del presente TFG, en el que se propone el uso de diversas tecnologías de aprendizaje profundo con objetivo de generar un modelo de clasificación capaz de predecir si determinado tráfico es benigno o un ataque.

Los contenidos a continuación explicados permitirán entender más fácilmente los antecedentes del trabajo, explicados en el segundo capítulo.

1.1. Fundamentos

En este apartado se introducirán una serie de conceptos relacionados con el problema a tratar y que tienen relación con los modelos propuestos más adelante. Se comenzará explicando el concepto de **ciencia de datos**.

1.1.1. Ciencia de datos

La ciencia de datos es una ciencia que combina matemáticas, informática y estadística, entre otras disciplinas. Consiste en la extracción, tratamiento y análisis de datos con el objetivo de obtener conocimiento a partir de ellos. Dentro de ella, se pueden encontrar diversos métodos para lograr este fin, incluyendo el aprendizaje máquina (*Machine Learning*) [1] o el aprendizaje profundo (DL) [2].

Se hará especial hincapié sobre el DL en la memoria, ya que se empleará para

llevar a cabo un proceso de **minería de datos** con el objetivo de resolver un problema de **clasificación** que permita distinguir el tráfico benigno del malicioso a partir de diversas bases de datos con información de múltiples conexiones de índole variada.

1.1.2. Descubrimiento de conocimiento en bases de datos

Durante mucho tiempo, el mayor problema a la hora de tratar de extraer información de los datos recogidos era, precisamente, conseguir una cantidad de datos a analizar lo suficientemente grande como para poder obtener la información buscada. Sin embargo, con el desarrollo de diversas herramientas, entre las que destaca Internet, los datos hoy día se generan y obtienen a una velocidad vertiginosa.

El problema ya no radica en obtener datos de los que extraer información, sino en el conjunto de procesos a los que hay que someter dichos datos para poder lograrlo. Por esto precisamente surge el concepto del **descubrimiento de conocimiento en bases de datos**, o *KDD* por sus siglas en inglés, propuesto originalmente por Brachman y Anand [3] y expandido por Fayyad et al. [4]. El KDD es un proceso iterativo e interactivo compuesto por diversas etapas, expuestas a continuación:

1. **Integración y recopilación de los datos**: en esta primera etapa, se busca alcanzar la comprensión completa del dominio del problema, a menudo colaborando con expertos en el mismo. También se debe identificar el conocimiento *a priori* para partir de él como base y crear el almacén electrónico (*data warehouse* [5]) que contendrá los datos relevantes para el problema.

2. **Preprocesamiento de los datos**: esta etapa es la más costosa de todo el proceso. Consiste en elegir el conjunto de datos adecuado para el resto del proceso de KDD. Las tareas de esta etapa se pueden repartir en tres subetapas [6]:

    - **Limpieza de datos**. En esta primera etapa, se busca operar sobre los datos para limpiarlos y, en consecuencia, mejorar el funcionamiento de los algoritmos. Concretamente, se busca rellenar **valores perdidos**, esto es, valores desconocidos para algún dato del problema; **suavizar el ruido**, datos incorrectos que se han introducido en la base de datos por errores a la hora de tomar sus valores; **detectar y tratar valores anómalos**, que suelen conllevar valores extremadamente altos y bajos y que, si bien son correctos, pueden influenciar negativamente los resultados del proceso y, por último, **resolver inconsistencias** de cualquier otra índole que se encuentren en los datos.

- **Transformación de los datos**. En esta etapa, se opera sobre los datos para conseguir un formato que logre mejorar el funcionamiento de los algoritmos que se aplicarán sobre ellos [7]. Entre estas operaciones cabe destacar, entre otras, la **agregación**, que agrupa o totaliza varios atributos en uno solo; la **normalización**, que cambia el rango del dominio de los datos; la **creación de nuevos atributos**, habitualmente combinando otros ya existentes, ya que pueden aportar más información y la **discretización**, que facilita el manejo de los atributos continuos.
- **Reducción de la dimensionalidad**. En esta etapa, se pretende obtener una representación reducida del conjunto de datos de menor volumen que preserve la información relevante contenida en los datos originales. Esta fase es opcional, ya que en muchas ocasiones se puede trabajar directamente sobre el conjunto de datos completo sin necesidad de reducirlo.

3. **Minería de datos**: en esta etapa se realiza el análisis de los datos trabajados en etapas previas. Se verá en mayor detalle a continuación, en esta misma sección.
4. **Evaluación de resultados**: en esta etapa, se deben interpretar y presentar los datos obtenidos a partir del proceso de minería de datos. A menudo, lo que generará el proceso de minería de datos previo será un conjunto de hipótesis de modelos, por lo que en esta etapa se deberá elegir cuál de todos es el más adecuado en base a la precisión de los mismos, así como lo interesantes que resulten de cara al problema a resolver y lo fáciles de comprender que resulten.
5. **Difusión y utilización del nuevo conocimiento**: una vez el modelo ha sido construido y validado, debe usarse para recomendar acciones o para aplicarse sobre diversos conjuntos de datos, ya sea utilizándolo de forma independiente o incorporándolo a sistemas ya existentes.

Minería de datos

La minería de datos es la etapa más significativa del proceso de descubrimiento de conocimiento en bases de datos hasta el punto de que, a menudo, ambos términos se equiparan coloquialmente.

Durante esta fase, se aplica una serie de algoritmos sobre los datos que se están manejando con objetivo de extraer información de los mismos o analizarlos. El resultado de esta etapa es un modelo que permita bien agrupar los datos, bien clasificarlos, etc. Debido a que la minería de datos es un campo muy amplio, a continuación se desarrollarán con mayor detenimiento las características de esta etapa.

Enfoques

Los algoritmos de minería de datos pueden ser clasificados en base a diversos criterios:

- En base a la estructura de los datos:
  - **Aprendizaje supervisado**: los datos deben contener un campo concreto, denominado **etiqueta**, que determina para cada ejemplo su tipo o clase. Los algoritmos, en consecuencia, buscan patrones que permitan identificar la clase de un ejemplo determinado en base al resto de sus propiedades.
  - **Aprendizaje no supervisado**: los datos no contienen una etiqueta, al contrario que en el caso previo. Los algoritmos en este caso buscan patrones de los que extraer información a partir de todos los atributos y sus relaciones entre sí.
  - **Aprendizaje semisupervisado**: es una combinación de los dos tipos previos. Algunos de los datos del conjunto de datos (*dataset*) presentan una etiqueta, mientras que otros no. Habitualmente, se construye un modelo a partir de los datos etiquetados, y se usan los que no lo están para refinarlo.
  - **Aprendizaje por refuerzo**: en ocasiones, ni siquiera los humanos sabemos clasificar de forma totalmente determinista un ejemplo, pero sí sabemos determinar si es mejor o peor que otro ejemplo determinado. El aprendizaje por refuerzo se basa en esta idea, refinando el modelo mediante un sistema de recompensas en función a las acciones que toma.
- En base al objetivo del algoritmo:
  - **Minería de datos descriptiva**: pretende ser capaz de describir la información de los datos a partir de sus atributos, habitualmente centrándose en las relaciones que existen entre ellos, su estructura, etc. Suele emplearse con aprendizaje no supervisado.
  - **Minería de datos predictiva**: pretende generar un modelo que pueda predecir un valor de salida determinado: su clase, si el problema es de **clasificación**, o un valor numérico, si el problema es de **regresión**. Habitualmente está estrechamente relacionado con el aprendizaje supervisado.

Tareas

La minería de datos permite llevar a cabo tareas muy diversas:

- **Clasificación**: consiste en asignar a los datos una etiqueta determinada. Se verá con más detenimiento en el apartado 1.1.4.
- **Regresión**: consiste en predecir una salida numérica continua a partir de los datos de entrada.
- **Manejo de series temporales**: busca analizar una serie de datos que están relacionados entre sí cronológicamente, además de por la información que puedan contener.
- **Agrupamiento**: busca crear grupos, o *clusters*, de datos que tienen características comunes o similares.
- **Asociaciones**: permiten encontrar relaciones entre datos, detectando de esta manera qué elementos conllevan la presencia o ausencia de otros.

Técnicas

Para llevar a cabo todos los procesos previamente descritos, se emplean una serie de técnicas. Algunas de las más relevantes se muestran a continuación:

- **Creación de modelos**: la creación de modelos consiste en generar un conjunto de datos, patrones y estadísticos que se puedan aplicar sobre nuevos datos con el objetivo de generar predicciones y deducir relaciones.
- **Análisis de regresión**: particularmente usado en problemas de regresión, el análisis de regresión se emplea para determinar la relación lineal entre variables, permitiendo en consecuencia la predicción del valor futuro de las mismas.
- **Patrones secuenciales**: esta técnica se enfoca en el descubrimiento de patrones dentro de una secuencia de sucesos. Es usado habitualmente en minería de datos transaccionales, pero también por empresas para crear sistemas de recomendación, entre otros usos.
- **Detección de valores atípicos**: los datos de los que se pretende extraer información a menudo cuentan con valores atípicos, o *outliers*, que es interesante detectar. Pueden ser datos que presentan errores o, en su defecto, contener información interesante, según el contexto del problema.

Algunas de las técnicas más relevantes para el presente TFG, como el modelo *RandomForest* [8] o las redes neuronales [9], se explicarán con más detalle en el apartado 2.2.

1.1.3. Deep Learning

Los algoritmos de aprendizaje profundos son un tipo de red neuronal con una elevada complejidad debido a su estructura y parámetros, como se explica en la propuesta de Bishop [1].

El nombre de estos sistemas viene porque, al representarlos en un espacio bidimensional, como podría ser una hoja de papel, y observarlos "desde arriba", descienden mucho, y en consecuencia son "profundos". El sistema de DL más simple es el perceptrón multicapa, y dentro de esto, el más sencillo tiene una capa de entrada, dos capas ocultas intermedias y una capa de salida.

Cada una de las capas tiene un número determinado de neuronas, que se encargan de recibir impulsos (información) ponderados por pesos, recogerlos y transmitir la nueva información hacia el resto de neuronas con las que estén conectadas si se supera un determinado umbral de activación. Este esquema pretende imitar el funcionamiento de la transmisión de información a través de las neuronas del cerebro humano, y por eso a menudo se trata a las unidades lógicas elementales de programación como "neuronas" y a las conexiones que se producen entre ellas como "dendritas". Esta estructura puede apreciarse en la figura 1.1.

En las redes neuronales, los únicos valores que cambian entre iteraciones son los pesos y las entradas de cada neurona. El número de neuronas en todas las capas se mantienen constante, y esto es lo que permite entrenar el modelo, modificando los valores de pesos en base a los resultados obtenidos para tratar de alcanzar una solución óptima. La solución puede ser, sin ir más lejos, la posibilidad de que un elemento pertenezca a una clase determinada, permitiendo así usar herramientas de aprendizaje profundo para la minería de datos predictiva.

Independientemente del número de capas, que pueden ser miles, y del número total de neuronas, que pueden ser millones o miles de millones, el funcionamiento siempre es el mismo: Se va modificando gradualmente el peso de las entradas, modificando en consecuencia las salidas, hasta que se alcanza el resultado deseado [2].

Una red neuronal como la previamente mostrada es una aproximación muy burda.

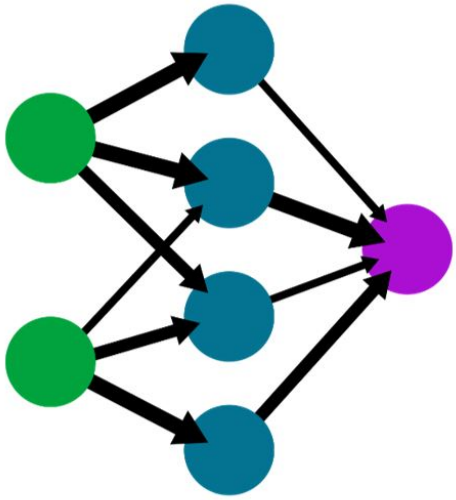

Figura 1.1: Una red neuronal muy sencilla. Fuente: Bishop (2007) [1].

Hoy día, no se utilizan tal y como se ha descrito, sino que se utilizan elementos similares que han sido reducidos a ser llamados neuronas. Además, se añade un nuevo valor, llamado *bias*, que no se recibe como entrada, sino que queda asignado de antemano a cada neurona sobre la que se vaya a aplicar el problema. Este hecho es porque los perceptrones (redes neuronales con solo una capa oculta) solo son capaces de trazar líneas rectas, y la mayoría de problemas de predicción (o minería de datos en general) no se pueden resolver con esa metodología.

Para hacer DL, se conectan muchas de estas neuronas en una red, que recibe el nombre de Red Neuronal Artificial (ANN, *Artificial Neural Network*). Estas redes neuronales son el comienzo de todo el proceso y de todos los métodos y modelos de DL que describiremos más adelante en el apartado 2.2.

Habitualmente, las redes neuronales están organizadas en capas, como previamente se describía. Lo más normal es que las neuronas de cada capa solo se comuniquen con neuronas de capas anteriores (para recibir las entradas) y de capas posteriores (para transmitir las salidas), pero nunca con neuronas de la misma capa. Esta estructura, si bien no única, es la más empleada, y la que da lugar a la definición de DL.

Por último, se mencionan algunos elementos y conceptos necesarios para entender el DL.

Tensores

Como hemos visto, cada neurona devuelve un único valor de salida, que posteriormente se combina con el de las demás hasta llegar a la capa de salida. Sin embargo, puede resultar interesante obtener un valor para la totalidad de una capa. Si las capas tienen una única neurona, esta es fácil de localizar. Sin embargo, este caso rara vez se da, y puede que en ocasiones la información que representan nuestras neuronas tenga metadatos que requieran una representación especial como, por ejemplo, valores RGB (en cuyo caso, podría convenir representar los datos en un bloque tridimensional). A este ejemplo previo se le podrían añadir elementos adicionales, como el valor alfa para la opacidad o un valor de luminosidad.

Un tensor es una estructura de datos n-dimensional que almacena los datos de forma estructurada. Cada capa genera un tensor de datos que se puede obtener y comprobar individualmente, si fuera necesario.

Funciones de activación

Cada neurona genera un resultado, pero ese resultado no siempre es el que se transmite a la siguiente capa de neuronas, o la red neuronal podría colapsar. A menudo, los resultados son sometidos a una función para convertirlos en un valor apropiado para la entrada de la siguiente capa. Estas funciones se denominan “funciones de activación”, y tienen que cumplir una propiedad fundamental: Todo valor X de la función debe tener únicamente un único valor de salida Y asociado.

El resultado a devolver puede ser cualquier número real, positivo o negativo. Los valores negativos actúan como inhibidor.

Softmax

Esta función se aplica a la capa de salida de una red neuronal, y si bien dista de ser una función de activación, su comportamiento es similar. El objetivo es convertir los valores devueltos por las neuronas de la capa de salida en probabilidades. Se emplea únicamente si hay más de una neurona en la capa de salida.

Esta función se usa porque la red neuronal puede devolver una puntuación de 14.32 para una clase, 7.42 para otra, y 9.7 para una tercera. Esto quiere decir que la primera clase es mejor que las otras dos, pero más allá de una comparación a

simple vista, es difícil manipular estos datos. *Softmax* permite convertir estos valores en probabilidades de que el ejemplo evaluado sea de una clase determinada.

Backpropagation

Este es el método fundamental por el que se entrenan las redes neuronales, ya que es el método más eficiente para ello. Todo se basa en la regla simple de que las redes neuronales aprenden minimizando los errores que cometen. Cuando una red neuronal recibe un ejemplo y evalúa su clase incorrectamente, el error cometido se puede calcular de muy diversas formas. A partir de este valor de error cometido, se modifican los valores de los pesos de todas las neuronas comenzando desde la capa de salida, y propagando los cambios hacia atrás (de ahí el nombre de *backpropagation*), para que, de cara al futuro, si un ejemplo es similar a aquel con el que se falló, se penalice la clase que ya se obtuvo, ya que se sabe que es incorrecta.

Con el tiempo suficiente, tarde o temprano la red neuronal clasificará los ejemplos **lo mejor que pueda**. Este proceso es extremadamente complejo y puede ser observado de manera completa en el estudio de I. Goodfellow [2].

1.1.4. Clasificación

La **clasificación** es una de las principales tareas dentro de la minería de datos. Como se expresó previamente, el objetivo de esta tarea es predecir una etiqueta categórica para cada ejemplo de un conjunto de datos a partir de sus atributos conocidos [10].

Existen diversos tipos de clasificación. Entre los más destacados, podemos encontrar:

- **Clasificación binaria**: en este problema concreto de clasificación, es necesario determinar a qué clase pertenece cada ejemplo de los datos, pero únicamente hay dos clases a elegir. Se suelen usar para problemas de carácter binario, como si una persona está sana o enferma o, en el contexto de esta memoria, si una conexión determinada es benigna o un ataque.

- **Clasificación multiclase**: al contrario que en el caso previo, se debe determinar la clase de cada ejemplo de entre un conjunto de clases de tamaño superior a

dos. En este TFG, también se hará uso de la clasificación multiclase para tratar de determinar el tipo de ataque de una conexión que se sabe maliciosa.

- **Clasificación multietiqueta**: un caso particular dentro de la clasificación multiclase es la clasificación multietiqueta, ya que, en lugar de asignar una sola clase a cada ejemplo del conjunto de datos, se le deben asignar un conjunto de clases en función a las características de los datos, a menudo siendo posible que cada elemento tenga un número de clases asociadas distinto.

1.2. Motivación

En diversos ámbitos de la informática, históricamente se ha dejado la seguridad en un relativo segundo plano, centrándose los esfuerzos habitualmente en alcanzar una mayor eficiencia de los programas y herramientas a desarrollar sin prestar tanta atención a cómo de seguros son. Un ejemplo más de esto es en el uso de herramientas de DL, que en la actualidad se emplean con mucha asiduidad en muy diversos ámbitos de la informática, como la minería de datos o el procesamiento del lenguaje natural.

Cabría esperar que, tras desarrollarse estas tecnologías tan ampliamente, su uso se hubiera extendido más allá de estos campos y empezara a usarse en ámbitos como el de la ciberseguridad. Sin embargo, el uso de DL en este campo es aún muy limitado, siendo los primeros escritos que tratan al respecto de alrededor de los años 2010 y 2015 [11], [12]. Adicionalmente, si bien desde ese entonces se han realizado diversos estudios de las posibles aplicaciones del DL para la seguridad de los sistemas informáticos [13], se carece de un conjunto de *datasets* preestablecido para probar su utilidad, así como métodos predeterminados que se puedan aplicar de forma general para estos experimentos.

El DL contiene, sin embargo, un conjunto de algoritmos que lo convierten en un paradigma muy poderoso que puede resultar muy útil a la hora de diseñar e implementar determinados sistemas, como podrían ser los sistemas de detección de intrusiones, por lo que investigar su uso y experimentar para comprobar hasta qué punto puede ser empleado puede resultar útil para asegurar conexiones y accesos seguros en el mundo actual, donde los ciberataques de muy diversa índole son cada vez más comunes. Conforme la complejidad de los ciberataques avanza, también habrán de desarrollarse sistemas cada vez más complejos que sean capaces de detectar cuándo se está produciendo un ataque, el tipo y la causa, de manera que el problema pueda ser resuelto antes de generar un estado peligroso en el sistema.

1.3. Objetivos

En esta sección se expondrán los objetivos que se pretenden cumplir al concluir el desarrollo de este trabajo.

1.3.1. Objetivo general

Uno de los principales objetivos del presente TFG es emplear diversos métodos de minería de datos, múltiples de ellos apoyados en el DL, con el propósito de determinar primero si una conexión determinada a una red es tráfico benigno o maligno a partir de los datos que se pueden extraer de ella y, posteriormente, si se determina que la conexión es un ataque, saber identificar de qué tipo es el ataque. Adicionalmente, otro objetivo fundamental es la implementación de un modelo capaz de predecir si una conexión es o no benigna y, en caso de no serlo, qué tipo de ataque representa.

Idealmente, se busca que este modelo a generar sea capaz de detectar ataques incluso si son *0-day*. Considerando que se pretende usar la información del flujo de datos como base sobre la que tratar de predecir los ataques y sus tipos, mientras los *0-day* se subordinen a uno de los tipos de ataques que hemos estudiado - esto es, provoquen fluctuaciones en el flujo normal de datos lo suficientemente notables como para que el modelo las detecte y clasifique como anómalas - este debería ser capaz de predecir como mínimo que el tráfico correspondiente forma parte de un ataque, incluso si posteriormente no se puede predecir correctamente el tipo de ataque (mayormente porque es desconocido en caso de ser un *0-day*).

De nuevo, se recalca la importancia de la presencia de un administrador humano regulando todo el proceso previo, y particularmente de cara al último caso propuesto. Si se detecta un ataque desconocido y que no se puede encasillar bajo ningún otro tipo de ataque debido a sus particularidades, es necesario actualizar el modelo con este nuevo tipo de ataque, realizando un nuevo entrenamiento, con el objetivo de que, si se dan más ataques del mismo tipo, puedan ser detectados y controlados de la forma más adecuada posible.

1.3.2. Objetivos específicos

Los objetivos específicos surgen como resultado de desglosar el objetivo general expuesto en el apartado previo y se presentan a continuación.

- **Lograr una visión general de los campos de minería de datos**. Con este propósito, es necesario llevar a cabo un estudio sobre los procesos asociados al proceso de KDD, ya descritos en el apartado 1.1.2. Este conocimiento es necesario para el presente TFG por ser necesario para llevar a cabo un procesamiento correcto de la información a tratar.
- **Lograr una visión general sobre el aprendizaje profundo**. Las técnicas de aprendizaje profundo son prometedoras para problemas de gran complejidad, incluyendo la clasificación, por lo que el conocer el contexto de estas herramientas y su funcionamiento es vital.
- **Conocer el funcionamiento de los modelos de clasificación más empleados**. Los modelos descritos en el apartado 2.2 son los más habituales para problemas de clasificación, y muchos de ellos adicionalmente se apoyan en el DL. El conocimiento de estos modelos y su funcionamiento es esencial para el presente trabajo.
- **Revisión del estado del arte en el ámbito de la ciberseguridad**, y más particularmente, del uso de modelos de DL con objetivo de detectar ataques, intrusiones y otras brechas de seguridad de distinta índole. Esta revisión aparece en el apartado 2.3.
- **Estudio y análisis de conjuntos de datos** relacionados con el ámbito de la ciberseguridad y más concretamente con intrusiones en la red. En esto se incluye aplicar un preprocesamiento y limpieza de los datos con objetivo de facilitar la extracción de conocimiento posterior. Este objetivo queda desarrollado en el apartado 3.2.
- **Implementación de los modelos previos** con objetivo de poder aplicarlos en problemas de clasificación reales del ámbito de la detección de intrusiones y particularmente de tráfico sospechoso en la red. El proceso de implementación de los modelos y código asociado puede encontrarse en el apartado 3.3.
- **Experimentación con los modelos para diferenciar tráfico benigno de maligno**. Uno de los dos objetivos principales del presente TFG es ser capaz de discernir si una conexión cualquiera a una red determinada es tráfico benigno o, en su defecto, es un ataque de algún tipo. Para este objetivo concreto, basta con poder determinar si una conexión dada es benigna o maliciosa, independientemente del tipo de ataque que pudiera ser. La experimentación se desarrolla en el apartado 3.4.
- **Experimentación con los modelos para diferenciar entre tipos de tráfico maligno**. A partir de la clasificación realizada de acuerdo con el objetivo previo,

el segundo objetivo principal del TFG consiste en poder diferenciar un ataque de todos los demás en base a la información de las conexiones. Si una conexión es detectada como maligna, el objetivo consiste en lograr determinar el tipo de ataque de esa conexión. Al igual que en el caso previo, estos conceptos se ven en mayor detalle en el apartado 3.4.

- **Realizar un análisis de los resultados** para determinar qué modelo o *ensemble* de modelos, sería el más adecuado a la hora de predecir, dada la información de una conexión determinada, si es tráfico benigno o malicioso y, en caso de ser malicioso, el tipo de ataque con el que se corresponde. Este análisis se da en el apartado 4.2.

1.4. Presupuesto del proyecto

En este apartado, se mostrará el presupuesto del proyecto en base al tiempo y recursos empleados. Este presupuesto se puede observar en la tabla 1.1. Para el coste del trabajador, se ha consultado el salario promedio de los ingenieros en informática en la comunidad autónoma de Andalucía, y para el precio del equipo se ha considerado el precio real del equipo empleado para las pruebas.

	Descripción	**Coste (€ / mes)**	**Valor total**
Trabajador	Ingeniero en informática con los conocimientos necesarios para la implementación de los modelos de Deep Learning y análisis de los resultados para identificar el más adecuado.	2.000	16.000
Equipo	Equipo informático con las características adecuadas para el proceso de entrenamiento y evaluación de los modelos. Los detalles del equipo usado se pueden ver en la tabla 3.6.	900 (pago único)	900
		Coste total:	16.900€

Tabla. 1.1: Presupuesto del proyecto.

1.5. Cronograma del proyecto

En este apartado, se muestra un cronograma del proyecto, representado en el diagrama de Gantt mostrado en la figura 1.2.

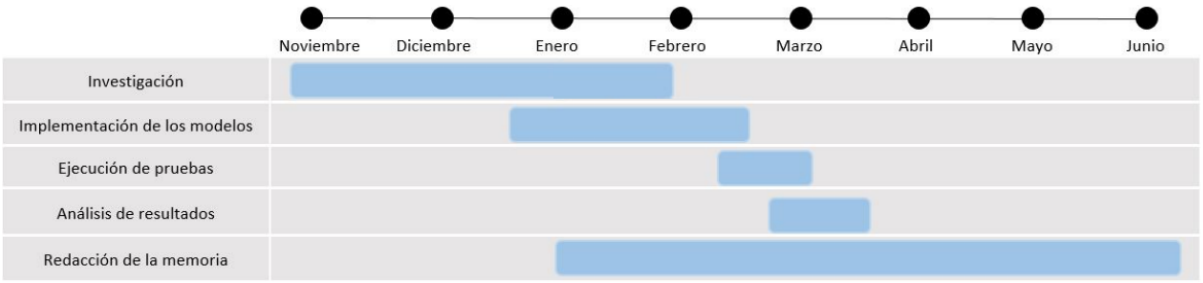

Figura 1.2: Diagrama de Gantt del proyecto. Fuente: Elaboración propia.

1.6. Estructura de la memoria

La memoria final de este proyecto está organizada en una serie de capítulos que se ajustan a la siguiente estructura:

- Capítulo 2: Capítulo en que se discute el estado del arte de sistemas de ciberseguridad que aplican DL, así como diversas líneas de investigación y desarrollo al respecto.
- Capítulo 3: Capítulo en que se detallan los conjuntos de datos empleados para el desarrollo del trabajo y el procesamiento de los mismos, así como las metodologías empleadas para realizar el análisis de esos datos.
- Capítulo 4: En este capítulo, se muestran los resultados del análisis de datos del capítulo previo, y se detallan los mismos. También se realiza una comparación de los resultados obtenidos con respecto a los del estado del arte antes descrito.
- Capítulo 5: Último capítulo de la memoria en que se realiza una recopilación de todo el trabajo desarrollado, así como los resultados obtenidos y las conclusiones del estudio.

Capítulo 2

ANTECEDENTES

2.1. Introducción

Históricamente, la seguridad siempre ha sido uno de los campos que más dejado de lado ha quedado. Incluso antes de la informática, en otras ramas ingenieriles e industriales se priorizaba el desarrollo y la eficiencia antes que la seguridad, lo que ha provocado que esta haya ido avanzando de forma comparativamente lenta.

Un ejemplo claro de esto es en la aplicación de la IA. Incluso antes del reciente estallido en la popularidad de las herramientas que la emplean, se había dado un crecimiento constante del interés en ella en los últimos años, que ha culminado en la situación actual, donde podemos ver programas que emplean modelos de *Machine Learning* en prácticamente todas las ramas, incluyendo la ciencia de datos, pero también en generación de imágenes y LLM.

Podría esperarse que la impresionante capacidad predictiva de la IA podría llegar a ser muy útil para determinadas tareas, como la ciberseguridad. Sin necesidad de tener una base de datos constantemente actualizada, como es el caso de muchos SDI antiguos, y centrándose más en el flujo de información, se podría desarrollar un sistema considerablemente más eficaz contra ataques *0-Day*. Sin embargo, ha quedado demostrado en diversas pruebas (que pueden verse en el estudio de Wu et al. [14], por ejemplo) que lo más seguro es que el desarrollo muestre un enfoque *Human-in-the-loop*, de manera que en primera instancia la IA se encargue de filtrar los datos y un experto en seguridad se encargue de determinar si realmente se está produciendo un ataque a partir de la información extraída por la IA en un paso previo.

Sin embargo, pese a todas las ventajas que ofrece la IA en el ámbito de la seguri-

dad, hasta muy recientemente no ha empezado a haber progresos en ese campo [13]. Apenas se han realizado estudios que relacionen estos dos campos del conocimiento, siendo el más antiguo de ellos de 2013 [11], describiendo brevemente cómo realizar la monitorización de un flujo de datos continuo como los que podrían darse en la red. Desde entonces, se ha continuado investigando en diversas direcciones y con distintos propósitos. A continuación, primero se explicarán varios modelos de clasificación que pueden ser empleados para la detección de ataques en la sección 2.2. Posteriormente, se analizará en detalle el estado actual de la investigación de ciberseguridad en relación con la IA, así como los mayores problemas encontrados durante los mismos en la sección 2.3.

2.2. Modelos de clasificación

Para resolver el problema de clasificación, es pertinente usar diversos modelos y comparar los resultados para determinar cuál es el más adecuado. Con este fin, comprobaremos diversos modelos de DL, así como el modelo *RandomForest* [8], que, si bien no es DL, ha demostrado tener resultados muy prometedores a la hora de trabajar con *datasets* tabulares como los usados en el presente trabajo.

2.2.1. RandomForest

El modelo *RandomForest* [8] consiste en un modelo de *ensemble* (esto es, un modelo que combina diversos modelos más simples) que utiliza una gran cantidad de modelos de tipo árbol, como J48 [15], para obtener un resultado final. En la figura 2.1 se representa el esquema de funcionamiento de este algoritmo.

Lo que diferencia a *RandomForest* del resto de algoritmos de *ensemble* es el hecho de que, para cada árbol, se eligen una serie de características del conjunto de datos en lugar del conjunto completo de características. Posteriormente, a partir de los resultados proporcionados por los árboles individuales, se puede determinar qué etiqueta o clase es la más apropiada para cada dato.

Dado que cada árbol considera subconjuntos de atributos distintos y luego la información se pone en común, habitualmente mediante una votación, la robustez del algoritmo mejora considerablemente con respecto a los algoritmos de tipo árbol ejecutados individualmente. A su vez, también permite mitigar parcialmente el sobreajuste y los problemas asociados a un alto número de atributos.

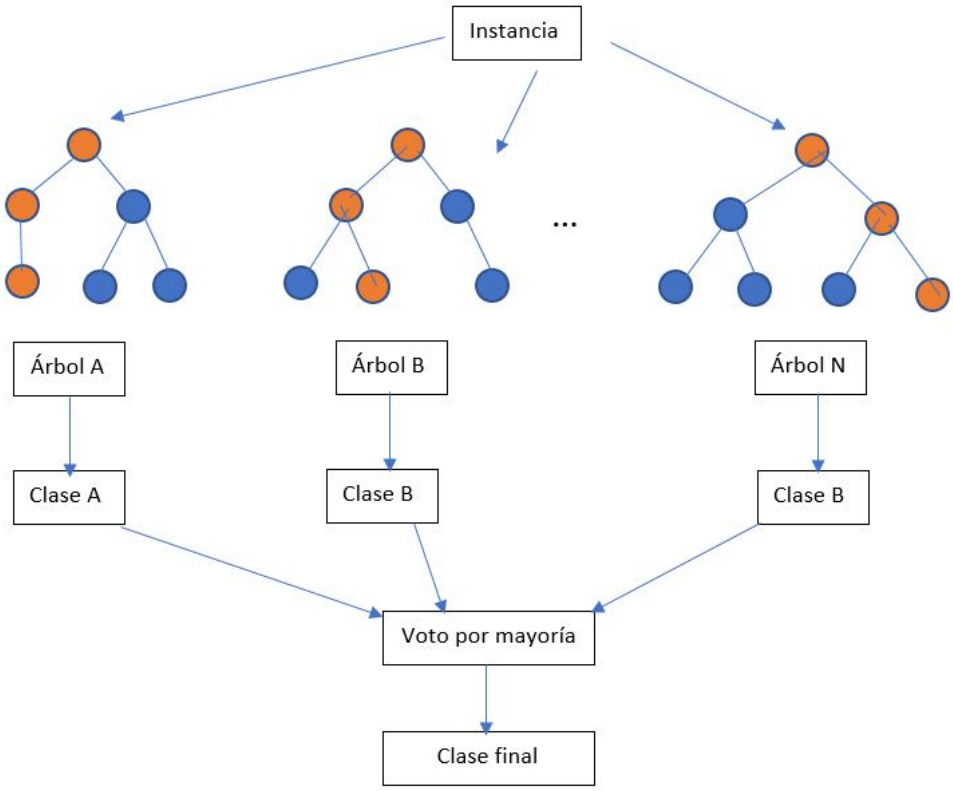

Figura 2.1: Esquema de un *RandomForest*. Fuente: Elaboración propia.

2.2.2. Perceptrón multicapa

El perceptrón multicapa es el sistema de DL más sencillo, como ya se mencionaba en el apartado 1.1.3.

La estructura básica de un perceptrón (una capa de entrada, una capa intermedia y una capa de salida) a menudo es insuficiente para resolver problemas de predicción particularmente complejos, puesto que las formas que pueden trazar (y en consecuencia separar los datos en clases) están limitadas a líneas rectas, por lo que es común añadir más capas intermedias entre la inicial y la final, dando lugar al perceptrón multicapa.

Pese a su simpleza, el perceptrón multicapa es un método de clasificación empleado a menudo ya que, dentro de lo relacionado con el DL, es uno de los modelos más sencillos de implementar y su coste computacional es relativamente bajo. Añadiendo un número de capas intermedias más elevado, de acuerdo a lo previamente descrito, se pueden lograr grados de acierto relativamente elevados pese a la simpleza del sistema, y mediante la parametrización de otros factores, como el LR, se pueden ajustar y afinar los resultados del modelo todo lo posible.

Sin embargo, precisamente debido a la simpleza de estos modelos, su capacidad de predicción sigue siendo relativamente limitada en comparación con modelos más avanzados como los que se verán más adelante.

2.2.3. Red neuronal convolucional

La **convolución** (del inglés *convolution*) es una de las estrategias de DL más empleadas, y se ha convertido en la principal para la clasificación, manipulación y generación de datos multidimensionales, como imágenes o procesamiento del lenguaje natural [16]. La convolución es fácil de emplear en DL porque todo su proceso se puede englobar en una sola capa de soporte, que en este caso recibirá el nombre de capa convolucional.

Utilizando imágenes, el ejemplo más sencillo es reconocer un color determinado. Cada píxel de la imagen es un valor de entrada, de manera que una imagen puede ser reducida a un tensor tridimensional con coordenadas {x, y} y un número determinado de valores por píxel (que puede ser uno solo si la imagen está en blanco y negro, o tres, si usa el modelo RGB, etc). Cuando un tensor atraviesa una capa convolucional, es habitual que cambie en alto, ancho y número de valores. Esto provoca que resulte complicado seguir definiendo un píxel como tal, ya que puede que tras atravesar la capa convolucional, las dimensiones de los tensores varíen, y la información asociada a un píxel pueda alcanzar cardinalidades arbitrarias en unas coordenadas dadas.

El resultado final de ejecutar una red neuronal convolucional se obtiene tras aplicar uno o más filtros u operadores sobre los datos de entrada. Para aplicar estos filtros, el proceso a seguir es el siguiente, tal y como se puede observar en la figura 2.2.

- La capa de convolución se encargaría de recoger los datos de la capa de entrada. Siguiendo con el ejemplo de una imagen previamente propuesto, esta capa obtendría el valor RGB correspondiente a cada píxel de entrada.
- Posteriormente, a través de una función matemática u operador que recibiese como entradas los valores de cada tensor inicial, se obtendría un valor determinado para cada combinación. Siguiendo con el ejemplo de la imagen, unificaría los valores RGB en una escala de 0 a 1, correspondiente a un tono de blanco, negro o gris determinado.
- Los pesos empleados para determinar la función exacta que se emplea son comunes a toda la capa y, en consecuencia, se aplican igual sobre todos los datos en una iteración dada del algoritmo.

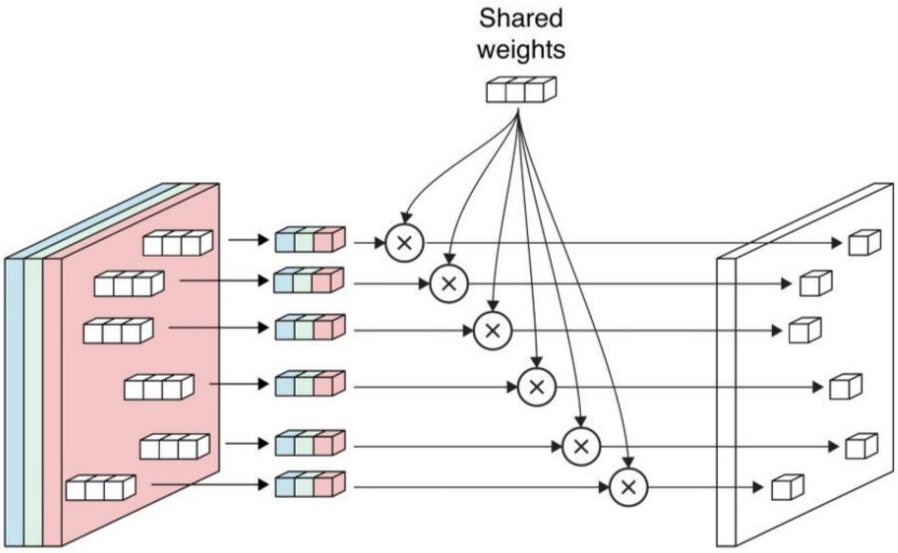

Figura 2.2: Convolución en tres capas RGB. Fuente: Bishop (2007) [1].

- La transformación se aplica sobre todos los datos, obteniendo un resultado determinado que se corresponde a una combinación mediante alguna función matemática de los que se recibieron como entrada. Recuperando el ejemplo de la imagen, a partir de los tres valores RGB se obtendría un valor de gris apropiado aplicando el operador correspondiente sobre ellos.

Existen filtros mucho más complejos que permiten emborronar imágenes, detectar bordes en horizontal o vertical, o una combinación de todas ellas.

A partir de estos filtros, se pueden obtener sistemas capaces de identificar determinados objetos. Mientras se parta de un conjunto de datos apropiado para el entrenamiento, que permita definir los datos de los objetos que se pretende identificar como un conjunto de píxeles, se pueden aplicar esos conjuntos de píxeles como un filtro sobre los nuevos datos, de forma que, si coinciden, se pueden clasificar correctamente nuevas entradas, aunque sean relativamente distintas a los datos que se poseen.

2.2.4. Red neuronal recurrente

A menudo, se trabaja con un conjunto de elementos en el que el orden de los elementos importa, esto es, una secuencia de elementos. Hay muchos algoritmos que trabajan con secuencias y son capaces de crear nuevas secuencias o continuar las ya existentes en mayor o menor medida.

Las redes neuronales recurrentes (*Recurring Neural Network*, RNN) son capaces

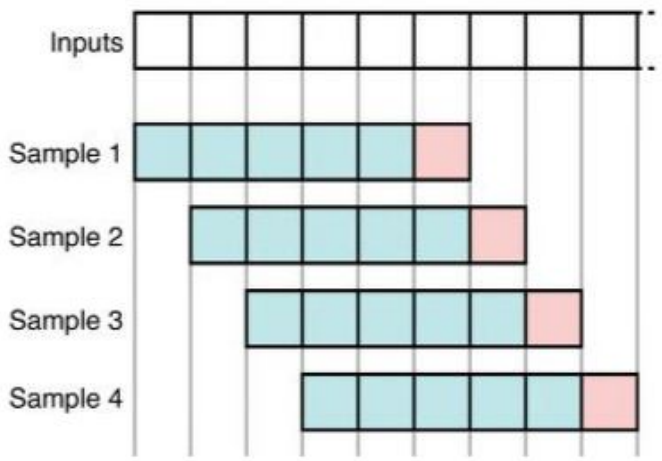

Figura 2.3: *Sliding window* para predecir el elemento rojo. Fuente: Bishop (2007) [1].

de memorizar información de cada elemento de la secuencia (*token*) conforme lo reciben, y aprovechar esta información de cara al resto de *tokens* que se reciben en el futuro para proporcionar información contextual adicional. Este modelo fue planteado originalmente por Rumelhart y McClelland [9].

Este tipo de redes neuronales se utilizan muy a menudo en problemas de LLM. Sin embargo, lo más habitual es convertir el lenguaje en números, ya que el procesamiento de los mismos es más rápido.

Habiendo reducido el problema del lenguaje a un problema de números, el funcionamiento es análogo al de las redes neuronales ya mostradas previamente, solo que, en cada paso del entrenamiento, el valor de "tiempo" asociado incrementa en x, de manera que, en cada paso, el primer elemento considerado es mayor que en el anterior. Tomando un x = 1, el primer elemento del segundo paso sería el segundo del anterior, el primero del tercero sería el tercero del primero... etc. Este esquema, denominado *sliding window*, puede verse en la figura 2.3.

En cada una de estas muestras, lo que se pretende es predecir el valor del último elemento en base no solo a los de la muestra, sino también a lo que ya se sabe del resto de muestras previas.

Para predicciones o secuencias numéricas, estas redes dan lugar a resultados muy favorables. Si se pretende predecir algo de mayor complejidad (como pedir a la red que escriba una novela, por ejemplo), hace falta un poder computacional muy grande (con múltiples capas y numerosas neuronas por capa), y fallaría igualmente, ya que solo trabaja con palabras individuales, y no con su semántica, por lo que escribir algo coherente resultaría muy difícil, dado que el sistema puede devolver con variaciones

numéricas muy pequeñas palabras que no tienen correlación alguna.

Para resolver todos estos problemas, las RNN introducen el concepto de **estado**. El estado es una descripción del sistema para un instante de tiempo determinado, así como información adicional que pueda resultar relevante.

El mayor cambio se basa en que la salida de cada capa depende no solo de las entradas y los pesos, sino también del estado, de manera que una misma capa con exactamente las mismas entradas puede producir salidas distintas si el estado en que recibe la entrada es distinto.

Por último, hay un cambio adicional: el orden en que se reciben las entradas influye también en la salida, ya que se supone que cada entrada es un elemento con información temporal asociada. Si el orden de las entradas no tiene efecto ninguno sobre la salida, usar una RNN no suele ser la solución ideal para el problema.

Para entrenar una RNN, hace falta un tipo especial de *backpropagation*, denominada **backpropagation through time**, que permite que los ajustes se muevan a lo largo de toda la red neuronal de forma proporcional. Esto a su vez genera el problema de que los gradientes cada vez se hacen más pequeños conforme ascienden a través de la red neuronal, hasta el punto de ser casi minúsculo y enlentecer el proceso de aprendizaje [17].

Para corregir este problema, se aplica otro método, *long short-term memory* [18], que se basa en que el estado interno del sistema cambia tan a menudo que se puede considerar una memoria a corto plazo, pero podemos querer guardar determinada información del estado a largo plazo. Esto permite ajustar los pesos de las celdas también en base al estado de la red en que se encuentre, permitiendo así el entrenamiento. De esta manera, si la memoria olvida una red, lo que está haciendo es manipular sus valores para que sean cercanos a 0, mientras que recordarla consiste en aproximar sus valores a 1. En la figura 2.4, se muestra la estructura de una RNN con LSTM.

Esta estructura se puede replicar varias veces de forma paralela para permitir mayor capacidad predictiva a costa de un mayor coste computacional.

2.2.5. Transformers

Las RNN han demostrado tener una capacidad predictiva bastante elevada. Además, permiten incorporar al aprendizaje la información adicional del estado del sistema

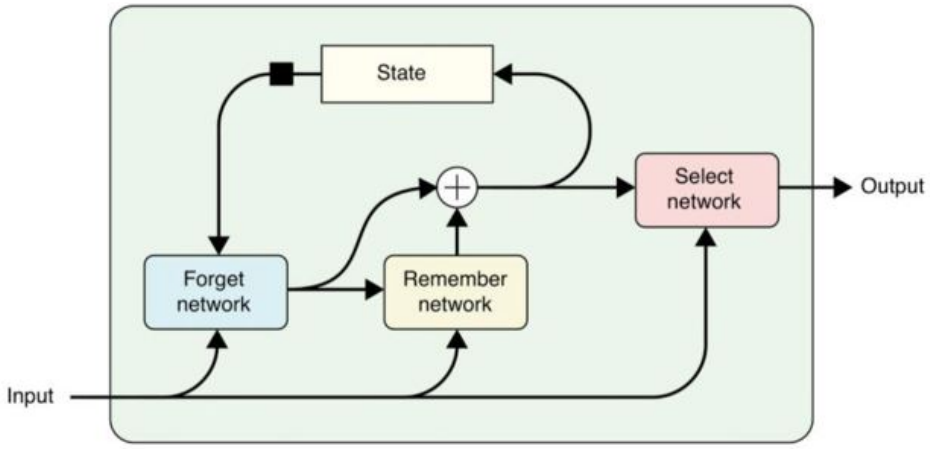

Figura 2.4: Estructura LSTM. Fuente: Bishop (2007) [1].

y, gracias a modificaciones como la *backwards propagation through time* y las LSTM, son capaces de ser entrenadas con relativa eficiencia y eficacia.

El problema que presentan, sin embargo, es que toda la información que se encarga del entrenamiento y que determina la salida de una neurona concreta se almacena en el estado del sistema, por lo que el gasto de espacio es muy amplio [2].

Además, debido al entrenamiento aprendiendo y olvidando y al funcionamiento de los gradientes, el tiempo necesario para tener la red completamente entrenada es muy grande. La solución para estos problemas es una red de atención, que es una pequeña red que puede ser entrenada y utilizada simultáneamente. Varias de estas redes pueden ser combinadas en estructuras más grandes llamadas *transformers* que pueden servir como modelos de lenguaje capaces de tareas complejas como la traducción o la generación de texto.

La idea clave detrás de esta metodología es que las palabras ya no se representan como un mero número, sino como un conjunto de información que puede ser comparada de muchas otras formas. Como metáfora, se puede colocar cada palabra en un gráfico bidimensional. No importa qué es cada eje, pero sí se sabe que las palabras con más correlación entre sí están más cercanas. A su vez, tomando como origen las coordenadas x=0, y=0, se pueden hacer operaciones matemáticas con palabras en base a lógica vectorial.

Trasladando eso a un espacio n-dimensional, es necesario colocar cada palabra en las coordenadas que le correspondan. A este proceso se le llama *embedding*. Por ejemplo, si se entrena al sistema con palabras, y se dice “Hoy bebí”, la palabra inmediatamente posterior se reconoce como una bebida, ya que la combinación de palabras previa siempre suele ir seguida de bebidas, y se coloca “cerca”, en el espacio

n-dimensional, de otros elementos que hayan sido catalogados como bebidas.

Este método presenta problemas con palabras que tienen más de un significado, puesto que el sistema no es capaz de distinguir si la palabra “banco” es un lugar donde sentarse o un conjunto de peces.

Esto genera la necesidad de tener que crear *embeddings* para cada uno de los significados, y hay que elegir el correcto cuando la palabra se analiza. Para esto se utiliza ELMo.

ELMo

Embedding from Language Models (ELMo) presenta una estructura similar a dos redes neuronales, una de las cuales tiene propagación hacia delante, mientras que la restante tiene propagación hacia atrás. Estas redes trabajan en tándem, dando lugar a algo similar a una LSTM bidireccional. De esta manera, cada palabra se convierte en dos tensores: uno que considera las palabras previas y otro que considera las palabras posteriores. Agregando estos dos tensores, se obtiene un significado contextualizado de la palabra en base al resto de palabras de la oración.

Combinando este contexto con el hecho de que los *transformers* almacenan la posición de cada palabra como parte de la información asociada a la palabra y el *embedding*, se permite así predicciones o tareas mucho más eficientemente que con RNN, que además tienen la ventaja de poder ser entrenados en paralelo y obtener resultados a partir de entradas “enmascarando” las palabras que se quieren predecir. Este es el funcionamiento de *transformers* muy conocidos, como BERT (y sus derivados) y GPT-2.

2.3. Estado del arte

Como se puede ver en varios estudios [13], existe un claro avance en la combinación de las herramientas de IA con la ciberseguridad en los últimos años. Con el propósito de clasificar los avances y simplificar las explicaciones pertinentes, se distinguen cuatro ramas de estudio, cada una de ellas enfocada en enfrentarse a un determinado tipo de problema de ciberseguridad. A continuación, explicaremos brevemente las cuatro ramas, antes de centrarnos en la última de ellas de cara al desarrollo del TFG.

2.3.1. Identificación de usuarios

La primera de estas ramas es la **identificación de usuarios**. Esta rama se encarga de desarrollar nuevos métodos para identificar maniobras de camuflaje o usuarios haciéndose pasar por otros, pero también busca la detección de objetos ilegales y potencialmente peligrosos. Estas investigaciones incluyen formas de asegurar que la persona que está iniciando sesión es realmente quien dice ser. Por ejemplo, se puede emplear una red neuronal para detectar no solo si la contraseña introducida por un usuario es la correcta, sino también para analizar si el tiempo invertido por el usuario es el habitual o no con objeto de evitar que un tercero pueda lograr el acceso si se hace con la contraseña [19]. Otra rama interesante es la multiautenticación [20], basada en mantener un flujo continuo de autenticación de usuario en lugar de tener que realizarla una única vez. Con este propósito, se empleaba un algoritmo *Random-Forest* para determinar si todas las autenticaciones eran válidas y predecir al usuario correctamente.

Por último, en esta rama también se estudian diversos medios de biometría en los que se destaca el uso de histogramas de gradientes orientados y *Extreme Machine Learning* para análisis de huellas [21], así como el uso generalizado de redes neuronales recurrentes para el análisis de voz [22].

2.3.2. Situación de la red

La segunda rama en la que está enfocada la investigación es el análisis de la **situación de la red**. En este caso, lo que se pretende es tener información sobre el flujo de datos completo al que está sometido un servicio web o una red cualquiera y extraer de esa información parámetros clave que permitan definir una serie de estados. A partir de estos estados, se pueden predecir flujos de datos irregulares en el mismo momento en que empiezan a darse, permitiendo ajustar la red para evitar que un potencial atacante logre acceso a datos restringidos, fuerce un ataque de denegación de servicio, o similar.

Para poder realizar este tipo de predicciones, es necesario contar con una base de conocimiento inicial muy amplia que permita, simultáneamente, detectar la información necesaria para asignar un estado de la red en cada momento, extraer los patrones pertinentes para saber cómo actuar si se da un cambio de estado y proporcionar resultados fiables.

Algunos de los ejemplos de modelos diseñados con esta idea en mente incluyen las

MEBN. Estas redes presentan una gran capacidad de determinar la situación de la red en cada momento [23] y, en consecuencia, de extraer el estado correspondiente. Sin embargo, en algunos problemas de especial complejidad se volvía a traer de vuelta el concepto de *Human-in-the-loop*, de manera que el humano pudiera intervenir a partir de la información extraída por el sistema. Otro estudio también empleaba redes bayesianas, pero con un enfoque difuso [12]. Esta metodología es capaz de obtener siempre buenos resultados a la hora de detectar el estado de los sistemas, pero suelen requerir de la ayuda del ser humano en lo que la toma de decisiones respecta.

Otros enfoques emplean en su lugar metodologías como *RandomForest* [24], de manera que cada árbol extrae datos distintos de la situación de la red, y el estado final en el que la red se encuentra se consigue al combinar los resultados de cada uno de los árboles mediante un proceso de agregación, haciendo que el sistema fuera más objetivo a la hora de determinar la situación de la red. Entre otros modelos también cabe destacar el uso de WNN [25], así como la adaptación de todos estos sistemas previos a entornos *Big Data* actuales [26].

2.3.3. Detección de tráfico atípico

Otra de las ramas a estudiar es la **detección de tráfico atípico**. Las redes, a menudo, tienen una capacidad de transportar datos limitada. Si el umbral de lo que la red puede transportar es superado, a menudo por acción de la inyección artificial de datos por parte de un atacante, los nodos y enlaces de la red se vuelven incapaces de proporcionar los servicios esperados para los usuarios, y pueden darse casos en los que se llegue a producir una pérdida de información por desbordamiento. La investigación relacionada a esta rama busca el ser capaz de detectar estos posibles ataques, conocidos como ataques de denegación de servicio (DoS), para evitarlos y en consecuencia reducir el daño que puedan ser capaces de causar.

Los esfuerzos de esta rama pueden ser divididos a su vez en cuatro subapartados [27], en base al funcionamiento de los métodos de detección empleados: clasificación, métodos estadísticos, *clustering* y teoría de información. Si bien esta clasificación servía como punto inicial a partir del que desarrollar la actividad investigadora, rápidamente se comprobó que era demasiado estricta, y los modelos posteriores, de mayor complejidad, empezaron a combinar varios de los subapartados de investigación desarrollados para obtener mejores resultados, aprovechando mejor la información.

En los años posteriores, surgirían varias propuestas adicionales de modelos para la detección de tráfico atípico. La mayoría de ellos aplicaban ANN o SVM. Otros ejemplos

emplean una CNN para la detección de tráfico atípico en entornos de *cloud computing* [28] o una SVM para clasificar y evaluar múltiples tipos de ataques de tráfico, y también se plantea la posibilidad de paralelizar los cálculos para acelerar el proceso [29].

2.3.4. Monitorización de comportamientos peligrosos

La última rama que se va a presentar es la **monitorización de comportamientos peligrosos**. Conforme las herramientas informáticas y de manejo de información, como el *cloud computing* o el manejo de *big data*, se van desarrollando, es natural que tarde o temprano los atacantes desarrollen nuevos métodos ofensivos.

Con el rápido crecimiento del uso de Internet, así como de la cantidad de datos que en él se emplea, resulta evidente que los SDI de antaño ahora carecen de las capacidades como para enfrentarse a potenciales ataques más modernos, sea bien porque siendo ataques *0-day* no están aún en ninguna base de datos y, en consecuencia, son indetectables para aquellos sistemas que dependen de las susodichas, o bien porque el rápido flujo puede permitir infiltraciones de datos que escapen el alcance de los sistemas de defensa actuales.

El objetivo de esta línea de investigación, que será en la que se centrará el presente TFG, es la detección de ataques a un sistema determinado, y, más concretamente, la defensa de lo que se conoce como "puntos letales", esto es, puntos vulnerables que son más sencillos de ser explotados. De esta manera, aplicando las medidas correspondientes, se pueden evitar situaciones de tratamiento de emergencia - momentos en los cuales el problema aún puede ser solucionado, pero en la que el coste para resolverlo incrementa considerablemente [13].

Los esfuerzos en este campo se basan, pues, en la mejoría de los sistemas de detección de intrusiones clásicos con objetivo de mejorar su escalabilidad y, en consecuencia, su eficacia. Debido también a la gran variedad de ataques que se pueden dar, se han desarrollado dos formas de investigar: modelos que apoyan a la detección de intrusiones de forma general, para cualquier tipo de ataque, y otros que están centrados en un tipo específico de ataque, que ofrecen mejores resultados para ese ataque concreto, pero no para el resto. A continuación, se expondrán una serie de ejemplos de ambos casos, antes de profundizar en algunos de ellos.

Modelos generales

Comenzaremos por modelos de detección de ataques en general y, más concretamente, por una propuesta que combina una extracción de patrones basada en DL con un SVM multicapa integrado, empleando adicionalmente vez un DBN para realizar una reducción de la escala de conjuntos de datos de tráfico web y poder localizar en consecuencia los comportamientos anómalos [30]. Otro ejemplo emplea un sistema de ANN optimizado mediante hiperparámetros capaz de detectar ataques de *bots*, y cuya principal ventaja es el hecho de que podía ser desplegado en varias máquinas simultáneamente [31]. Otra propuesta, enfocada a entornos de *cloud computing*, es un modelo híbrido de detección de intrusiones que emplea un algoritmo de *clustering* con *K-means* y un algoritmo de clasificación empleando una máquina de soporte vectorial [32]. Por último, en esta rama de investigación, podemos destacar también una propuesta que utiliza un sistema de detección de intrusiones basado en un hipervisor que utiliza una red neuronal basada en lógica difusa. En sistemas en los que la frecuencia de ataque era baja, podía detectar adecuadamente los ataques con los conjuntos de datos empleados durante las pruebas [33].

Modelos para ataques específicos

A continuación, trataremos modelos que se han centrado en prever un comportamiento anómalo concreto. Fundamentalmente, estos modelos se han hecho de cara a la predicción del tipo de ataque más común en los tiempos actuales, los ataques de denegación de servicio distribuido (DDoS), si bien también cabe destacar algún modelo individual que está centrado en la defensa de sistemas de 5G, dado que ahora mismo están en proceso de desarrollo. A continuación, mostraremos algunos ejemplos de ambos:

- En lo que los ataques DDoS respecta, existen bastantes modelos que ofrecen buenos resultados. Cabe destacar sin embargo una propuesta que presenta un *framework* completo para la detección de ataques DDoS que demostró tener buenos resultados [34]. Empleaba *K-means* para realizar *clustering* de los comportamientos detectados y, posteriormente, un SVM para realizar la clasificación. Otras propuestas emplean una serie de ANNs y RNNs con capas completamente interconectadas entre sí [35] o dividen el modelo en cinco elementos bien diferenciados: recolector de datos, Hadoop-HPFS, conversor de formato, dispositivo de procesamiento de datos y módulo de detección. Este sistema era capaz de

responder de forma eficiente y eficaz ante los diversos ataques que se utilizaron durante las pruebas [36].

- En la detección de intrusiones en 5G, la principal propuesta utiliza dos capas de modelos de DL [37]. La primera de ellas utilizaba métodos de agregación de flujos de la red para detectar rápidamente signos de un ataque, mientras que la segunda conectaba la situación del sistema con datos históricos del mismo empleando una LSTM. Esta segunda capa estaba a su vez vinculada al módulo de monitorización y diagnóstico de intrusiones para que actuara sobre las anomalías detectadas.

2.4. Problemas descubiertos

A continuación, se presentarán una serie de problemas de la IA que dificulta notablemente su uso en el contexto de la ciberseguridad. Estos problemas son comunes a todos los modelos previamente descritos, puesto que depende menos de la metodología concreta usada y más del paradigma general de la IA que se utiliza actualmente.

Comenzaremos mencionando que el modificar los datos mínimamente hace que la IA pueda llegar a devolver resultados completamente distintos. Modificar un solo píxel de una imagen puede modificar el resultado que proporcione una red neuronal que esté tratando de identificarla [38]. Análogamente, si se modifican algunos bits o bytes de información en un flujo de datos, se puede engañar a la IA y esta nunca es totalmente capaz de detectar de forma certera y sin errores absolutamente todos los intentos de ataque. Se ha llegado a emplear una GAN para crear ataques *malware* y determinar la capacidad de otro modelo de detectarlos, y a menudo lograron pasar desapercibidos [39].

Otro de los problemas descubiertos, y que se extiende en gran medida a todas las herramientas de IA disponibles actualmente, es la falta de transparencia en la toma de decisiones de la IA. Concretamente, la mayoría de modelos son de caja negra, por lo que, si bien se conocen las entradas y las salidas, e incluso, potencialmente, los procesos internos que pueden estar teniendo lugar, no se conoce el por qué se ajustan determinados pesos, de qué forma la red neuronal asocia determinada propiedad o característica a un ataque o no, etc. Esto provoca que el sistema pueda dar resultados buenos, pero los usuarios que lo estén empleando no puedan alcanzar el conocimiento de por qué ocurre, únicamente del hecho de que ocurre, limitando en consecuencia la capacidad del humano de aprender de los modelos y poder extraer nueva información

para mejorarlos, entre otros factores.

Adicionalmente, las pruebas siempre se ejecutan sobre conjuntos de entrenamiento y test preestablecidos que, si bien pueden dar una idea de cómo de bien o mal funcionará la herramienta ante nuevos casos, nunca son totalmente fiables. Especialmente en un campo de avance tan rápido como es la ciberseguridad (y en consecuencia el desarrollo de nuevos ataques conforme se descubren las vulnerabilidades), sería necesario realizar una cantidad elevada de pruebas para asegurar el funcionamiento, lo que resulta costoso en recursos y en tiempo.

Estrechamente relacionado con lo anterior, el último problema que se mencionará es la necesidad de una cantidad ingente de datos para poder entrenar adecuadamente estos modelos. Debido a la delicadeza de la función que los modelos de detección de intrusiones realizan, es necesario disponer de una enorme cantidad de datos para poder entrenarlos y experimentar con la mayor cantidad de casuísticas, ejemplos y ataques posibles, de manera que incluso bajo la situación de enfrentarse a nuevos ataques, hasta antes nunca vistos, los modelos sean capaces de responder adecuadamente y poder detectar los ataques como tales, asegurando así su correcto funcionamiento.

Capítulo 3

MATERIALES Y MÉTODOS

En este capítulo se describen los materiales que se han empleado de cara a la consecución de los objetivos planteados en el capítulo previo. Adicionalmente, se especificará el preprocesamiento realizado sobre cada uno de los *datasets* sobre los que se experimentará, así como la implementación de los diversos modelos de clasificación usados para predecir los tipos de tráfico previamente descritos. Por último, se detallarán los parámetros usados para cada modelo y dataset, así como las características del equipo sobre el que se hicieron las pruebas.

3.1. Framework

Con objetivo de facilitar la implementación y prueba de los modelos de ejecución, se ha utilizado el *framework* de **PyTorch**. *PyTorch* es una librería para *Machine Learning* de *Python* basada en *Torch*, su contraparte implementada para C. Si bien su utilidad es enorme y permite la implementación de muy diversos modelos, nos centraremos fundamentalmente en el perceptrón multicapa, las redes neuronales convolucionales, las redes neuronales recurrentes y los *transformers* [40].

PyTorch proporciona principalmente dos servicios de alto nivel que son muy relevantes para el presente TFG:

1. **Tensores de PyTorch**. Funcionando de manera análoga a los vectores de *Numpy* [41], los tensores de *PyTorch* permiten almacenar la información en matrices rectangulares multidimensionales de tamaño arbitrario y trabajar con ellas mediante cálculos y transformaciones. Sin embargo, ofrecen una ventaja notable sobre los

vectores de *Numpy*, y es el hecho de que permiten el uso de CUDA [42] para realizar diversas operaciones matriciales sobre la GPU en lugar de la CPU, acelerando considerablemente el proceso de entrenamiento y evaluación de los modelos que posteriormente se presentarán.

2. **Redes neuronales profundas** construidas con un sistema de diferenciación automática y que se pueden entrenar a partir de conjuntos de datos. *PyTorch* define la clase *torch.nn*, que incluye a su vez métodos para la construcción de redes neuronales por componentes, permitiendo al usuario combinar estos elementos para dar lugar a modelos complejos como los que se explicarán más adelante. Entre otros componentes, caben destacar diversos tipos de capa (lineal, recurrente, convolucional...) y funciones de activación (sigmoide, ReLU...) que permiten implementar los modelos utilizados para las pruebas.

Adicionalmente, y pese a no ser un algoritmo de Deep Learning como los previamente mencionados, también se experimentará con *RandomForest*. Dado que *PyTorch* carece de la implementación para este modelo, se utilizará en su lugar ***scikit-learn***[43], cuyos tensores son compatibles con los de *PyTorch*, facilitando así la comparación de modelos.

3.2. Conjuntos de datos

A continuación, se presentarán algunos de los *datasets* con información relevante en el estudio que se va a desarrollar. Estos *datasets* han sido empleados en investigaciones desarrolladas en el apartado 2.3, y se han elegido los cuatro más empleados. En consecuencia, esto nos permite comparar nuestros resultados con los de las pruebas previas de forma directa.

A continuación, se especificarán en detalle las características de cada uno de los *datasets* a tratar, así como el preprocesamiento de la información contenida en ellos que se ha llevado a cabo con objetivo de facilitar el proceso de entrenamiento y mejorar los resultados de la consecuente evaluación.

3.2.1. CIC-IDS2017

El primer dataset que mencionaremos será el **CIC-IDS2017** [44]. Este dataset contiene información sobre los ataques más comunes en el día de hoy, así como infor-

mación benigna para realizar aprendizaje que sea capaz de distinguir entre ataques y tráfico habitual. También contiene información adicional obtenida mediante el análisis del tráfico de datos, obtenida mediante el uso de CICFlowMeter, un programa diseñado por el CIC [45] capaz de obtener información del tráfico de la red, incluyendo flujos basados en el momento en el que se produjeron, IP de origen y destino, protocolos y ataques, todo ello representado en ficheros CSV. Los ataques contenidos incluyen fuerza bruta de SSH y FTP, así como DoS y DDoS, *Heartbleed*, ataques web, infiltraciones y el uso de *bots*. Este *dataset* está dividido en base a varios días de la semana, disponiendo cada día de diversos tipos de ataque. Hay un total de 78 campos para cada ejemplo, y a cada ejemplo se le asocia una etiqueta diferenciando entre si el tráfico es benigno o maligno y, en ese caso, el tipo de ataque.

Las variables que contempla cada ejemplo se pueden apreciar en la tabla 3.1.

Puerto de destino	Duración del flujo	Total de paquetes fwd
Total de paquetes bwd	Longitud total de paquetes fwd	Longitud total de paquetes bwd
Longitud máxima de paquetes fwd	Longitud mínima de paquetes fwd	Longitud máxima de paquetes bwd
Longitud mínima de paquetes bwd	Longitud media de paquetes fwd	Desviación estándar de la longitud de paquetes fwd
Longitud media de paquetes bwd	Desviación estándar de la longitud de paquetes bwd	Flujo en Bytes/s
Flujo en paquetes/s	Flujo IAT medio	Desviación estándar del flujo IAT
Máximo flujo IAT	Mínimo flujo IAT	Flujo IAT total fwd
Flujo IAT medio fwd	Desviación estándar del flujo IAT fwd	Flujo IAT fwd máximo
Flujo IAT fwd mínimo	Flujo IAT total bwd	Flujo IAT medio bwd
Desviación estándar del flujo IAT bwd	Flujo IAT bwd máximo	Flujo IAT bwd mínimo
Flags PSH fwd	Flags PSH bwd	Flags URG fwd
Flags URG bwd	Longitud de la cabecera fwd	Longitud de la cabecera bwd
Paquetes/s fwd	Paquetes/s bwd	Longitud mínima del paquete
Longitud máxima del paquete	Longitud media del paquete	Desviación estándar de la longitud del paquete

Varianza de la longitud del paquete	Conteo de flags FIN	Conteo de flags SYN
Conteo de flags RST	Conteo de flags PSH	Conteo de flags ACK
Conteo de flags URG	Conteo de flags CWE	Conteo de flags ECE
Ratio de bajada/subida	Tamaño medio de paquete	Tamaño medio de segmento fwd
Tamaño medio de segmento bwd	Longitud de la cabecera fwd media	Bytes/bulk promedio fwd
Packets/bulk medios fwd	Ratio de bulk fwd medio	Bytes/bulk medios bwd
Packets/bulk medios bwd	Ratio de bulk bwd medio	Subflujo de paquetes fwd
Subflujo de bytes fwd	Subflujo de paquetes bwd	Subflujo de bytes bwd
Número total de bits fwd en la ventana inicial (fwd)	Número total de bits fwd en la ventana inicial (bwd)	Conteo de paquetes fwd que tengan al menos un byte de información TCP
Tamaño mínimo de segmentos fwd	Tiempo mínimo que un segmento estuvo activo	Tiempo medio que un segmento estuvo activo
Tiempo máximo que un segmento estuvo activo	Desviación estándar de los tiempos que los segmentos estuvieron activos	Tiempo mínimo que un segmento estuvo inactivo
Tiempo medio que un segmento estuvo inactivo	Tiempo máximo que un segmento estuvo inactivo	Desviación estándar de los tiempos que los segmentos estuvieron inactivos

Tabla. 3.1: Variables del dataset CIC-IDS-2017

A continuación, se muestran ejemplos del dataset previo, tanto con tráfico benigno como con un ataque:

- 88, 773, 9, 4, 612, 2944, 306, 0, 68, 134.9333169, 1472, 0, 736, 849.8595962, 4600258.732, 16817.59379, 64.41666667, 148.698, 531, 1, 773, 96.625, 196.665, 580, 1, 675, 225, 348.901, 627, 1, 0, 0, 0, 0, 204, 104, 11642.94955, 5174.644243, 0, 1472, 254, 527.5207615, 278278.1538, 0, 0, 0, 1, 0, 0, 0, 0, 0, 273.5384615, 68, 736, 204, 0, 0, 0, 0, 0, 0, 9, 612, 4, 2944, 8192, 2053, 2, 20, 0, 0, 0, 0, 0, 0, 0, 0, BENIGN
- 22, 13652185, 22, 33, 2008, 2745, 640, 0, 91.27272727, 138.182, 976, 0, 83.181, 217.2857356, 348.1493988, 4.0286, 252818.2407, 633086.5034, 2178689, 3, 11600000, 550887.1429, 869278.0704, 2232461, 839, 13700000, 426630.4063,

782897.5727, 2178689, 3, 0, 0, 0, 0, 712, 1064, 1.611463659, 2.417195489, 0, 976, 84.875, 186.8396554, 34909.05682, 0, 0, 0, 1, 0, 0, 0, 0, 1, 86.41818182, 91.27272727, 83.18181818, 712, 0, 0, 0, 0, 0, 0, 22, 2008, 33, 2745, 29200, 247, 16, 32, 0, 0, 0, 0,0, 0, 0, 0, SSH-Patator

El dataset original está extremadamente desbalanceado, habiendo más de dos millones de ejemplos de tráfico benigno y poco más de 500.000 de diversos tipos de ataques. Aprovechando que los ejemplos de cada tráfico estaban divididos por días y que, para cada día, el CIC indica qué tipo de tráfico hay, se eliminaron todos los ejemplos relativos al primer día del conjunto de datos, ya que este incluía únicamente tráfico benigno. A su vez, como la información correspondiente a cada día estaba representada en un fichero independiente, se unificaron el resto de días en un único archivo.

Por último, para determinados campos de los datos había valores con los que no se podía trabajar. Concretamente, en determinados ejemplos, se alcanzaban valores de infinito y NaN, siempre de forma conjunta y siempre en los mismos campos, habitualmente relacionados con valores promedio o desviaciones típicas. Dado que por el contexto del problema dichos campos solo pueden tomar valores positivos, se decidió sustituir los valores de infinito por el valor flotante más alto permitido por *Python* y los valores de NaN por -1.

Se planteó también simplemente eliminar los ejemplos que incluyeran estos valores, particularmente si eran de tráfico benigno, ya que ayudaría con el desbalanceo del problema; Sin embargo, dado que este caso puede darse en diversos tipos de ataque adicionalmente, algunos de los cuales presentan de por sí muy pocos ejemplos, se optó por realizar la sustitución de esos valores incorrectos de la forma previamente explicada.

Los detalles sobre los datos resultantes del dataset aparecen representados en la tabla 3.2.

Tipo de tráfico	Tipo de ataque	Número de ejemplos
Benigno	-	**1.743.179**
Ataque	DDoS	128.027
Ataque	Portscan	158.930
Ataque	Bot	1.966
Ataque	Infiltración	36
Ataque	WebAttack Bruteforce	1.507
Ataque	Heartbleed	11
Ataque	WebAttackXSS	652
Ataque	WebAttack SQL Injection	21
Ataque	FTP Patator	7.938
Ataque	SSH Patator	5.897
Ataque	Slowloris	5.796
Ataque	SlowHttpTest	5.449
Ataque	DoSHulk	231.073
Ataque	GoldenEye	10.293
Ataque	**Suma total**	**557.646**

Tabla. 3.2: Detalles de los datos del dataset CIC-IDS2017.

3.2.2. CIC-CSE-IDS2018

Acompañando al anterior, encontramos el **CSE-CIC-IDS2018**, un dataset creado de manera colaborativa entre el CSE y el CIC [46]. Este dataset presenta varios escenarios de ataque diferentes, generados empleando 50 víctimas, y con los ataques afectando 420 máquinas y 30 servidores. Se incluyen *logs* con la información del tráfico de datos para cada una de estas máquinas, además de 80 variables extraídas mediante el uso de CICFlowMeter. Los ataques incluidos en el dataset son similares a los del *dataset* previo. A su vez, las variables que contiene cada uno de los ejemplos de este dataset también coinciden con las del dataset previo, pero se han añadido a los 78 campos previos dos nuevos: el protocolo utilizado para la conexión en cuestión y la marca de tiempo de cuando se produjo.

De nuevo, mostraremos dos ejemplos de este dataset:

- 22, 6, 14/02/2018 08:40:23, 8804066, 14, 11, 1143, 2209, 744, 0, 81.64285714,

203.745, 976, 0, 200.8181818182, 362.249, 380.73317487, 2.839, 366836.083, 511356. 609732762, 1928102, 21, 8804066, 677235.846153846, 532416.971, 1928102, 246924, 7715481, 771548.1, 755543.082716951, 2174893, 90, 0, 0, 0, 0, 456, 360, 1.590, 1.2494227099, 0, 976, 128.9230769231, 279.7630315931, 78267.3538461539, 0, 0, 0, 1, 0, 0, 0, 0, 0, 134.08, 81.6428571429, 200.818, 0, 0, 0, 0, 0, 0, 14, 1143, 11, 2209, 5808, 233, 6, 32, 0, 0, 0, 0, 0, 0, 0, 0, Benign

- 80, 6, 16/02/2018 01:48:36,3209761, 4, 5, 302, 935, 302, 0, 75.5, 151, 935, 0, 187, 418.1447118, 385.3869494, 2.803947085, 401220.125, 1090269.609, 3098929, 6, 3190583, 1063527.667, 1762815.115, 3098935, 26986, 3209755, 802438.75, 1531507.937, 3098938, 50, 0, 0, 0, 0, 136, 168, 1.246198705, 1.557, 0, 935, 123.7, 300.4463531, 90268.01111, 0, 0, 0, 0, 1, 1, 0, 0, 1, 137.4444444, 75.5, 187, 0, 0, 0, 0, 0, 0, 4, 302, 5, 935, 225, 219, 1, 32, 0, 0, 0, 0, 0, 0, 0, 0, DoS attacks-Hulk

De manera análoga al CIC-IDS2017, el conjunto de datos CIC-CSE-IDS2018 presenta la información dividida en varios ficheros, separados a su vez por horas y días. Debido a que en función a la hora y el día el modelo de clasificación podría salir sesgado (ya que en días y horas determinados solo hay ataques de un tipo), se optó por eliminar la columna relativa de la marca de tiempo durante el preprocesamiento del dataset con objetivo de que los resultados fueran lo más cercanos que se pudiera a la realidad (habitualmente, los ataques no ocurren en un día ni hora específicos).

El CIC-CSE-IDS2018 es el dataset más extenso de todos con los que se trabaja, conteniendo más de dos millones de datos de tráfico benigno y casi un millón de tráfico maligno. Como se puede apreciar, si bien es un dataset desbalanceado, lo es menos que el previamente mencionado. Adicionalmente, posee solamente seis tipos de ataque que, en comparación con los 17 que presentaba el dataset previo, causa que ninguna de las clases de ataques sea extremadamente minoritaria con respecto al resto (siendo la clase de ataque con menor número de ejemplos el ataque *Slowloris*, con cerca de 11.000 ejemplos, y la que más ejemplos tiene los *DoSHulk*, con casi medio millón de ejemplos, mientras que en el CIC-IDS2017 la clase mayoritaria tenía 231.000 ejemplos y la minoritaria tan solo 11).

Al igual que en el caso previo, se unificaron los diversos archivos de datos en uno solo para facilitar su manejo. Sin embargo, al contrario que el dataset previo, en ningún fichero hay tráfico estrictamente benigno, por lo que se utiliza el dataset completo en esta ocasión. El mismo problema con respecto de los datos con valores de infinito o NaN aparecía en este dataset, así que se optó por resolverlo aplicando exactamente los mismos criterios que en el previo.

Los datos completos sobre el dataset se pueden encontrar en la tabla 3.3.

Tipo de tráfico	Tipo de ataque	Número de ejemplos
Benigno	-	**2.110.356**
Ataque	FTP-Bruteforce	193.360
Ataque	SSH-Bruteforce	187.589
Ataque	DoS-GoldenEye	41.508
Ataque	DoS-Slowloris	10.990
Ataque	DoS-SlowHttpTest	91.434
Ataque	DoS-AttackHulk	461.912
Ataque	**Suma total**	**986.793**

Tabla. 3.3: Detalles de los datos del dataset CIC-CSE-IDS2018.

3.2.3. CIC-DDoS2019

A estos *datasets* sumaremos el **CIC-DDoS2019**, que como su nombre indica, posee diversos ejemplos concretamente sobre ataques de tipo DDoS variados. Este dataset, también elaborado por el CIC [47], hace la distinción entre ataques DDoS por reflexión y ataques DDoS por explotación. En los ataques DDoS por reflexión, la identidad del atacante se mantiene oculta gracias al uso de componentes de terceros que resultan legítimos. Los paquetes son enviados a servidores reflectores que, a su vez, los reenvían a la IP de la víctima para sobresaturarla con paquetes de respuesta. Estos ataques pueden darse desde la capa de transporte usando protocolos como UDP, TCP, etc.

Los ataques de explotación, por otro lado, buscan el colapso del sistema enviando una cantidad enorme de paquetes SYN para que el servidor víctima falle o, en su defecto, el *UDP-Lag* , que busca ralentizar la conexión de quien lo sufre.

El *dataset* contiene ejemplos tanto de ataques de ambos tipos de DDoS como de tráfico benigno. En esta ocasión, toda la información aparece representada en un único archivo. Con respecto a las variables que hay para cada ejemplo, son las mismas que en los *datasets* ya mencionados. Cabe destacar que, en cada caso, además de la clase (benigno o ataque), también aparece el tipo de ataque que es, dando lugar a que haya dos variables que pueden actuar como clase (y, en tráfico benigno, aparece la etiqueta 'benigno' dos veces seguidas por dicho motivo). A continuación, se mostrarán dos ejemplos de elementos de este dataset:

- 17, 20770, 2, 2, 64.0, 630.0, 32.0, 32.0, 32.0, 0.0, 315.0, 315.0, 315.0, 0.0, 33413.58, 192.58546, 6923.3335, 11987.235, 20765.0, 2.0, 2.0, 2.0, 0.0, 2.0, 2.0, 3.0, 3.0, 0.0, 3.0, 3.0, 0, 0, 0, 0, 40, 40, 96.29273, 96.29273, 32.0, 315.0, 145.2, 155.00548, 24026.7, 0, 0, 0, 0, 0, 0, 0, 0, 1.0, 181.5, 32.0, 315.0, 0, 0, 0, 0, 0, 0, 2, 64, 2, 630, -1, -1, 1, 20, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, Benign, Benign
- 1, 17, 2, 2, 0, 802.0, 0.0, 401.0, 401.0, 401.0, 0.0, 0.0, 0.0, 0.0, 0.0, 401000000.0, 1000000.0, 2.0, 0.0, 2.0, 2.0, 2.0, 2.0, 0.0, 2.0, 2.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0, 0, 0, 0, 40, 0, 1000000.0, 0.0, 401.0, 401.0, 401.0, 0.0, 0.0, 0, 0, 0, 0, 0, 0, 0, 0, 0.0, 601.5, 401.0, 0.0, 0, 0, 0, 0, 0, 0, 2, 802, 0, 0, -1, -1, 1, 20, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, UDP, Attack

Al contrario que hasta ahora, toda la información relevante está en un único fichero. Este es, a su vez, el dataset menos desbalanceado, pero también es el más pequeño. Con menos de 500.000 datos totales, el dataset presenta alrededor de 300.000 ejemplos de ataques y alrededor de 100.000 ejemplos de tráfico benigno. Entre los ejemplos de tráfico maligno existen un total de 17 clases. Sin embargo, si prestamos atención a las clases, podemos ver que dos de ellas son *UDP Lag* y *UDP-Lag*. En consecuencia, se realizó una conversión, de manera que todos los ejemplos de *UDP Lag* (que eran únicamente 55) se modificaron para devolver la etiqueta *UDP-Lag* (que tenía 8872), entendiendo que había sido un problema de consistencia durante la redacción de los datos.

Los datos completos relativos al dataset pueden encontrarse en la tabla 3.4.

Tipo de tráfico	Tipo de ataque	Número de ejemplos
Benigno	-	**97.831**
Ataque	UDP	18.090
Ataque	MSSQL	8.523
Ataque	Portmap	685
Ataque	Syn	49.373
Ataque	NetBIOS	644
Ataque	UDPLag	8.927
Ataque	LDAP	1.906
Ataque	DrDos DNS	3.669
Ataque	Web DDoS	51
Ataque	TFTP	98.917
Ataque	DrDoS UDP	10.420
Ataque	DrDos SNMP	2.717
Ataque	DrDoS NetBIOS	598
Ataque	DrDoS LDAP	1.440
Ataque	DrDoS MSSQL	6.212
Ataque	DrDoS NTP	121.368
Ataque	**Suma total**	**333.540**

Tabla. 3.4: Detalles de los datos del dataset CIC-DDoS2019.

3.2.4. KDD Cup 1999

Otro dataset empleado en algunas de las investigaciones previamente nombradas es el **KDD Cup 1999**. Este dataset contiene información sobre diversos ataques que se pueden llevar a cabo sobre una red. Sin embargo, como su propio nombre indica, la información que contiene está bastante desactualizada, y varios de los ataques que incluye ya han quedado obsoletos porque la propia seguridad de la red dificulta considerablemente el que puedan llegar a ser eficaces. Pese a ello, se menciona este dataset debido a que fue el más empleado en aquellos artículos de 2016 y anteriores, que son previos a los *datasets* previamente desarrollados.

3.3. Implementación de modelos de clasificación

Con objetivo de cumplir los objetivos ya planteados, se ha llevado a cabo la implementación de diversos algoritmos, que pueden ser reducidos en dos grandes grupos:

- **Algoritmos auxiliares**. Estos algoritmos se emplean para facilitar la implementación de diversas funciones de utilidad, como la lectura de los parámetros de configuración o de los datos para entrenar y probar los modelos. Dentro de esta lista de algoritmos están presentes algoritmos para la normalización de cadenas de texto y extracción de parámetros de configuración del archivo correspondiente.
- **Algoritmos de modelos**. Estos algoritmos incluyen la funcionalidad necesaria para construir y entrenar un modelo, así como para probarlo y determinar su eficacia a la hora de resolver el problema propuesto. En este caso concreto, se incluyen algoritmos para la creación, entrenamiento y evaluación de *RandomForest*, perceptrón multicapa, CNN, RNN y *transformers*.

La lista completa de todos los algoritmos implementados puede ser encontrada en la tabla 3.5.

Algoritmo	Tipo	Descripción	Referencia
fixName	Utilidad	Algoritmo empleado para eliminar caracteres extraños del texto.	Listado 3.1.
readConfig	Utilidad	Algoritmo empleado para extraer la configuración del programa del fichero correspondiente.	Listado 3.2
readData	Utilidad	Algoritmo empleado para extraer los datos del *dataset* correspondiente y cargarlos en memoria.	Listados 3.3, 3.4 y 3.5
checkAccuracy	Calidad	Algoritmo empleado para determinar el porcentaje de acierto de las predicciones de un modelo.	Listado 3.6
confusionMatrix	Calidad	Algoritmo que genera una matriz de confusión a partir de las predicciones de un modelo y las etiquetas reales de los datos.	Listado 3.7
RandomForest	Modelo	Implementación, entrenamiento y evaluación el modelo RandomForest.	Listado 3.8
Perceptrón Multicapa	Modelo	Implementación, entrenamiento y evaluación del Perceptrón Multicapa.	Listados 3.9 y 3.10
CNN	Modelo	Implementación, entrenamiento y evaluación del modelo CNN.	Listados 3.11 y 3.12
RNN	Modelo	Implementación, entrenamiento y evaluación del modelo RNN.	Listados 3.13 y 3.14
Transformer	Modelo	Implementación, entrenamiento y evaluación del modelo Transformer.	Listado 3.15

Tabla. 3.5: Algoritmos implementados.

A continuación, se muestra el pseudocódigo de muy alto nivel correspondiente con todos los modelos que se implementarán posteriormente. En los subapartados consecuentes se mostrará el código correspondiente a cada modelo individual.

Algoritmo 1: Flujo de trabajo con los modelos de clasificación

```
Input: <Dataset>D, <String>modoEjecución M
Result: Un modelo entrenado con un set de datos concreto
variables ← leerVariables(D);
etiquetas ← leerEtiquetas(D, M);
resultadoFinal[] ;
folds ← 0 ;
modelo ← crearModelo() ;
while folds <nFolds do
    trainVar, trainLabel ← getStratifiedFold(variables, etiquetas);
    testVar ← variables - trainVar ;
    testLabel ← etiquetas - trainLabel ;
    modelo ← entrenarModelo(trainVar, trainLabel);
    predicciones ← modelo(testVar);
    ratioAcierto ← comparar(predicciones, testLabel) ;
    mostrar(ratioAcierto) ;
    resultadoFinal ← resultadoFinal.añadir(ratioAcierto) ;
    folds ← folds + 1;
end
mostrarResultadosGlobales() ;
```

A continuación, procederemos a explicar el pseudocódigo previo. Como se puede apreciar, partimos de un dataset D y un modo de ejecución M. Estos dos parámetros serán recibidos mediante un fichero de configuración. El dataset podrá ser cualquier archivo en formato CSV, aunque en este TFG se han usado meramente los tres previamente descritos. El modo de ejecución es una cadena de texto que determina cómo se procesarán los datos para el problema, y que puede tomar tres valores:

- **BinaryProblem**: Únicamente se considera si el tráfico es benigno o malicioso. Cualquier tráfico cuyo tipo sea distinto de benigno queda englobado bajo la misma etiqueta.
- **MaliciousOnly**: Únicamente se considera el tráfico malicioso. Esta configuración se utiliza para tratar de discernir entre tipos de tráfico malicioso.
- **ClassicProblem**: Utilizado solo en pruebas iniciales, este modo busca diferenciar directamente entre el tráfico benigno y los distintos tipos de tráfico malicioso.

Una vez los datos han sido procesados y almacenados en memoria, a continuación se debe establecer la separación entre los conjuntos de entrenamiento y test. El primero

de ellos será usado para entrenar al modelo con los datos, mientras que el segundo se reserva para analizar los resultados del modelo y ver hasta qué punto es realmente capaz de predecir la clase del tráfico que se recibe.

Para ello, se hace uso de validación cruzada estratificada, de manera que, para cada iteración del bucle, se divide el conjunto completo de datos en cinco subconjuntos cuya proporción de clases sea lo más similar posible a los datos originales. Cuatro de estos subconjuntos se usan como conjunto de entrenamiento, mientras que el restante se usa como conjunto de test. Para establecer los resultados finales, se devuelve la media de las ejecuciones como resultado final. Cada uno de estos subconjuntos recibe el nombre de *fold*.

En determinados modelos, es posible que el proceso de entrenamiento se realice en varias *epochs*, o épocas, para que el modelo pueda ajustarse mejor a los datos y mejorar sus predicciones. Si es el caso, simplemente habría otro bucle dentro del ya visto que solo englobaría en su interior a la parte del entrenamiento.

3.3.1. Funciones de utilidad

Como se ha podido apreciar en el apartado previo, además de la construcción, entrenamiento y evaluación de los modelos, es necesario llevar a cabo diversas tareas relacionadas con el conjunto de datos a tratar y el tipo de problema a tratar (distinguir entre tráfico benigno o malicioso; o distinguir entre tipos de tráfico malicioso). Adicionalmente, también se vuelve necesario el crear una función que permita mostrar por pantalla los resultados obtenidos, además de una adicional para crear una matriz de confusión en la que queden reflejados los errores a la hora de distinguir entre clases.

Con objetivo de modularizar el código lo máximo posible y facilitar la implementación de modelos a lo largo del proceso de desarrollo, se implementó un conjunto de funciones de utilidad relacionadas con esas tareas, que queda representado a continuación.

Comenzaremos con el listado 3.1, que presenta una función para eliminar caracteres como el retorno de carril del texto. Esta función se usa habitualmente en el procesamiento de documentos de texto, como en el listado 3.2 o en el listado 3.3.

```
1 def fixName(string):
2     return "".join(string.splitlines())
```

Listado 3.1: Normalización de cadenas de texto

A continuación, mostraremos el listado 3.2, cuyo objetivo es extraer los datos necesarios para la ejecución del programa del fichero de configuración que reciba como parámetro. El archivo de configuración debe incluir información sobre una ruta de salida donde volcar los datos, el fichero con el conjunto de datos que se vaya a utilizar y, por último, el modo de ejecución del programa de acuerdo a lo previamente descrito. Si falta alguno de los elementos mencionados, el programa lanzará una excepción.

```
def readConfig(configFile):
    rOutput = ""
    dataFile = ""
    executionMode = ""
    model = ""
    with open(configFile) as config:
        for line in config:
            if line.find("rOutput") != -1:
                aux = line.split('=')
                rOutput = fixName(aux[1])
            if line.find("dataFile") != -1:
                aux = line.split('=')
                dataFile = fixName(aux[1])
            if line.find("executionMode") != -1:
                aux = line.split('=')
                executionMode = fixName(aux[1])
            if line.find("targetModel") != -1:
                aux = line.split('=')
                model = fixName(aux[1])
    if rOutput == "":
        raise Exception("Falta un parámetro en el archivo de configuración. Debe incluir:
            rOutput (formato param=valor)")
    if dataFile == "":
        raise Exception("Falta un parámetro en el archivo de configuración. Debe incluir:
            dataFile (formato param=valor)")
    if executionMode == "":
        raise Exception("Falta un parámetro en el archivo de configuración. Debe incluir:
            executionMode (formato param=valor)")
    if model == "":
        raise Exception("Falta un parámetro en el archivo de configuración. Debe incluir:
            targetModel (formato param=valor)")
    return rOutput, dataFile, executionMode, model
```

Listado 3.2: Extracción de parámetros del fichero de configuración.

A continuación, mostraremos la función encargada de extraer los datos del fichero con el conjunto de datos que haya sido elegido en base al modo de ejecución del programa que también haya sido escogido. Como se explicó previamente, el si solo se toman los datos maliciosos, los benignos o una combinación de ambos depende del modo de ejecución elegido, como se puede apreciar en los listados 3.3, 3.4 y 3.5. Cabe destacar que estos tres algoritmos forman una única función en el código, pero por motivos de legibilidad se ha decidido separarlos en la presente memoria.

El listado 3.3 muestra la extracción de datos del conjunto de datos cuando el modo

de ejecución es *ClassicProblem*, esto es, cuando se reciben todos los datos de tráfico benigno y malicioso y se trata de discernir entre ellos.

```
def readData(dataFile, executionMode):
    variables = []
    labels = []
    labelDictionary = {}
    labelCount = 0
    if executionMode == "ClassicProblem":
        with open(dataFile) as data:
            for line in data:
                aux = line.split(",")
                label = fixName(aux.pop())
                if label in labelDictionary:
                    labels.append(labelDictionary[label])
                else:
                    labelDictionary[label] = labelCount
                    labels.append(labelCount)
                    labelCount+=1
                prunedData = []
                for variable in aux:
                    if variable == "NaN" or variable == "?":
                        prunedData.append(-1.0)
                    else:
                        if variable == "Infinity":
                            prunedData.append(INFINITY)
                        else:
                            prunedData.append(float(variable))
                variables.append(prunedData)
    return variables,labels,labelDictionary
```

Listado 3.3: Extracción de parámetros del fichero de configuración (formato *ClassicProblem*).

Por otro lado, el listado 3.4 muestra el cómo se extraen los datos cuando únicamente se considera el tráfico benigno o malicioso, ignorando el tipo concreto de tráfico malicioso. Como se puede apreciar, si la etiqueta de un dato determinado es distinta de *benign*, entonces simplemente se establece la etiqueta como *malicious*.

```
def readData(dataFile, executionMode):
    variables = []
    labels = []
    labelDictionary = {}
    labelCount = 0
    if executionMode == "BinaryProblem":
        labelDictionary["Benign"] = 0
        labelDictionary["Malicious"] = 1
        with open(dataFile) as data:
            for line in data:
                aux = line.split(",")
                label = fixName(aux.pop())
                if (label != "Benign"):
                    label = "Malicious"
                labels.append(labelDictionary[label])
                prunedData = []
```

```
                for variable in aux:
                    if variable == "NaN" or variable == "?":
                        prunedData.append(-1.0)
                    else:
                        if variable == "Infinity":
                            prunedData.append(INFINITY)
                        else:
                            prunedData.append(float(variable))
                variables.append(prunedData)
    return variables,labels,labelDictionary
```

Listado 3.4: Extracción de parámetros del fichero de configuración (formato *BinaryProblem*).

Por último, en el listado 3.5, se ignora cualquier dato cuya etiqueta sea *benign* para mantener únicamente el tráfico malicioso y, así, tratar de distinguir entre diversos tipos de ataque usando el modelo.

```
def readData(dataFile, executionMode):
    variables = []
    labels = []
    labelDictionary = {}
    labelCount = 0
    if executionMode == "MaliciousOnly":
        with open(dataFile) as data:
            for line in data:
                aux = line.split(",")
                label = fixName(aux.pop())
                if (label != "Benign"):
                    if label in labelDictionary:
                        labels.append(labelDictionary[label])
                    else:
                        labelDictionary[label] = labelCount
                        labels.append(labelCount)
                        labelCount+=1
                    prunedData = []
                    for variable in aux:
                        if variable == "NaN" or variable == "?":
                            prunedData.append(-1.0)
                        else:
                            if variable == "Infinity":
                                prunedData.append(INFINITY)
                            else:
                                prunedData.append(float(variable))
                    variables.append(prunedData)
    return variables,labels,labelDictionary
```

Listado 3.5: Extracción de parámetros del fichero de configuración (formato *MaliciousOnly*).

Ahora que hemos terminado con los algoritmos para extraer la información del conjunto de datos, pasaremos a los algoritmos que tienen como objetivo mostrar los resultados de las evaluaciones de los algoritmos. El primer listado que mostraremos

será el 3.6, que toma como entradas las predicciones y las etiquetas reales y hace una comparación entre todas ellas para determinar el porcentaje de acierto total y por clases.

```
def checkAccuracy(prediction, y_test):
    correct = 0
    for i in range (len(prediction)):
        if prediction[i] == y_test[i]:
            correct += 1
    print("Accuracy␣of␣the␣model:␣" + str(float(100 * correct / len(y_test))))
    class_correct = {}
    class_total = {}
    for i in range(len(y_test)):
        label = y_test[i].item()
        if label in class_total:
                class_total[label] += 1
        else:
            class_total[label] = 1
        if label in class_correct:
            if prediction[i] == y_test[i]:
                class_correct[label] += 1
        else:
            class_correct[label] = 0
    return class_total, class_correct
```

Listado 3.6: Cálculo del porcentaje de acierto de una predicción de un modelo.

Por último, mostraremos el listado 3.7, que calcula la matriz de confusión en base a los resultados de la predicción comparándolos con los de las etiquetas. Este listado, junto con el 3.6, nos permiten comprobar los resultados del modelo fácilmente durante la etapa de la experimentación.

```
def confusionMatrix(labelDictionary, prediction, y_test):
    toRet = []
    for i in range(len(labelDictionary)):
        aux = []
        for i in range(len(labelDictionary)):
            aux.append(0)
        toRet.append(aux)
    for i in range(len(y_test)):
        toRet[y_test[i].item()][prediction[i].item()] += 1
    return toRet
```

Listado 3.7: Cálculo de la matriz de confusión a partir de los resultados de un modelo.

3.3.2. RandomForest

A continuación, en el listado 3.8, se muestra el código necesario para crear el modelo *RandomForest* usado para diversas pruebas.

```
    from sklearn.ensemble import RandomForestClassifier
    import numpy as np

def run(variables, labels, labelDictionary):
    nFolds = 5
    kf = StratifiedKFold(n_splits=nFolds)
    accumulatedAccuracy = 0

    folds = enumerate(kf.split(variables,labels))
    for i, (train, test) in folds:
        clf = RandomForestClassifier(n_estimators = 100)
        X_train = []
        y_train = []
        X_test = []
        y_test = []
        for element in train:
           X_train.append(variables[element])
           y_train.append(labels[element])
        for element in test:
           X_test.append(variables[element])
           y_test.append(labels[element])

        X_train = torch.tensor(X_train)
        X_test = torch.tensor(X_test)
        y_train = torch.tensor(y_train)
        y_test = torch.tensor(y_test)

        clf.fit(X_train, np.ravel(y_train))
        prediction = clf.predict(X_test)
        print("Fold " + str(i) + ":\n")
        accuracy, class_total, class_correct = checkAccuracy(prediction, y_test)
        print(class_total)
        print(class_correct)
        print(labelDictionary)
        for i in range(len(labelDictionary)):
            print("Accuracy of " + str(i) + ": " +  str(float(round(100 * class_correct[i
                ] / class_total[i], 4))))
        matrix = confusionMatrix(labelDictionary, prediction, y_test)
        for i in range(len(labelDictionary)):
            print(matrix[i])

        accumulatedAccuracy += accuracy

    print("Precisión promedio del modelo: " + str(float(accumulatedAccuracy / nFolds)))
```

Listado 3.8: Creación del modelo *RandomForest*.

Dado que el modelo *RandomForest* no está disponible de manera directa en |textitPyTorch, para este caso concreto se utiliza *scikit-learn* [43], que presenta el modelo usado y que puede ser fácilmente compatibilizado con el resto de funciones usadas con los modelos restantes, ya que el paso de tensores de *Numpy* a *PyTorch* es trivial.

Como se puede apreciar, la biblioteca *sklearn.ensemble* incluye al objeto *RandomForestClassifier*, que permite crear el modelo de manera muy sencilla. Análogamente,

el entrenamiento también resulta totalmente trivial, siendo tan solo necesario utilizar la función *clf.fit* recibiendo como parámetros los atributos y clases del conjunto de entrenamiento. Posteriormente, es suficiente con utilizar la función *clf.predict* recibiendo como parámetro los atributos del conjunto de test para obtener las predicciones correspondientes.

3.3.3. Perceptrón multicapa

En este apartado, se muestra y explica cómo crear un modelo de un perceptrón multicapa. El código empleado se puede observar en el listado 3.9.

```
def run(variables, labels, labelDictionary):
    neurons_in = len(variables[0])
    neurons_out = len(labelDictionary)
    neurons_hidden = int((2*neurons_in/3) + neurons_out)
    nFolds = 5
    kf = StratifiedKFold(n_splits=nFolds)
    epochs = 5
    accumulatedAccuracy = 0

    if torch.cuda.is_available():
       device = torch.device("cuda")
    else:
       device = torch.device("cpu")

    folds = enumerate(kf.split(variables,labels))
    for i, (train, test) in folds:
        model = nn.Sequential(nn.Linear(neurons_in, neurons_hidden),
                              nn.ReLU(),
                              nn.Linear(neurons_hidden, neurons_hidden),
                              nn.ReLU(),
                              #Se pueden insertar capas adicionales
                              nn.Linear(neurons_hidden, neurons_out),
                              nn.Sigmoid())

        model.to(device)

        loss_criterion = nn.CrossEntropyLoss()
        optimizer = torch.optim.SGD(model.parameters(),
                                    lr=0.01)

        X_train = []
        y_train = []
        X_test = []
        y_test = []
        for element in train:
           X_train.append(variables[element])
           y_train.append(labels[element])
        for element in test:
           X_test.append(variables[element])
           y_test.append(labels[element])

```

```
        X_train = torch.tensor(X_train)
        X_test = torch.tensor(X_test)
        y_train = torch.tensor(y_train)
        y_test = torch.tensor(y_test)
        X_train = X_train.to(device)
        y_train = y_train.to(device)
```

Listado 3.9: Creación del modelo perceptrón multicapa.

En esta ocasión, sí estaremos usando *PyTorch* y, más concretamente, *nn.Sequential*. La ventaja que nos ofrece esta clase es que facilita considerablemente la implementación de modelos secuenciales, como puede ser el perceptrón multicapa. Únicamente hay que especificar el tipo de capa que se desea incluir y los parámetros de entrada y salida.

Para la primera capa, el número de entradas es equivalente al número de variables que tenga el dataset sobre el que se ejecutará el modelo. Las capas intermedias tienen un número de neuronas dependientes tanto del número de variables como del número de potenciales clases. La capa de salida cuenta con tantas neuronas como clases a predecir tenga el problema.

Tras definir el modelo, se contempla si la GPU del ordenador está disponible para aplicar CUDA. Esto permite acelerar considerablemente el proceso en caso afirmativo, ya que la GPU es capaz de paralelizar cálculos de esta índole con gran eficiencia. Por último, se definen el criterio de pérdida y el optimizador.

En el listado 3.10, se muestra el proceso de entrenamiento y evaluación de uno de los subconjuntos obtenidos mediante validación cruzada.

```
        model.train()
        for epoch in range(epochs):
            train_outputs = model(X_train) # model outputs raw logits
            loss = loss_criterion(train_outputs, y_train)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

        X_test = X_test.to(device)
        y_test = y_test.to(device)
        model.eval()
        with torch.inference_mode():
            test_logits = model(X_test)
            test_pred = torch.softmax(test_logits, dim=1).argmax(dim=1)
            print("Fold " + str(i) + ": \n")
            accuracy, class_total, class_correct = checkAccuracy(test_pred, y_test)
        print(class_total)
        print(class_correct)
        print(labelDictionary)
        for i in range(len(labelDictionary)):
            print("Accuracy of " + str(i) + ": " +  str(float(round(100 * class_correct[i
```

```
                ] / class_total[i], 4))))
        matrix = confusionMatrix(labelDictionary, test_pred, y_test)
        for i in range(len(labelDictionary)):
            print(matrix[i])

        accumulatedAccuracy += accuracy

    print("Precisión promedio del modelo: " + str(float(accumulatedAccuracy / nFolds)))
```

Listado 3.10: Entrenamiento y evaluación del modelo perceptrón multicapa.

El proceso de entrenamiento y evaluación es muy similar entre todos los modelos. Primero, para cada época, se entrena el modelo y se obtienen las predicciones del modelo a partir de las cuales se calcula la pérdida. Después, se llama a *optimizer.zero_grad* para dejar a 0 el optimizador y que no arrastre información de etapas previas. Posteriormente, se realiza la propagación hacia atrás.

Una vez el entrenamiento ha concluido, se activa la funcionalidad de evaluación del modelo y se le entregan los atributos del subconjunto de test. A partir de las predicciones, se llama a la función *softmax* para hacerlas legibles para el humano, y se calcula la precisión del modelo.

3.3.4. Red neuronal convolucional

En este apartado, se mostrará la implementación de una red neuronal convolucional. El código correspondiente puede verse en el listado 3.11.

```
import torch
from torch import nn

batch_size = 32

class Model(nn.Module):
  def __init__(self, input_size, output_size, hidden_dim, elements):
    super(Model, self).__init__()

    self.hidden_dim = hidden_dim
    self.n_elements = elements

    self.conv1 = nn.Conv1d(input_size, hidden_dim, kernel_size=(1))
    self.act1 = nn.ReLU()
    self.drop1 = nn.Dropout(0.3)

    self.conv2 = nn.Conv1d(hidden_dim, hidden_dim, kernel_size=(1))
    self.act2 = nn.ReLU()
    self.pool2 = nn.MaxPool1d(kernel_size=(1))

    self.fc3 = nn.Linear(hidden_dim, hidden_dim)
    self.act3 = nn.ReLU()
```

```
23      self.drop3 = nn.Dropout(0.5)
24
25      self.fc4 = nn.Linear(hidden_dim, output_size)
26
27    def forward(self, x):
28      x = self.act1(self.conv1(x))
29      x = self.drop1(x)
30      x = self.act2(self.conv2(x))
31      x = self.pool2(x)
32      x = self.act3(self.fc3(torch.transpose(x, 0, 1)))
33      x = self.drop3(x)
34      x = self.fc4(x)
35      return x
```

Listado 3.11: Implementación del modelo de red neuronal convolucional.

Al contrario que en el caso del perceptrón multicapa, no es posible utilizar *sequential* en esta ocasión, dado que solo permite capas lineales y de activación y para crear una CNN es necesario utilizar capas convolucionales, además de alguna capa de *pooling*. Por ello, definiremos nuestro modelo como una clase hija de *nn.Module*, creando las capas correspondientes, y la función de *forward* usada tanto en el entrenamiento como en la evaluación.

A continuación, en el listado 3.12, se muestra el resto del código necesario para el entrenamiento y evaluación usando el modelo.

```
1  def run(variables, labels, labelDictionary):
2      neurons_in = len(variables[0])
3      neurons_out = len(labelDictionary)
4      neurons_hidden = int((2*neurons_in/3) + neurons_out)
5      nFolds = 5
6      kf = StratifiedKFold(n_splits=nFolds)
7      epochs = 3
8      accumulatedAccuracy = 0
9
10     if torch.cuda.is_available():
11        device = torch.device("cuda")
12     else:
13        device = torch.device("cpu")
14
15     folds = enumerate(kf.split(variables,labels))
16
17     for i, (train, test) in folds:
18         X_train = []
19         y_train = []
20         X_test = []
21         y_test = []
22         for element in train:
23            X_train.append(variables[element])
24            y_train.append(labels[element])
25         for element in test:
26            X_test.append(variables[element])
27            y_test.append(labels[element])
28
```

```
        X_train = torch.tensor(X_train)
        X_test = torch.tensor(X_test)
        y_train = torch.tensor(y_train)
        y_test = torch.tensor(y_test)
        X_train = X_train.to(device)
        y_train = y_train.to(device)

        elements = X_train.size(0)
        X_train = torch.transpose(X_train, 0, 1)
        X_test = torch.transpose(X_test, 0, 1)

        model = Model(neurons_in, neurons_out, neurons_hidden, elements)
        loss_criterion = nn.CrossEntropyLoss()
        optimizer = torch.optim.SGD(model.parameters(),
                                    lr=0.0001)

        model.to(device)
        X_train = X_train.to(device)
        y_train = y_train.to(device)
        model.train()
        for epoch in range(epochs):
            optimizer.zero_grad()
            print("Empezando época " + str(epoch))
            train_outputs = model(X_train) # model outputs raw logits
            print("\tCalculando pérdida")
            loss = loss_criterion(train_outputs, y_train)
            print("\tAplicando backward propagation")
            loss.backward()
            print("\tActualizando optimizador")
            optimizer.step()
            print("Terminando época " + str(epoch))

        print("Comenzando evaluación.")
        X_test = X_test.to(device)
        y_test = y_test.to(device)
        model.eval()
        with torch.inference_mode():
          test_logits = model(X_test)
          test_pred = torch.softmax(test_logits, dim=1).argmax(dim=1)
          print("Fold " + str(i) + ": \n")
          accuracy, class_total, class_correct = checkAccuracy(test_pred, y_test)
        print(class_total)
        print(class_correct)
        print(labelDictionary)
        for i in range(len(labelDictionary)):
            print("Accuracy of " + str(i) + ": " +  str(float(round(100 * class_correct[i
                ] / class_total[i], 4))))
        matrix = confusionMatrix(labelDictionary, test_pred, y_test)
        for i in range(len(labelDictionary)):
          print(matrix[i])

        accumulatedAccuracy += accuracy

    print("Precisión promedio del modelo: " + str(float(accumulatedAccuracy / nFolds)))
```

Listado 3.12: Entrenamiento y evaluación con una CNN.

Debido al funcionamiento de las redes neuronales convolucionales, es necesario

pasar como parámetro adicionalmente el número de datos de entrada, además de los atributos que contiene cada dato. Por lo demás, la implementación funciona de manera completamente análoga al caso previamente mostrado.

3.3.5. Red neuronal recurrente

En este apartado, se presenta el código para la implementación de una red neuronal recurrente (listado 3.13),

```
1
2  class Model(nn.Module):
3    def __init__(self, input_size, output_size, hidden_dim, n_layers):
4        super(Model, self).__init__()
5
6        self.hidden_dim = hidden_dim
7        self.n_layers = n_layers
8
9        self.rnn = nn.RNN(input_size, hidden_dim, n_layers, batch_first=True)
10       self.fc = nn.Linear(hidden_dim, output_size)
11
12   def forward(self, x):
13
14       batch_size = x.size(0)
15       hidden = self.init_hidden(batch_size)
16       hidden = hidden.to(device)
17       out, hidden = self.rnn(x, hidden)
18
19       out = out.contiguous().view(-1, self.hidden_dim)
20       out = self.fc(out)
21
22       return out, hidden
23
24   def init_hidden(self, batch_size):
25       hidden = torch.zeros(self.n_layers, self.hidden_dim)
26       return hidden
```

Listado 3.13: Implementación de una RNN.

Al igual que en el caso previo, no podemos usar *nn.Sequential* debido a las características particulares de las redes neuronales recurrentes, así que crearemos de nuevo una clase modelo con las funciones necesarias para poder realizar el entrenamiento y evaluación.

También implementaremos la función *init_hidden*, que crea el primer conjunto de 0s que dan lugar al estado inicial del modelo, y que se irá modificando conforme se realice el aprendizaje del mismo.

El código para el entrenamiento y la evaluación usando el modelo se encuentra en

la figura 3.14.

```
def run(variables, labels, labelDictionary):
    neurons_in = len(variables[0])
    neurons_out = len(labelDictionary)
    neurons_hidden = int((2*neurons_in/3) + neurons_out)
    hidden_layers_amount = 1
    nFolds = 5
    kf = StratifiedKFold(n_splits=nFolds)
    epochs = 3
    accumulatedAccuracy = 0

    folds = enumerate(kf.split(variables,labels))

    for i, (train, test) in folds:
        model = Model(neurons_in, neurons_out, neurons_hidden, hidden_layers_amount)
        loss_criterion = nn.CrossEntropyLoss()
        optimizer = torch.optim.SGD(model.parameters(),
                                    lr=0.2)

        X_train = []
        y_train = []
        X_test = []
        y_test = []
        for element in train:
           X_train.append(variables[element])
           y_train.append(labels[element])
        for element in test:
           X_test.append(variables[element])
           y_test.append(labels[element])

        X_train = torch.tensor(X_train)
        X_test = torch.tensor(X_test)
        y_train = torch.tensor(y_train)
        y_test = torch.tensor(y_test)

        model.train()
        for epoch in range(epochs):
            optimizer.zero_grad()
            print("Empezando época " + str(epoch))
            train_outputs, hidden = model(X_train) # model outputs raw logits
            print("\tCalculando pérdida")
            loss = loss_criterion(train_outputs, y_train)
            print("\tAplicando backward propagation")
            loss.backward()
            print("\tActualizando optimizador")
            optimizer.step()
            print("Terminando época " + str(epoch))

        print("Comenzando evaluación.")
        model.eval()
        with torch.inference_mode():
            test_logits, hidden = model(X_test)
            test_pred = torch.softmax(test_logits, dim=1).argmax(dim=1)
            print("Fold " + str(i) + ": \n")
            accuracy, class_total, class_correct = checkAccuracy(test_pred, y_test)
        print(class_total)
        print(class_correct)
        print(labelDictionary)
```

```
        for i in range(len(labelDictionary)):
            print("Accuracy of " + str(i) + ": " +  str(float(round(100 * class_correct[i
                ] / class_total[i], 4))))
        matrix = confusionMatrix(labelDictionary, test_pred, y_test)
        for i in range(len(labelDictionary)):
            print(matrix[i])

        accumulatedAccuracy += accuracy

    print("Precisión promedio del modelo: " + str(float(accumulatedAccuracy / nFolds)))
```

Listado 3.14: Entrenamiento y evaluación con una RNN.

En este caso, la red neuronal recurrente se basa en una única capa recurrente, pero es un parámetro que se puede modificar y que queda definido en la variable *hidden_layers_amount*. Por lo demás, el funcionamiento del entrenamiento y la evaluación usando el modelo es análogo a los demás descritos, con la excepción de que la evaluación y el entrenamiento también devuelven el estado oculto.

3.3.6. Transformers

Por último, mostraremos el código para la implementación de los *transformers*, como se puede ver en el listado 3.15.

```
def run(variables, labels, labelDictionary):
    nFolds = 5
    kf = StratifiedKFold(n_splits=nFolds)
    accumulatedAccuracy = 0

    numLabels = len(labelDictionary)
    stringVariable = []
    for example in variables:
        dataString = ""
        for element in example:
            dataString += str(element)
        stringVariable.append(dataString)

    data = pd.DataFrame(stringVariable, columns=["Features"])
    data["labels"] = labels
    print(data.describe())
    t = data.labels
    folds = enumerate(kf.split(np.zeros(len(t)), t))

    for i, (train, test) in folds:
        train_set = data.iloc[train]
        test_set = data.iloc[test]

        model_args = ClassificationArgs(num_train_epochs=1, overwrite_output_dir=True)
        model = ClassificationModel(
        "bert", "bert-base-cased", num_labels=numLabels,
        args=model_args, use_cuda=True)
```

```
28
29          model.train_model(train_set)
30
31          test_variables = []
32          test_labels = []
33          datalist = test_set.values.tolist()
34          for element in datalist:
35              test_variables.append(element[0])
36              test_labels.append(element[1])
37          test_labels = torch.tensor(test_labels)
38
39          result, model_outputs = model.predict(test_variables)
40          accuracy, class_total, class_correct = checkAccuracy(result, test_labels)
41          print(class_total)
42          print(class_correct)
43          print(labelDictionary)
44          for i in range(len(labelDictionary)):
45              print("Accuracy␣of␣" + str(i) + ":␣" +  str(float(round(100 * class_correct[i
                    ] / class_total[i], 4))))
46          matrix = confusionMatrix(labelDictionary, result, test_labels)
47          for i in range(len(labelDictionary)):
48            print(matrix[i])
49
50          accumulatedAccuracy += accuracy
51
52      print("Precisión␣promedio␣del␣modelo:␣" + str(float(accumulatedAccuracy / nFolds)))
```

Listado 3.15: Implementación, entrenamiento y evaluación de los *transformer*.

Como se puede apreciar, el modelo de los *transformers* es considerablemente distinto al resto. Esto es debido a que se utilizó la librería *simpleTransformers*, que permite una implementación sencilla de los *transformers* proporcionados por *HuggingFace* [48] en código de *PyTorch*. Como se verá en el apartado 3.4, se hicieron diversas pruebas con los *transformers* hasta encontrar el que mejor resultados obtenía usando el modelo de BERT para, posteriormente, entrenarlo con nuestros datos.

3.4. Marco experimental

En este apartado se detallarán elementos a tener en cuenta para considerar la experimentación llevada a cabo, incluyendo las características del equipo donde se ejecutaron la mayoría de las pruebas, los parámetros usados en cada modelo a la hora de definirlo y entrenarlo, y una breve descripción del proceso seguido para la experimentación.

3.4.1. Características del equipo

El proceso de experimentación se ha llevado a cabo en un ordenador portátil personal. Los detalles de las especificaciones del ordenador relevantes para la experimentación se muestran en la tabla 3.6.

Componente	**Modelo**
Sistema Operativo	Windows 10 Home
Procesador	Intel Core i7-10875h CPU @ 2.30GH
Memoria RAM	32 GB
Disco Duro	SSD 1 TB
Tarjeta Gráfica	NVIDIA GeForce RTX 2070

Tabla. 3.6: Características del equipo empleado para la experimentación.

A continuación, se añadirá información adicional que se considera relevante a la hora de la experimentación. Para comenzar, el procesador empleado (Intel Core i7-10875h CPU) presenta un total de 8 núcleos, cada uno de los cuales es capaz de soportar dos hilos, lo que permite ejecutar simultáneamente 16 subprocesos en paralelo.

Con respecto a la tarjeta gráfica, la GeForce RTX 2070 de NVIDIA cuenta con una memoria de 8 GB. Sin embargo, lo más destacable de la tarjeta es que presenta una serie de núcleos de tensor (*tensor cores*), que funcionan de manera particularmente eficiente a la hora de trabajar con tensores, facilitando las tareas de aprendizaje y evaluación de los modelos sobre los que se experimenta gracias a la aplicación de herramientas como CUDA. Concretamente, la tarjeta cuenta con 2.304 núcleos CUDA FP32, además de 288 núcleos tensoriales.

Esta es la única GPU que presenta la máquina. Con una GPU con mayor cantidad de núcleos compatibles, o, en su defecto, un mayor número de GPUs en la máquina, los procesos podrían paralelizarse más aún, reduciendo en consecuencia el tiempo total. Sin embargo, con solo una tarjeta, el entrenamiento y evaluación de los modelos se realiza de manera relativamente rápida, rara vez tardando más de entre cinco y diez minutos en función del modelo.

3.4.2. Parámetros usados en la experimentación

A la hora de trabajar con modelos de clasificación, y particularmente con modelos profundos, hay una serie de hiperparámetros que se deben definir de cara a asegurar el buen funcionamiento del modelo, tales como la ratio de aprendizaje, el número de *epochs* consideradas durante el entrenamiento, y algunos otros parámetros en función del modelo en concreto (que aparecen en la columna titulada *Otros comentarios*).

Si bien se ha hecho una experimentación exhaustiva, probando diversos valores de los parámetros a lo largo de varias pruebas, a continuación se muestra un listado de todos los parámetros empleados en **la mejor ejecución lograda** para cada modelo, representado en la tabla 3.7.

Cabe destacar que, con objetivo de mantener una cierta constancia a la hora de la experimentación, cada capa oculta tiene siempre un número de neuronas equivalente a $(2/3 * e) + s$, donde e es el número de atributos que tiene cada ejemplo y s es el número de clases posibles de salida.

Modelo	**Número de *epochs***	**Learning Rate**	**Otros comentarios**
RandomForest	N/A	N/A	Número de árboles: 100
Perceptrón Multicapa	20	0.01	Número de capas ocultas: 7
CNN	3	0.0001	Tamaño del *kernel*: 1; Número de capas de *pooling*: 1; Número de capas de *dropout*: 2 (0.3 y 0.5, respectivamente)
RNN	3	0.2	Número de capas recurrentes: 1
Transformers	1	0.00004	Modelo basado en BERT

Tabla. 3.7: Parámetros empleados en la mejor ejecución de cada modelo.

3.4.3. Descripción de la experimentación

En este apartado del capítulo, se hará una breve introducción al proceso de experimentación seguido. La experimentación fue llevada a cabo en un total de cinco fases, cada una de ellas separada en dos etapas. Cada fase se corresponde con uno de los modelos a comprobar, y cada etapa consiste a su vez en una de las dos variantes del problema a resolver que se ha presentado (*BinaryProblem* para distinguir entre tráfico benigno o malicioso o *MaliciousOnly* para distinguir entre distintos tipos de ataques).

Adicionalmente, para cada una de las etapas, se hacía la comprobación sobre cada uno de los tres *datasets* descritos en el apartado 3.2, alternando los valores de los parámetros especificados en la tabla 3.7 para encontrar la combinación que ofreciera los mejores resultados. En todos los casos, la mejor combinación para un dataset suele serlo también para el resto de *datasets*, hecho posiblemente debido a que los datos presentan formatos y estructuras muy similares en todos los conjuntos de datos empleados.

Para la experimentación como tal, únicamente era necesario el determinar el *dataset* sobre el que se iba a trabajar y el modo de trabajo de los dos previamente descritos mediante un archivo de configuración y ejecutar el código en *Python* correspondiente. El propio programa invocaba las funciones que mostraban los resultados y estos se guardaban en un archivo de salida (definido también en el fichero de configuración), por lo que bastaba con ejecutar el programa y esperar para obtener los mismos, modificando los parámetros tras cada ejecución para tratar de mejorarlos.

Para la ejecución del programa propiamente dicho, basta con ejecutar el comando *python main.py config.txt*, siendo config.txt el fichero de configuración en el que deben aparecer los siguientes datos en formato campo=valor:

- **rOutput**: Ruta de salida. Se creará un fichero en la ruta especificada donde se volcarán los resultados de la ejecución.
- **dataFile**: Ruta al fichero correspondiente al conjunto de datos que emplear.
- **executionMode**: Modo de ejecución del programa. Las opciones permitidas son *BinaryProblem, MaliciousOnly* o *ClassicProblem*.
- **targetModel**: Modelo a emplear en la ejecución del programa. Las opciones permitidas son *RandomForest*, PM (para el perceptrón multicapa), CNN, RNN o *Transformer*.

Capítulo 4

RESULTADOS

En este capítulo, se mostrarán los resultados de la experimentación llevada a cabo según lo especificado en el apartado 3.4. Primero, se hará una breve introducción al proceso de evaluación propiamente dicho, explicando los dos modos posibles de uso y las métricas de evaluación empleadas.

Posteriormente se mostrarán los resultados del proceso, separados por modo de ejecución, incluyendo en cada caso los resultados de la mejor ejecución obtenida a lo largo del proceso de experimentación para cada modelo y conjunto de datos. Adicionalmente, en algunos casos, se mostrará también la matriz de confusión correspondiente para poder observar cuándo se cometen errores.

Por último, se realizará un análisis de los resultados obtenidos con objetivo de determinar cuál es el modelo ideal para este problema, además de algunos comentarios adicionales sobre las diversas ejecuciones.

4.1. Evaluación

En esta primera sección se explicará con mayor detenimiento los dos modos de trabajo que se han seguido durante la experimentación, de manera que sirva como introducción para el apartado 4.2, en el que se mostrarán los resultados para cada uno de estos modos de ejecución. También se mencionan brevemente las métricas empleadas a la hora de evaluar un modelo, diferenciando en las que cobran más importancia en un modo de trabajo que en otro y cuáles son consideradas para ambos.

4.1.1. Problema binario

Este tipo de problema se corresponde con el modo de ejecución *BinaryProblem* descrito en el apartado 3.3.1. Para cada conjunto de datos, se observan todos los ejemplos y se transforman los datos para que únicamente existan dos clases: *Benign* y *Malicious*. De esta manera, se entrena al modelo para que sea capaz de distinguir entre tráfico benigno y todos los tipos de tráfico malicioso.

Dado que es más fácil resolver un problema binario que uno de clasificación multiclase, es habitual que los resultados sean generalmente mejores con este modo de ejecución, aunque dependa también del modelo empleado y del conjunto de datos analizado.

Sin embargo, también era más habitual que se diera sobreajuste en este problema, ya que las clases suelen estar desbalanceadas (habitualmente existiendo más ejemplos de tráfico benigno que malicioso, pero no necesariamente).

4.1.2. Tráfico malicioso

Este tipo de problema, también descrito en el apartado 3.3.1, se corresponde con el modelo de ejecución *MaliciousOnly*. Al contrario que en el caso previo, en este caso se ignoran todos los ejemplos cuya clase sea *benign*, dejando únicamente los ejemplos que representan ataques, y no se modifican los atributos de estos más allá de ninguna forma.

El resultado es obtener diversos ejemplos de cada tipo de ataque recogido en el conjunto de datos, con objetivo de que los modelos sean capaces de aprender a diferenciar entre diversos tipos de ataques con el propósito final de identificar, para una conexión cualquiera que se sabe maligna, el tipo de ataque al que se corresponde la conexión, con el posterior objetivo de poder proteger el sistema ante el ataque o mitigar lo máximo posible el daño que pudiera llegar a causar.

El problema principal de este modo de ejecución son las clases muy minoritarias, que a menudo tienen un porcentaje de acierto muy bajo por la muy pequeña cantidad de ejemplos que se tienen de ellas. En base al modelo, también se puede dar sobreajuste a las clases mayoritarias, aunque es menos frecuente que con el problema binario explicado en el apartado previo.

4.1.3. Métricas de evaluación

En este apartado, se explicarán las métricas de evaluación empleadas a la hora de determinar cuál de los modelos presentados es más adecuado para el tratamiento de los datos.

Las métricas consideradas son las siguientes:

- **Precisión**: la precisión se define como el porcentaje de casos en los que el modelo es capaz de predecir satisfactoriamente la clase de un ejemplo determinado. Naturalmente, es de nuestro interés que la precisión sea lo más alta posible, pues se corresponde con una mayor capacidad del modelo de predecir cuándo una conexión es o no maligna y el tipo de ataque, en caso de que lo sea. Es la principal medida considerada independientemente del modo de trabajo.

- ***Recall***: debido a las circunstancias particulares del problema, en el caso de *BinaryProblem*, es muy importante el determinar si una conexión es o no un ataque. Resulta más peligroso calificar como benigno un ataque que el caso contrario, por lo que utilizaremos el *recall*, métrica definida como la ratio de verdaderos positivos, para determinar la calidad del modelo en ese tipo de problemas.

- **Tiempo de ejecución**: en caso de que se diera un ataque, nos interesa que el tiempo de ejecución del modelo fuera lo menor posible. Siendo modelos de aprendizaje profundo en su mayoría, el tiempo de entrenamiento suele ser considerablemente mayor que el tiempo necesario para evaluar uno o más ejemplos de conexiones, pero es una medida que igualmente se considerará, particularmente por la facilidad de uso (es más sencillo instalar un modelo en 3 minutos que uno que tarda 3 horas en entrenarse, aunque sea algo que solo se debe hacer una vez). Esto se debe a que los ataques y conexiones varían constantemente, por lo que es probable que se deba reentrenar el modelo si se reciben nuevos datos.

- **Claridad**: si bien no es una métrica en el sentido estricto de la palabra, también resulta importante que el modelo sea fácilmente comprensible por alguien que pueda observar los resultados. En caso de que dos modelos ofrezcan resultados similares, se considerará mejor aquel que permita saber la procedencia de esos resultados, ya sea de manera visual (como mediante árboles), porque pueda ser convertido en reglas o asociaciones de atributos, etc.

4.2. Resultados

En esta sección se muestran los resultados del proceso de experimentación llevado a cabo para cada modelo mencionado. Cada subsección posterior se centrará en uno de los modos de ejecución mencionados, mostrando los resultados de cada modelo en el mismo.

4.2.1. Problema binario

A continuación, se muestran los resultados proporcionados por todos los modelos para el modo de ejecución *BinaryProblem*. Cada conjunto de datos aparece en su respectiva tabla para facilitar la lectura:

- **CIC-IDS2017**: Tabla 4.1.
- **CIC-CSE-IDS2018**: Tabla 4.2.
- **CIC-DDoS2019**: Tabla 4.3.

Modelo	Porcentaje de acierto	Aciertos / total
RandomForest	0.9987	459810/460165
Perceptrón Multicapa	0.6882	316681/460165
CNN	0.7576	348636/460165
RNN	0.7954	366027/460165
Transformer	0.7576	348636/460165

Tabla. 4.1: Resultados de los varios modelos en el modo de ejecución *BinaryProblem* sobre el conjunto de datos de CIC-IDS2017.

Modelo	Porcentaje de acierto	Aciertos / total
RandomForest	0.9999	619367/619430
Perceptrón Multicapa	0.6372	392249/619430
CNN	0.6814	422071/619430
RNN	0.7714	477807/619430
Transformer	0.6814	422071/619430

Tabla. 4.2: Resultados de los varios modelos en el modo de ejecución *BinaryProblem* sobre el conjunto de datos de CIC-CSE-IDS2018.

Modelo	Porcentaje de acierto	Aciertos / total
RandomForest	0.9993	86216/86275
Perceptrón Multicapa	0.9348	80648/86275
CNN	0.7732	66709/86275
RNN	0.8827	76157/86275
Transformer	0.7749	66709/86275

Tabla. 4.3: Resultados de los varios modelos en el modo de ejecución *BinaryProblem* sobre el conjunto de datos de CIC-DDoS2019.

Como podemos apreciar, para este primer modo de ejecución, el modelo *RandomForest* obtiene resultados mejores que el resto de modelos con diferencia, independientemente del conjunto de datos empleado. El resto de modelos, si bien logran alcanzar buenos resultados en algún caso (tanto el perceptrón multicapa como la RNN alcanzan alrededor de un 90 % de acierto en el CIC-DDoS2019, como se aprecia en la tabla 4.3), siguen sin alcanzar los resultados de *RandomForest*.

4.2.2. Solo tráfico malicioso

En este apartado se presentan los resultados correspondientes al modo de ejecución *MaliciousOnly*. Al igual que en el caso previo, los resultados aparecen separados en base al conjunto de datos:

- **CIC-IDS2017**: Tabla 4.4.
- **CIC-CSE-IDS2018**: Tabla 4.5.
- **CIC-DDoS2019**: Tabla 4.6.

Modelo	Porcentaje de acierto	Aciertos / total
RandomForest	0.9982	25606/111529
Perceptrón Multicapa	0.2296	25606/111529
CNN	0.2296	25606/111529
RNN	0.7843	86975/111529
Transformer	0.9977	111277/111529

Tabla. 4.4: Resultados de los varios modelos en el modo de ejecución *MaliciousOnly* sobre el conjunto de datos de CIC-IDS2017.

Modelo	Porcentaje de acierto	Aciertos / total
RandomForest	0.9352	184570/197359
Perceptrón Multicapa	0.1959	38672/197359
CNN	0.1959	38672/197359
RNN	0.7485	147724/197359
Transformer	0.9480	187105/197359

Tabla. 4.5: Resultados de los varios modelos en el modo de ejecución *MaliciousOnly* sobre el conjunto de datos de CIC-CSE-IDS2018.

Modelo	Porcentaje de acierto	Aciertos / total
RandomForest	0.9061	60444/66709
Perceptrón Multicapa	0.0264	1766/66709
CNN	0.0542	3618/66709
RNN	0.7666	51142/66709
Transformer	0.9166	61143/66709

Tabla. 4.6: Resultados de los varios modelos en el modo de ejecución *MaliciousOnly* sobre el conjunto de datos de CIC-DDoS2019.

En este caso concreto, podemos apreciar que, si bien el modelo *RandomForest* sigue proporcionando buenos resultados, es superado por el modelo *transformer* en los conjuntos de datos CIC-CSE-IDS2018 y CIC-DDoS2019. Si bien apenas es superado por alrededor de un 2 %, esto se traduce en cerca de 3000 aciertos más en el primer caso y alrededor de 1500 en el segundo.

Con objetivo de investigar más a fondo todos los modelos empleados, a continuación, se hará una vista pormenorizada de cada uno de ellos, mostrando los porcentajes

de acierto de cada clase individual, así como las matrices de confusión correspondientes.

RandomForest

A continuación, se muestran los resultados detallados del modelo *RandomForest*, incluyendo la precisión relativa a cada clase y las matrices de confusión correspondientes:

- **CIC-IDS2017**: Tablas 4.7 y 4.8.
- **CIC-CSE-IDS2018**: Tablas 4.9 y 4.10.
- **CIC-DDoS2019**: Tablas 4.11 y 4.12.

	Porcentaje de acierto	**Aciertos / total**
Resultados generales	0.9982	25606/111529
DDoS	1	25606/25606
Portscan	0.9996	31777/31789
Bot	1	393/393
Infiltration	0.8571	6/7
WebAttack Bruteforce	0.7807	236/299
WebAttack XSS	0.3231	43/130
WebAttack SQL Injection	0.7500	3/4
FTP Patator	1	1587/1587
SSH Patator	1	1180/1180
Slowloris	0.9931	1152/1159
SlowHttpTest	0.9864	1086/1099
DoSHulk	0.9998	46211/46215
GoldenEye	0.9961	2052/2059
Heartbleed	1	2/2

Tabla. 4.7: Resultados del modelo *RandomForest* en el conjunto de datos CIC-IDS2017 en el modo de ejecución *MaliciousOnly*.

	DDoS	Portscan	Bot	Infiltration	Webattack Bruteforce	Webattack XSS	Webattack SQL Injection	FTP Patator	SSH Patator	Slowloris	Slowhttptest	DoSHulk	Goldeneye	Heartbleed
DDoS	25606	0	0	0	0	0	0	0	0	0	0	0	0	0
Portscan	0	31777	0	0	1	1	0	0	0	0	0	6	1	0
Bot	0	0	393	0	0	0	0	0	0	0	0	0	0	0
Infiltration	0	0	0	6	0	0	0	0	0	0	0	1	0	0
Webattack Bruteforce	0	2	0	0	236	62	0	0	0	1	0	0	0	0
Webattack XSS	0	0	0	0	86	43	0	0	0	0	0	1	0	0
Webattack SQL Injection	0	0	0	0	1	0	3	0	0	0	0	0	0	0
FTP Patator	0	0	0	0	0	0	0	1587	0	0	0	0	0	0
SSH Patator	0	0	0	0	0	0	0	0	1180	0	0	0	0	0
Slowloris	0	1	0	0	2	0	0	0	0	1152	3	0	1	0
Slowhttptest	0	0	0	0	0	1	0	0	0	9	1086	1	3	0
DoSHulk	0	1	0	0	0	0	0	0	0	0	0	46211	3	0
Goldeneye	0	0	0	0	0	0	0	0	0	0	1	6	2052	0
Heartbleed	0	0	0	0	0	0	0	0	0	0	0	0	0	2

Tabla. 4.8: Matriz de confusión del modelo *RandomForest* en el conjunto de datos CIC-IDS2017 en el modo de ejecución *MaliciousOnly*.

	Porcentaje de acierto	**Aciertos / total**
Resultados generales	0.9352	184570/197359
FTP Bruteforce	0.9177	35395/38672
SSH Bruteforce	0.9999	37514/37518
Goldeneye	0.9995	8301/8302
Slowloris	0.9995	2197/2198
SlowHttp	0.4710	8525/18287
DoSHulk	0.9999	92381/92382

Tabla. 4.9: Resultados del modelo *RandomForest* en el conjunto de datos CIC-CSE-IDS2018 en el modo de ejecución *MaliciousOnly*.

	FTP-Bruteforce	SSH-Bruteforce	GoldenEye	Slowloris	SlowHttp	DoS Hulk
FTP-Bruteforce	35395	0	0	0	3277	0
SSH-Bruteforce	2	37514	0	0	2	0
GoldenEye	0	0	8301	0	0	1
Slowloris	0	0	0	2197	0	1
SlowHttp	9762	0	0	0	8525	0
DosHulk	0	0	1	0	0	92381

Tabla. 4.10: Matriz de confusión del modelo *RandomForest* en el conjunto de datos CIC-CSE-IDS2018 en el modo de ejecución *MaliciousOnly*.

	Porcentaje de acierto	**Aciertos / total**
Resultados generales	0.9061	60444/66709
UDP	0.6918	2503/3618
MS SQL	0.4633	790/1705
Portmap	0.3649	50/137
Syn	0.9915	9792/9875
Netbios	0.3333	43/129
UDPLag	0.7619	1360/1785
LDAP	0.2230	85/381
DrDoS DNS	0.4523	332/734
DrDoS WebDDoS	0	0/10
DrDoS TFTP	0.9974	19732/19783
DrDoS UDP	0.3680	767/2084
DrDoS SNMP	0.6316	343/543
DrDoS NetBIOS	0.0500	6/120
DrDoS LDAP	0.1563	45/288
DrDoS MSSQL	0.2809	349/1242
DrDoS NTP	0.9988	24247/24274

Tabla. 4.11: Resultados del modelo *RandomForest* en el conjunto de datos CIC-DDoS2019 en el modo de ejecución *MaliciousOnly*.

	UDP	MS SQL	Portmap	Syn	Netbios	UDPlag	LDAP	DrDoS DNS	DrDoS WebD-DoS	DrDoS TFTP	DrDoS UDP	DrDoS SNMP	DrDoS Net-BIOS	DrDoS LDAP	DrDoS MSSQL	DrDoS NTP
UDP	2504	41	2	3	0	133	0	4	0	1	909	0	2	0	19	0
MS SQL	23	791	1	1	1	2	6	24	0	1	12	22	12	3	799	7
Portmap	0	0	50	8	59	1	0	1	0	0	0	0	13	0	4	1
Syn	3	3	7	9793	0	55	0	1	0	6	1	0	1	0	0	5
Netbios	1	0	44	1	44	0	0	2	0	0	0	7	30	0	0	0
UDPlag	246	4	1	45	0	1360	0	3	0	19	98	4	0	2	0	2
LDAP	0	5	1	0	0	2	85	101	0	0	1	58	0	122	6	0
DrDoS DNS	6	67	4	0	5	1	92	333	0	2	7	52	2	75	73	15
DrDoS WebD-DoS	0	0	0	0	0	0	0	0	0	2	0	0	0	0	0	8
DrDoS TFTP	0	0	1	2	0	34	0	0	0	19732	2	0	0	0	2	10
DrDoS UDP	1222	15	1	2	0	60	1	4	0	1	768	0	1	0	8	1
DrDoS SNMP	0	21	1	0	18	0	60	52	0	0	0	344	4	36	17	0
DrDoS Netbios	1	14	17	0	63	0	0	1	0	0	1	1	6	0	16	0
DrDoS LDAP	0	2	0	0	0	0	118	72	0	0	0	44	0	45	4	3
DrDoS MSSQL	13	795	3	2	0	0	10	28	0	1	10	17	6	4	349	4
DrDoS NTP	0	3	0	5	0	2	0	1	5	6	1	1	0	0	2	24248

Tabla. 4.12: Matriz de confusión del modelo *RandomForest* en el conjunto de datos CIC-DDoS2019 en el modo de ejecución *MaliciousOnly*.

Como se puede apreciar, el modelo *RandomForest* ofrece resultados favorables en todos los casos, si bien tiene algunos problemas a la hora de analizar las clases minoritarias. Sin embargo, es capaz de asignar correctamente algunos ejemplos de estas salvo en casos muy extremos.

Perceptrón multicapa

En este apartado se muestran los resultados pormenorizados del perceptrón multicapa. Al igual que en casos previos, la información aparece recogida en tablas para facilitar su lectura:

- **CIC-IDS2017**: Tablas 4.13 y 4.14.
- **CIC-CSE-IDS2018**: Tablas 4.15 y 4.16.
- **CIC-DDoS2019**: Tablas 4.17 y 4.18.

	Porcentaje de acierto	**Aciertos / total**
Resultados generales	0.2296	25606/111529
DDoS	1	25606/25606
Portscan	0	0/31789
Bot	0	0/393
Infiltration	0	0/7
WebAttack Bruteforce	0	0/299
WebAttack XSS	0	0/130
WebAttack SQL Injection	0	0/4
FTP Patator	0	0/1587
SSH Patator	0	0/1180
Slowloris	0	0/1159
SlowHttpTest	0	0/1099
DoSHulk	0	0/46215
GoldenEye	0	0/2059
Heartbleed	0	0/2

Tabla. 4.13: Resultados del perceptrón multicapa en el conjunto de datos CIC-IDS2017 en el modo de ejecución *MaliciousOnly*.

	DDoS	Portscan	Bot	Infiltration	Webattack Bruteforce	Webattack XSS	Webattack SQL Injection	FTP Patator	SSH Patator	Slowloris	Slowhttptest	DoSHulk	Goldeneye	Heartbleed
DDoS	25606	0	0	0	0	0	0	0	0	0	0	0	0	0
Portscan	31789	0	0	0	0	0	0	0	0	0	0	0	0	0
Bot	393	0	0	0	0	0	0	0	0	0	0	0	0	0
Infiltration	299	0	0	0	0	0	0	0	0	0	0	0	0	0
Webattack Bruteforce	299	0	0	0	0	0	0	0	0	0	0	0	0	0
Webattack XSS	130	0	0	0	0	0	0	0	0	0	0	0	0	0
Webattack SQL Injection	4	0	0	0	0	0	0	0	0	0	0	0	0	0
FTP Patator	1587	0	0	0	0	0	0	0	0	0	0	0	0	0
SSH Patator	1180	0	0	0	0	0	0	0	0	0	0	0	0	0
Slowloris	1159	0	0	0	0	0	0	0	0	0	0	0	0	0
Slowhttptest	1099	0	0	0	0	0	0	0	0	0	0	0	0	0
DoSHulk	46215	0	0	0	0	0	0	0	0	0	0	0	0	0
Goldeneye	2059	0	0	0	0	0	0	0	0	0	0	0	0	0
Heartbleed	2	0	0	0	0	0	0	0	0	0	0	0	0	0

Tabla. 4.14: Matriz de confusión del perceptrón multicapa en el conjunto de datos CIC-IDS2017 en el modo de ejecución *MaliciousOnly*.

	Porcentaje de acierto	**Aciertos / total**
Resultados generales	0.1959	38672/197359
FTP Bruteforce	1	38672/38672
SSH Bruteforce	0	0/37518
Goldeneye	0	0/8302
Slowloris	0	0/2198
SlowHttp	0	0/18287
DoSHulk	0	0/92382

Tabla. 4.15: Resultados del perceptrón multicapa en el conjunto de datos CIC-CSE-IDS2018 en el modo de ejecución *MaliciousOnly*.

	FTP-Bruteforce	SSH-Bruteforce	GoldenEye	Slowloris	SlowHttp	DoS Hulk
FTP-Bruteforce	38672	0	0	0	0	0
SSH-Bruteforce	37518	0	0	0	0	0
GoldenEye	8302	0	0	0	0	0
Slowloris	2198	0	0	0	0	0
SlowHttp	18287	0	0	0	0	0
DosHulk	92382	0	0	0	0	0

Tabla. 4.16: Matriz de confusión del perceptrón multicapa en el conjunto de datos CIC-CSE-IDS2018 en el modo de ejecución *MaliciousOnly*.

	Porcentaje de acierto	**Aciertos / total**
Resultados generales	0.0264	1766/66709
UDP	0	0/3618
MS SQL	0	0/1705
Portmap	0	0/137
Syn	0	0/9875
Netbios	0	0/129
UDPLag	0.9893	1766/1785
LDAP	0	0/381
DrDoS DNS	0	0/734
DrDoS WebDDoS	0	0/10
DrDoS TFTP	0	0/19783
DrDoS UDP	0	0/2084
DrDoS SNMP	0	0/543
DrDoS NetBIOS	0	0/120
DrDoS LDAP	0	0/288
DrDoS MSSQL	0	0/1242
DrDoS NTP	0	0/24274

Tabla. 4.17: Resultados del perceptrón multicapa en el conjunto de datos CIC-DDoS2019 en el modo de ejecución *MaliciousOnly*.

	UDP	MS SQL	Portmap	Syn	Netbios	UDPlag	LDAP	DrDoS DNS	DrDoS WebD-DoS	DrDoS TFTP	DrDoS UDP	DrDoS SNMP	DrDoS Net-BIOS	DrDoS LDAP	DrDoS MSSQL	DrDoS NTP
UDP	0	0	0	0	0	3618	0	0	0	0	0	0	0	0	0	0
MS SQL	0	0	0	0	0	1704	0	0	0	0	1	0	0	0	0	0
Portmap	0	0	0	0	0	136	0	0	0	0	1	0	0	0	0	0
Syn	0	0	0	0	0	9851	0	15	0	0	9	0	0	0	0	0
Netbios	0	0	0	0	0	129	0	0	0	0	0	0	0	0	0	0
UDPlag	0	0	0	0	0	1766	0	17	0	0	2	0	0	0	0	0
LDAP	0	0	0	0	0	381	0	0	0	0	0	0	0	0	0	0
DrDoS DNS	0	0	0	0	0	734	0	0	0	0	0	0	0	0	0	0
DrDoS WebD-DoS	0	0	2	0	0	3	0	1	0	0	4	0	0	0	0	0
DrDoS TFTP	0	0	1	0	0	19770	0	7	0	0	5	0	0	0	0	0
DrDoS UDP	0	0	0	0	0	2084	0	0	0	0	0	0	0	0	0	0
DrDoS SNMP	0	0	0	0	0	543	0	0	0	0	0	0	0	0	0	0
DrDoS Netbios	0	0	0	0	0	120	0	0	0	0	0	0	0	0	0	0
DrDoS LDAP	0	0	0	0	0	288	0	0	0	0	0	0	0	0	0	0
DrDoS MSSQL	0	0	0	0	0	1242	0	0	0	0	0	0	0	0	0	0
DrDoS NTP	0	0	6	0	0	24239	0	5	0	0	24	0	0	0	0	0

Tabla. 4.18: Matriz de confusión del perceptrón multicapa en el conjunto de datos CIC-DDoS2019 en el modo de ejecución *MaliciousOnly*.

Al contrario que con el *RandomForest*, el perceptrón multicapa presenta resultados muy negativos. Esto es debido a que, en todo caso, elige una clase determinada y asigna todos los ejemplos a esa clase. Habitualmente, la clase elegida suele ser la más mayoritaria, aunque en determinados modelos, particularmente en el CIC-DDoS2019, ni siquiera es el caso. Como consecuencia, su porcentaje de acierto es extremadamente bajo.

CNN

En este apartado se muestran los detalles del modelo CNN. A continuación, se presentan las tablas correspondientes a la precisión de cada clase y las correspondientes matrices de confusión:

- **CIC-IDS2017**: Tablas 4.19 y 4.20.
- **CIC-CSE-IDS2018**: Tablas 4.21 y 4.22.
- **CIC-DDoS2019**: Tablas 4.23 y 4.24.

	Porcentaje de acierto	Aciertos / total
Resultados generales	0.2296	25606/111529
DDoS	1	25606/25606
Portscan	0	0/31789
Bot	0	0/393
Infiltration	0	0/7
WebAttack Bruteforce	0	0/299
WebAttack XSS	0	0/130
WebAttack SQL Injection	0	0/4
FTP Patator	0	0/1587
SSH Patator	0	0/1180
Slowloris	0	0/1159
SlowHttpTest	0	0/1099
DoSHulk	0	0/46215
GoldenEye	0	0/2059
Heartbleed	0	0/2

Tabla. 4.19: Resultados del modelo CNN en el conjunto de datos CIC-IDS2017 en el modo de ejecución *MaliciousOnly*.

	DDoS	Portscan	Bot	Infiltration	Webattack Bruteforce	Webattack XSS	Webattack SQL Injection	FTP Patator	SSH Patator	Slowloris	Slowhttptest	DoSHulk	Goldeneye	Heartbleed
DDoS	25606	0	0	0	0	0	0	0	0	0	0	0	0	0
Portscan	31789	0	0	0	0	0	0	0	0	0	0	0	0	0
Bot	393	0	0	0	0	0	0	0	0	0	0	0	0	0
Infiltration	299	0	0	0	0	0	0	0	0	0	0	0	0	0
Webattack Bruteforce	299	0	0	0	0	0	0	0	0	0	0	0	0	0
Webattack XSS	130	0	0	0	0	0	0	0	0	0	0	0	0	0
Webattack SQL Injection	4	0	0	0	0	0	0	0	0	0	0	0	0	0
FTP Patator	1587	0	0	0	0	0	0	0	0	0	0	0	0	0
SSH Patator	1180	0	0	0	0	0	0	0	0	0	0	0	0	0
Slowloris	1159	0	0	0	0	0	0	0	0	0	0	0	0	0
Slowhttptest	1099	0	0	0	0	0	0	0	0	0	0	0	0	0
DoSHulk	46215	0	0	0	0	0	0	0	0	0	0	0	0	0
Goldeneye	2059	0	0	0	0	0	0	0	0	0	0	0	0	0
Heartbleed	2	0	0	0	0	0	0	0	0	0	0	0	0	0

Tabla. 4.20: Matriz de confusión del modelo CNN en el conjunto de datos CIC-IDS2017 en el modo de ejecución *MaliciousOnly*.

	Porcentaje de acierto	**Aciertos / total**
Resultados generales	0.1959	38672/197359
FTP Bruteforce	1	38672/38672
SSH Bruteforce	0	0/37518
Goldeneye	0	0/8302
Slowloris	0	0/2198
SlowHttp	0	0/18287
DoSHulk	0	0/92382

Tabla. 4.21: Resultados del modelo CNN en el conjunto de datos CIC-CSE-IDS2018 en el modo de ejecución *MaliciousOnly*.

	FTP-Bruteforce	SSH-Bruteforce	GoldenEye	Slowloris	SlowHttp	DoS Hulk
FTP-Bruteforce	38672	0	0	0	0	0
SSH-Bruteforce	37518	0	0	0	0	0
GoldenEye	8302	0	0	0	0	0
Slowloris	2198	0	0	0	0	0
SlowHttp	18287	0	0	0	0	0
DosHulk	92382	0	0	0	0	0

Tabla. 4.22: Matriz de confusión del modelo CNN en el conjunto de datos CIC-CSE-IDS2018 en el modo de ejecución *MaliciousOnly*.

	Porcentaje de acierto	**Aciertos / total**
Resultados generales	0.0542	3618/66709
UDP	1	3618/3618
MS SQL	0	0/1705
Portmap	0	0/137
Syn	0	0/9875
Netbios	0	0/129
UDPLag	0	0/1785
LDAP	0	0/381
DrDoS DNS	0	0/734
DrDoS WebDDoS	0	0/10
DrDoS TFTP	0	0/19783
DrDoS UDP	0	0/2084
DrDoS SNMP	0	0/543
DrDoS NetBIOS	0	0/120
DrDoS LDAP	0	0/288
DrDoS MSSQL	0	0/1242
DrDoS NTP	0	0/24274

Tabla. 4.23: Resultados del modelo CNN en el conjunto de datos CIC-DDoS2019 en el modo de ejecución *MaliciousOnly*.

	UDP	MS SQL	Portmap	Syn	Netbios	UDPlag	LDAP	DrDoS DNS	DrDoS WebD-DoS	DrDoS TFTP	DrDoS UDP	DrDoS SNMP	DrDoS Net-BIOS	DrDoS LDAP	DrDoS MSSQL	DrDoS NTP
UDP	3618	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
MS SQL	1705	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Portmap	137	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Syn	9875	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Netbios	129	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
UDPlag	1785	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
LDAP	381	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DrDoS DNS	734	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DrDoS WebD-DoS	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DrDoS TFTP	19783	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DrDoS UDP	2084	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DrDoS SNMP	543	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DrDoS Netbios	120	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DrDoS LDAP	288	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DrDoS MSSQL	1242	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DrDoS NTP	24274	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Tabla. 4.24: Matriz de confusión del modelo CNN en el conjunto de datos CIC-DDoS2019 en el modo de ejecución *MaliciousOnly*.

De manera similar a lo que ocurría con el perceptrón multicapa, la CNN también tiende a sobreajustarse a una clase determinada y asignar todos los ejemplos son a la misma. En consecuencia, este modelo tampoco es deseable por exactamente los mismos motivos que el previamente mencionado.

RNN

A continuación, se muestran los resultados detallados del modelo RNN, incluyendo la precisión relativa a cada clase y las matrices de confusión correspondientes:

- **CIC-IDS2017**: Tablas 4.25 y 4.26.
- **CIC-CSE-IDS2018**: Tablas 4.27 y 4.28.
- **CIC-DDoS2019**: Tablas 4.29 y 4.30.

	Porcentaje de acierto	**Aciertos / total**
Resultados generales	0.7843	86975/111529
DDoS	0.4554	11163/25606
Portscan	0.9967	31682/31789
Bot	0	0/393
Infiltration	0	0/7
WebAttack Bruteforce	0	0/299
WebAttack XSS	0	0/130
WebAttack SQL Injection	0	0/4
FTP Patator	0	0/1587
SSH Patator	0	0/1180
Slowloris	0	0/1159
SlowHttpTest	0	0/1099
DoSHulk	0.9548	44130/46215
GoldenEye	0	0/2059
Heartbleed	0	0/2

Tabla. 4.25: Resultados del modelo RNN en el conjunto de datos CIC-IDS2017 en el modo de ejecución *MaliciousOnly*.

	DDoS	Portscan	Bot	Infiltration	Webattack Bruteforce	Webattack XSS	Webattack SQL Injection	FTP Patator	SSH Patator	Slowloris	Slowhttptest	DoSHulk	Goldeneye	Heartbleed
DDoS	11663	26	0	0	0	0	0	0	0	0	0	13917	0	0
Portscan	0	31682	8	0	0	3	0	0	0	0	0	88	5	0
Bot	9	142	0	0	0	0	0	0	0	0	0	242	0	0
Infiltration	1	1	0	0	0	0	0	0	0	0	0	5	0	0
Webattack Bruteforce	0	0	0	0	0	0	0	0	0	0	0	301	0	0
Webattack XSS	0	0	0	0	0	0	0	0	0	0	0	130	0	0
Webattack SQL Injection	0	0	0	0	0	0	0	0	0	0	0	4	0	0
FTP Patator	0	794	0	0	0	0	0	0	0	0	0	794	0	0
SSH Patator	0	0	0	0	0	0	0	0	0	0	0	1179	1	0
Slowloris	349	114	0	0	0	0	0	0	0	0	0	696	0	0
Slowhttptest	605	73	0	0	0	0	0	0	0	0	0	422	0	0
DoSHulk	229	488	5	0	0	0	0	0	0	0	1	44130	7	1335
Goldeneye	6	0	0	0	0	0	0	0	0	0	0	2053	0	0
Heartbleed	0	0	0	0	0	0	0	0	0	0	0	2	0	0

Tabla. 4.26: Matriz de confusión del modelo RNN en el conjunto de datos CIC-IDS2017 en el modo de ejecución *MaliciousOnly*.

	Porcentaje de acierto	**Aciertos / total**
Resultados generales	0.7485	147724/197359
FTP Bruteforce	1	38672/38672
SSH Bruteforce	0.4991	18729/37518
Goldeneye	0	0/8302
Slowloris	0	0/2198
SlowHttp	0	0/18287
DoSHulk	0.9777	90323/92382

Tabla. 4.27: Resultados del modelo RNN en el conjunto de datos CIC-CSE-IDS2018 en el modo de ejecución *MaliciousOnly*.

	FTP-Bruteforce	SSH-Bruteforce	GoldenEye	Slowloris	SlowHttp	DoS Hulk
FTP-Bruteforce	38672	0	0	0	0	0
SSH-Bruteforce	18765	18729	0	0	0	24
GoldenEye	0	4756	0	0	0	3546
Slowloris	0	957	0	0	0	1241
SlowHttp	18287	0	0	0	0	0
DosHulk	0	2059	0	0	0	90323

Tabla. 4.28: Matriz de confusión del modelo RNN en el conjunto de datos CIC-CSE-IDS2018 en el modo de ejecución *MaliciousOnly*.

	Porcentaje de acierto	**Aciertos / total**
Resultados generales	0.7666	51142/66709
UDP	0	0/3618
MS SQL	0	0/1705
Portmap	0	0/137
Syn	0.7736	7640/9875
Netbios	0	0/129
UDPLag	0.0011	2/1785
LDAP	0	0/381
DrDoS DNS	0	0/734
DrDoS WebDDoS	0	0/10
DrDoS TFTP	0.9933	19651/19783
DrDoS UDP	0	0/2084
DrDoS SNMP	0	0/543
DrDoS NetBIOS	0	0/120
DrDoS LDAP	0	0/288
DrDoS MSSQL	0	0/1242
DrDoS NTP	0.9825	23849/24274

Tabla. 4.29: Resultados del modelo RNN en el conjunto de datos CIC-DDoS2019 en el modo de ejecución *MaliciousOnly*.

	UDP	MS SQL	Portmap	Syn	Netbios	UDPlag	LDAP	DrDoS DNS	DrDoS WebD-DoS	DrDoS TFTP	DrDoS UDP	DrDoS SNMP	DrDoS Net-BIOS	DrDoS LDAP	DrDoS MSSQL	DrDoS NTP
UDP	0	0	0	3	0	27	0	0	0	3244	0	0	0	1	0	343
MS SQL	1	0	0	2	0	43	0	0	0	53	0	0	0	0	0	1606
Portmap	0	0	0	1	0	4	0	0	0	17	0	0	0	0	0	115
Syn	6	0	0	7640	0	0	0	0	1	635	0	0	0	2	0	1591
Netbios	0	0	0	2	0	1	0	0	0	5	0	0	0	0	0	121
UDPlag	0	0	0	1131	0	2	0	0	0	583	0	0	0	0	0	69
LDAP	0	0	0	0	0	1	0	0	0	3	0	0	0	0	0	377
DrDoS DNS	0	0	0	2	0	6	0	0	0	18	0	0	0	0	0	708
DrDoS WebD-DoS	0	0	0	2	0	0	0	0	0	8	0	0	0	0	0	0
DrDoS TFTP	1	0	0	29	1	23	0	0	0	19651	0	0	0	0	0	78
DrDoS UDP	0	0	0	1	0	34	0	0	0	1861	0	0	0	0	0	188
DrDoS SNMP	0	0	0	0	0	1	0	0	0	5	0	0	0	0	0	537
DrDoS Netbios	0	0	0	0	0	4	0	0	0	7	0	0	0	0	0	109
DrDoS LDAP	0	0	0	2	0	2	0	0	0	5	0	0	0	0	0	279
DrDoS MSSQL	0	0	0	1	0	38	0	0	0	26	0	0	0	0	0	1177
DrDoS NTP	3	0	0	60	2	34	33	0	0	278	6	1	2	0	0	23849

Tabla. 4.30: Matriz de confusión del modelo RNN en el conjunto de datos CIC-DDoS2019 en el modo de ejecución *MaliciousOnly*.

El modelo RNN ofrece resultados superiores al resto de modelos de DL mostrados hasta ahora. Al contrario que el resto, si bien sobreajusta, es capaz de llevar a cabo predicciones correctas, aunque sea únicamente en el caso de clases mayoritarias. Sin embargo, sigue siendo peor que el modelo *RandomForest* en cualquier caso.

Transformer

Por último, se muestran los resultados pormenorizados del *transformer*. Al igual que en el resto de casos, la información aparece recogida en tablas para facilitar su lectura:

- **CIC-IDS2017**: Tablas 4.31 y 4.32.
- **CIC-CSE-IDS2018**: Tablas 4.33 y 4.34.
- **CIC-DDoS2019**: Tablas 4.35 y 4.36.

	Porcentaje de acierto	**Aciertos / total**
Resultados generales	0.9977	111277/111529
DDoS	0.9993	25588/25606
Portscan	0.9990	31757/31789
Bot	1	393/393
Infiltration	0	0/7
WebAttack Bruteforce	0.9898	296/299
WebAttack XSS	0	0/130
WebAttack SQL Injection	0	0/4
FTP Patator	0.9993	1586/1587
SSH Patator	0.9992	1179/1180
Slowloris	0.9914	1149/1159
SlowHttpTest	0.9887	1087/1099
DoSHulk	0.9997	46201/46215
GoldenEye	0.9914	2041/2059
Heartbleed	0	0/2

Tabla. 4.31: Resultados del modelo *transformer* en el conjunto de datos CIC-IDS2017 en el modo de ejecución *MaliciousOnly*.

	DDoS	Portscan	Bot	Infiltration	Webattack Bruteforce	Webattack XSS	Webattack SQL Injection	FTP Patator	SSH Patator	Slowloris	Slowhttptest	DoSHulk	Goldeneye	Heartbleed
DDoS	25588	8	0	0	0	0	0	0	0	0	0	10	0	0
Portscan	6	31757	0	0	1	1	0	0	0	0	0	18	3	0
Bot	0	0	393	0	0	0	0	0	0	0	0	0	0	0
Infiltration	0	1	0	0	0	0	0	0	0	0	0	6	0	0
Webattack Bruteforce	0	2	0	0	296	2	0	0	0	1	0	0	0	0
Webattack XSS	0	1	0	0	126	0	0	0	0	0	0	3	0	0
Webattack SQL Injection	0	0	0	0	4	0	0	0	0	0	0	0	0	0
FTP Patator	0	0	0	0	0	0	0	1586	0	0	0	1	0	0
SSH Patator	0	0	0	0	0	0	0	0	1179	0	0	1	0	0
Slowloris	0	1	0	0	3	0	0	0	0	1149	5	0	1	0
Slowhttptest	0	0	0	0	0	1	0	0	0	8	1087	1	3	0
DoSHulk	4	4	0	0	0	0	0	0	0	0	0	46201	6	0
Goldeneye	0	0	0	0	0	0	0	0	0	0	3	13	2041	0
Heartbleed	0	0	0	0	0	0	0	0	0	0	0	2	0	0

Tabla. 4.32: Matriz de confusión del modelo *transformer* en el conjunto de datos CIC-IDS2017 en el modo de ejecución *MaliciousOnly*.

	Porcentaje de acierto	**Aciertos / total**
Resultados generales	0.9480	187105/197359
FTP Bruteforce	0.9988	38626/38672
SSH Bruteforce	0.9997	37506/37518
Goldeneye	0.9976	8282/8302
Slowloris	0.9878	2171/2198
SlowHttp	0.4450	8138/18287
DoSHulk	1	92382/92382

Tabla. 4.33: Resultados del modelo *transformer* en el conjunto de datos CIC-CSE-IDS2018 en el modo de ejecución *MaliciousOnly*.

	FTP-Bruteforce	SSH-Bruteforce	GoldenEye	Slowloris	SlowHttp	DoS Hulk
FTP-Bruteforce	38626	0	0	0	6	0
SSH-Bruteforce	7	37506	0	0	5	0
GoldenEye	0	0	8282	0	2	18
Slowloris	0	0	0	2171	6	21
SlowHttp	10149	0	0	0	8138	0
DosHulk	0	0	0	0	0	92382

Tabla. 4.34: Matriz de confusión del modelo *transformer* en el conjunto de datos CIC-CSE-IDS2018 en el modo de ejecución *MaliciousOnly*.

	Porcentaje de acierto	**Aciertos / total**
Resultados generales	0.9166	61143/66709
UDP	0.9794	3543/3618
MS SQL	0.9673	1649/1705
Portmap	0	0/137
Syn	0.9825	9702/9875
Netbios	0.9593	124/129
UDPLag	0.7389	1321/1785
LDAP	0.4371	167/381
DrDoS DNS	0.4350	319/734
DrDoS WebDDoS	0	0/10
DrDoS TFTP	0.9963	19710/19783
DrDoS UDP	0	0/2084
DrDoS SNMP	0.7584	412/543
DrDoS NetBIOS	0	0/120
DrDoS LDAP	0	0/288
DrDoS MSSQL	0	0/1242
DrDoS NTP	0.9968	24196/24274

Tabla. 4.35: Resultados del modelo *transformer* en el conjunto de datos CIC-DDoS2019 en el modo de ejecución *MaliciousOnly*.

	UDP	MS SQL	Portmap	Syn	Netbios	UDPlag	LDAP	DrDoS DNS	DrDoS WebD-DoS	DrDoS TFTP	DrDoS UDP	DrDoS SNMP	DrDoS Net-BIOS	DrDoS LDAP	DrDoS MSSQL	DrDoS NTP
UDP	3543	3	2	3	0	10	0	3	0	1	49	0	2	0	2	0
MS SQL	1	1649	1	1	1	2	4	4	0	1	2	4	2	1	31	1
Portmap	0	0	0	16	94	1	0	1	0	0	0	0	20	0	4	1
Syn	11	13	27	9702	1	98	0	4	0	9	2	0	1	0	0	7
Netbios	0	0	4	0	124	0	0	0	0	0	0	0	1	0	0	0
UDPlag	276	4	1	45	1	1321	1	3	0	20	104	4	0	2	0	2
LDAP	0	5	1	0	0	2	167	66	0	0	1	46	0	87	6	0
DrDoS DNS	6	67	4	0	5	1	96	319	0	2	7	57	2	78	75	15
DrDoS WebD-DoS	0	0	0	0	0	0	0	0	0	2	0	0	0	0	0	8
DrDoS TFTP	0	0	1	2	0	49	0	0	0	19710	2	0	0	0	3	16
DrDoS UDP	1890	35	2	4	0	120	2	10	0	2	0	0	2	0	14	2
DrDoS SNMP	0	14	0	0	14	0	40	34	0	0	0	412	2	26	11	0
DrDoS Netbios	1 14	17	0	68	0	0	1	0	0	1	1	0	0	10	0	
DrDoS LDAP	0	2	0	0	0	0	148	82	0	0	0	49	0	0	4	3
DrDoS MSSQL	13	798	3	2	0	0	10	28	0	1	10	17	6	0	350	4
DrDoS NTP	0	9	0	15	0	4	1	2	11	28	2	2	0	0	4	24196

Tabla. 4.36: Matriz de confusión del modelo *transformer* en el conjunto de datos CIC-DDoS2019 en el modo de ejecución *MaliciousOnly*.

Pese a los malos resultados que ofrecía en el modo de ejecución *BinaryProblem*, el modelo *transformer* es el que mejores resultados globales ofrece a la hora de diferenciar entre los tipos de ataque. Sus resultados mejoran los del modelo *RandomForest*, aunque sea por apenas un uno o dos por ciento. Sin embargo, es ligeramente peor que este a la hora de distinguir entre algunas de las clases minoritarias, si bien es capaz de asignar clases a las mismas correctamente, en función de la clase.

4.3. Análisis de resultados

A continuación, se realizará un análisis de los resultados mostrados en apartados previos, comentando los susodichos. Como se puede apreciar, estamos ante un problema de dificultad considerable, en el que la mayoría de modelos ofrecen resultados mayormente mediocres, siendo la única excepción el modelo *RandomForest*, que alcanza resultados iguales o superiores al 90 % en todos los conjuntos de datos y en ambos modos de ejecución para cada uno de ellos.

Caben destacar los resultados particularmente nefastos del **modelo CNN**. Esto se puede deber principalmente al hecho de que es un modelo más diseñado para la extracción de conocimiento de imágenes o elementos de ámbito bidimensional o tridimensional, mientras que en nuestro caso cada elemento presenta una sola dimensión, por lo que simplemente estamos usando el método incorrecto para el problema a tratar. Considerando también su extenso tiempo de entrenamiento, de algunas horas, queda descartado como el modelo a emplear para resolver el problema de la detección de tráfico malicioso.

El perceptrón multicapa, si bien alcanza unos resultados aparentemente buenos en el problema binario, es únicamente porque ajusta de tal manera que aplica una clasificación análoga al ZeroR, escogiendo siempre la clase mayoritaria. De esta manera, los resultados que ofrece son buenos, pero dado que nunca detecta la clase minoritaria en un problema sensible a distinguir apropiadamente entre clases, es un comportamiento inaceptable. Cabe destacar, sin embargo, que es uno de los modelos de entrenamiento más rápido, siendo este de apenas unos minutos.

Las RNN son el primer modelo que ofrece resultados aceptables, con porcentajes de acierto que rondan el 80 %. Sin embargo, de manera similar a lo previamente explicado con el perceptrón multicapa, esto se debe mayoritariamente a que acierta con enorme precisión en la clase mayoritaria y con una precisión considerablemente menor en la minoritaria. Sin embargo, los resultados son algo más favorables que

en el caso previo puesto que, si bien con poca precisión, sí es capaz de distinguir la clase minoritaria ocasionalmente. Sin embargo, la existencia de modelos que ofrecen mejores resultados combinado con su muy extenso tiempo de entrenamiento (de la orden de entre uno y tres días en función del dataset) hace que tampoco sea el modelo óptimo para el problema tratado.

Por último, mencionaremos en conjunto a los modelos ***RandomForest*** y ***transformer***. El modelo *RandomForest* es el que mejores resultados ha proporcionado en general, superando el 90 % de acierto tanto en el problema binario como en el de solo tráfico malicioso. Por otro lado, el modelo *transformer* no obtiene buenos resultados en el problema de tráfico binario, pero obtiene resultados que son mejores que los del *RandomForest* en el caso del problema de tráfico malicioso. Comparando más a fondo ambos modelos, el modelo *RandomForest* requiere un tiempo de entrenamiento considerablemente menor, de apenas un par de minutos, mientras que el *transformer* requiere entre dos y cuatro horas en función del conjunto de datos utilizado para el entrenamiento.

Esto da a pensar la posibilidad de crear un modelo de tipo *ensemble*, en el que el proceso de clasificación se diera en dos pasos. En el primer paso, se utilizaría el modelo *RandomForest* para determinar si un nuevo ejemplo detectado se corresponde con una conexión de tráfico benigno o, por contrario, con un ataque. Una vez realizada esta primera comprobación, de detectarse el ejemplo como maligno, se aplicaría el modelo *transformer* para determinar concretamente con qué tipo de ataque se corresponde, aprovechando así las mejores capacidades de ambos modelos.

Capítulo 5

CONCLUSIONES

En este TFG se ha llevado a cabo la investigación de los resultados de diversos algoritmos, incluyendo *RandomForest* y los modelos principalmente usados de la metodología DL, con el objetivo de determinar su efectividad a la hora de tratar el problema de la detección de tráfico maligno en conexiones de red.

Con objetivo de que la investigación fuera lo más exhaustiva posible, primero se realizó una revisión del estado del arte, analizando tanto los modelos más empleados en Deep Learning como una serie de proyectos y artículos que trataban generar un modelo de predicción para un sistema de detección de intrusiones empleando esta tecnología. A su vez, también se analizaron los principales conjuntos de datos empleados con este propósito.

A partir de este análisis del estado del arte, se implementaron un total de cinco modelos: *RandomForest*, perceptrón multicapa, RNN, CNN y *transformer*. Adicionalmente, también se implementaron diversas funcionalidades de utilidad que son empleadas en todos los modelos especificados con el objetivo de facilitar los procesos de entrenamiento y evaluación de los mismos, así como la obtención de medidas para determinar cuál de los modelos empleados era más adecuado para el problema planteado.

Posteriormente, se procedió al análisis experimental de los resultados de los modelos, entrenando cada uno de ellos con cada uno de los tres conjuntos de datos elegidos (CIC-IDS2017, CIC-CSE-IDS2018 y CIC-DDoS2019), distinguiendo adicionalmente dos tipos de ejecución. En la primera de ellas, únicamente se trataba de distinguir si una conexión determinada era tráfico benigno o maligno, mientras que, en la segunda, se ignoraba todo el tráfico benigno y se usaba únicamente el tráfico maligno con objetivo de poder distinguir entre distintos tipos de ataque.

A continuación, se estudiaron los resultados de cada uno de los modelos para cada uno de los modos de ejecución para determinar cuál de todos los modelos era el más adecuado para resolver el problema. Los resultados fueron que el algoritmo *RandomForest* ofrece mejores resultados en lo que precisión y *recall* respecta que el resto de modelos a la hora de tratar el problema de distinguir entre tráfico maligno y tráfico benigno y, si bien también ofrece resultados muy buenos a la hora de distinguir entre tipos de ataques, los resultados del modelo *transformer* son ligeramente mejores que los del *RandomForest* en este tipo de problema.

El resto de modelos obtenían resultados considerablemente peores, siendo el RNN el tercer mejor modelo, pero con una diferencia bastante notable con respecto a *RandomForest* y *transformer*, incluso en el modo de ejecución binario, en el que los resultados del *transformer* no eran particularmente buenos. Los modelos perceptrón multicapa y CNN no son apropiados para el problema, y tienden al sobreajuste muy rápidamente, a menudo reduciéndose a aplicar una estrategia *ZeroR* y predecir siempre la clase mayoritaria, algo inaceptable dado el problema con el que se está tratando.

Estos hechos, sumados al muy poco tiempo de entrenamiento y evaluación que requiere el *RandomForest* en comparación con otros modelos, lo convierten en la opción predilecta a la hora de predecir si una conexión es benigna o maliciosa y, en caso de que lo fuera, el tipo de ataque con el que se corresponde.

5.1. Valoración personal

Esta memoria actúa como un resumen de todo el trabajo realizado con la experimentación de los modelos, pero también como una representación del aprendizaje que he ido recibiendo a lo largo de la carrera. Desde cursar la asignatura de Seguridad en Tecnologías de la Información en segundo de carrera, la idea de los ataques, las vulnerabilidades y las formas de explotarlas me fascinó, y quise continuar por ese camino. Cursar Inteligencia Artificial en ese mismo año y Metaheurísticas el año siguiente provocó un interés por estos modelos más complejos, capaces de aprender y obtener mejores resultados mediante una aproximación al comportamiento humano. Todo ello llevó a la solicitud y obtención de una Beca de Colaboración del Ministerio de Educación y Formación Profesional, que me permitió llevar a cabo este trabajo y continuar desarrollándolo incluso más allá del ámbito de este TFG, como se verá en el apartado posterior.

Si hay algo a mencionar en este apartado es el proceso de aprendizaje total lle-

vado a cabo para la ejecución de este trabajo. En las primeras etapas del mismo, a duras penas era capaz de implementar un modelo de aprendizaje profundo como los descritos, por no mencionar el comprender su funcionamiento desde una perspectiva teórica. Sin embargo, en el proceso de redactar esta memoria aprendí no solo el cómo funcionan, sino también detalles importantes en su uso, como qué tipo de problemas son resueltos más adecuadamente con un modelo u otro, modelos que no funcionan bien para determinados tipos de problemas y, por supuesto, el cómo realizar la implementación de código necesaria para llevar todo esto a cabo, en gran medida gracias al *framework* de *PyTorch*, que simplifica considerablemente todo el proceso de desarrollo permitiendo construir modelos a partir de bloques lógicos, resultando sencillo e intuitivo de emplear.

Las tareas asociadas a la seguridad informática y la minería de datos me resultan fascinantes, especialmente después de haber podido ahondar en ellas como he tenido oportunidad de hacer en este trabajo. Las posibilidades que la minería de datos y la extracción de conocimiento ofrecen parecen ser aparentemente infinitas, con la posibilidad de hallar patrones desconocidos en datos aparentemente inconexos para resolver problemas. A su vez, vuelvo a destacar mi interés por los asuntos de seguridad, tema que, a menudo, queda relegado a un segundo lado pese a ser también de vital importancia para instituciones, empresas y, por supuesto, software, y poder ver más de cerca cómo una conexión puede ser tratada como benigna o maligna, así como la cantidad de datos que realmente se manejan para poder calificarlas como tales, ha resultado fascinante.

Por último, cabe destacar la redacción de esta memoria, proceso también novedoso para mí. Las memorias más largas escritas a lo largo del grado rara vez superaban las veinte páginas, por lo que redactar una memoria que supera las cien ha sido también un desafío gratificante de superar. Ha sido, a su vez, la primera vez que trabajo usando el entorno LaTeX , que ha demostrado ser sorprendentemente intuitivo de emplear una vez se superan las dificultades iniciales que puede conllevar el adaptarse a un nuevo programa para redactar documentos. Las utilidades que ofrece son, sin embargo, inconmensurablemente útiles a la hora de redactar memorias como la del presente TFG.

Para finalizar esta sección, basta decir que todo el proceso de investigación y experimentación, mayormente novedoso para mí, ha resultado gratificante, pese a lo cansado que pueda llegar a ser en determinadas ocasiones, y que el aprendizaje y uso de *Python* y *PyTorch*, así como LaTeX , han demostrado formar, en conjunto, una experiencia muy fructífera para mí.

5.2. Trabajo futuro

Si bien los resultados obtenidos denotan la superioridad de *RandomForest* ante el resto de algoritmos, el hecho de que el *transformer* pueda llegar a tener mejores resultados que este a la hora de diferenciar entre diversos tipos de ataque permite considerar algunas vías por las que continuar la investigación.

La primera de ellas consistiría en idear un modelo en *ensemble*. Dado que *RandomForest* es indiscutiblemente mejor que el resto de modelos a la hora de diferenciar si una conexión dada es maliciosa o benigna, la primera parte del modelo utilizaría *RandomForest* y se encargaría de determinar el tipo de la conexión. En caso de que fuera benigna, el análisis habría terminado y no sería necesario tomar ninguna otra acción. Sin embargo, en caso de que se clasificara como tráfico malicioso, la información sería trasladada a la segunda parte del modelo, que podría emplear un *transformer* en su lugar, dado que ofrece mejores resultados a la hora de diferenciar entre tipos de ataque. De esta manera, se pueden aprovechar tanto los resultados de *RandomForest* como la ligera mejoría en los resultados que causa el *transformer* a la hora de distinguir entre ataques.

Otro camino a tomar, posiblemente de manera simultánea al ya descrito, consistiría en limpiar los conjuntos de datos y modificarlos, de manera que los datos de todos ellos fueran compatibles entre sí. Esto permitiría entrenar al modelo con una mayor cantidad de datos, tanto de tráfico benigno como maligno, de manera que algunas de las clases especialmente minoritarias podrían ser detectadas más fácilmente. Sin embargo, debido a la gran cantidad total de datos, esto probablemente también provocaría un mayor desbalanceo de los mismos y la necesidad de aplicar algunas técnicas para resolverlo en mayor o menor medida, como un sobremuestreo de las clases minoritarias y una reducción de las mayoritarias.

Tras la finalización de este trabajo, se pretende idear un modelo como el propuesto, además de crear una aplicación de línea de comandos que lo utilice con el objetivo de predecir el tipo de tráfico, de manera que un usuario determinado pueda emplearla para analizar una conexión cualquiera, siempre que el formato de la misma se corresponda con el del conjunto de datos conocido. Adicionalmente, se pretende que el programa incluya la funcionalidad para poder volver a entrenar los modelos con los datos recibidos, mejorando las capacidades del mismo en el proceso.

Bibliografía

[1] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer, 1 edition, 2007.

[2] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[3] Ronald Brachman and Tej Anand. The process of knowledge discovery in databases: A first sketch. pages 1–12, 01 1994.

[4] Usama Fayyad, Gregory Piatetsky-Shapiro, and Padhraic Smyth. From data mining to knowledge discovery in databases. *AI Magazine*, 17(3):37, Mar. 1996.

[5] Barry A. Devlin and Paul T. Murphy. An architecture for a business and information system. *IBM Syst. J.*, 27:60–80, 1988.

[6] Dorian Pyle. *Data Preparation for Data Mining*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1999.

[7] Tsau Lin. Attribute transformations for data mining i: Theoretical explorations. *International Journal of Intelligent Systems*, 17:213 – 222, 02 2002.

[8] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[9] David E. Rumelhart and James L. McClelland. *Learning Internal Representations by Error Propagation*, pages 318–362. 1987.

[10] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The elements of statistical learning: data mining, inference and prediction*. Springer, 2 edition, 2009.

[11] Tariq Mahmood and Uzma Afzal. Security analytics: Big data analytics for cybersecurity: A review of trends, techniques and tools. In *2013 2nd National Conference on Information Assurance (NCIA)*, pages 129–134, 2013.

[12] Mohsen Naderpour and Jie Lu. A situation analysis decision support system based on dynamic object oriented bayesian networks. *Journal of Software*, 9, 08 2014.

[13] Zhimin Zhang, Huansheng Ning, Feifei Shi, Fadi Farha, Yang Xu, Jiabo Xu, Fan Zhang, and Kim-Kwang Raymond Choo. Artifcial intelligence in cyber security: research advances, challenges, and opportunities. *Artificial Intelligence Review*, 55:1029–1053, 2022.

[14] Xingjiao Wu, Luwei Xiao, Sun Yixuan, Junhang Zhang, Tianlong Ma, and Liang He. A survey of human-in-the-loop for machine learning. *Future Generation Computer Systems*, 135, 05 2022.

[15] Rachida Ihya, Abdelwahed Namir, Sanaa Filali, Mohammed Aitdaoud, and Fatima zahra Guerss. J48 algorithms of machine learning for predicting user's the acceptance of an e-orientation systems. pages 1–8, 10 2019.

[16] Yi Sun, Xiaogang Wang, and Xiaoou Tang. Deep learning face representation by joint identification-verification. *CoRR*, abs/1406.4773, 2014.

[17] Danilo Jimenez Rezende, Shakir Mohamed, and Daan Wierstra. Stochastic backpropagation and approximate inference in deep generative models, 2014.

[18] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, nov 1997.

[19] Yusuf Korkmaz. Developing password security system by using artificial neural networks in user log in systems. In *2016 Electric Electronics, Computer Science, Biomedical Engineerings' Meeting (EBBT)*, pages 1–4, 2016.

[20] Abdulhadi Shoufan. Continuous authentication of uav flight command data using behaviometrics. In *2017 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, pages 1–6, 2017.

[21] Fahman Saeed, Muhammad Hussain, and Hatim A Aboalsamh. Classification of live scanned fingerprints using histogram of gradient descriptor. In *2018 21st Saudi Computer Society National Computer Conference (NCC)*, pages 1–5, 2018.

[22] Aditya Amberkar, Parikshit Awasarmol, Gaurav Deshmukh, and Piyush Dave. Speech recognition using recurrent neural networks. In *2018 International Conference on Current Trends towards Converging Technologies (ICCTCT)*, pages 1–4, 2018.

[23] Cheol Young Park, Kathryn Blackmond Laskey, Paulo C. G. Costa, and Shou Matsumoto. A process for human-aided multi-entity bayesian networks learning in predictive situation awareness. In *2016 19th International Conference on Information Fusion (FUSION)*, pages 2116–2124, 2016.

[24] Yunhu Jin, Yongjun Shen, Guidong Zhang, and Hua Zhi. The model of network security situation assessment based on random forest. In *2016 7th IEEE International Conference on Software Engineering and Service Science (ICSESS)*, pages 977–980, 2016.

[25] Zhao Dongmei and Liu Jinxing. Study on network security situation awareness based on particle swarm optimization algorithm. *Computers & Industrial Engineering*, 125:764–775, 2018.

[26] Hongrui Bao, Haiguang He, Zhe Liu, and Zhongwei Liu. Research on information security situation awareness system based on big data and artificial intelligence technology. In *2019 International Conference on Robots & Intelligent System (ICRIS)*, pages 318–322, 2019.

[27] Mohiuddin Ahmed, Abdun Mahmood, and Jiankun Hu. A survey of network anomaly detection techniques. *Journal of Network and Computer Applications*, 60:19–31, 11 2015.

[28] Nouf Saleh Aljurayban and Ahmed Emam. Framework for cloud intrusion detection system service. In *2015 2nd World Symposium on Web Applications and Networking (WSWAN)*, pages 1–5, 2015.

[29] Lingjing Kong, Guowei Huang, Ying Zhou, and Jianfeng Ye. Fast abnormal identification for large scale internet traffic. In *Proceedings of the 8th International Conference on Communication and Network Security*, ICCNS '18, page 117–120, New York, NY, USA, 2018. Association for Computing Machinery.

[30] Naila Marir, Huiqiang Wang, Guangsheng Feng, Bingyang Li, and Meijuan Jia. Distributed abnormal behavior detection approach based on deep belief network and ensemble svm using spark. *IEEE Access*, 6:59657–59671, 2018.

[31] V. Kanimozhi and Prem Jacob. Artificial intelligence based network intrusion detection with hyper-parameter optimization tuning on the realistic cyber dataset csecic-ids2018 using cloud computing. *ICT Express*, 5, 04 2019.

[32] Ibraheem Aljamal, Ali Tekeoğlu, Korkut Bekiroglu, and Saumendra Sengupta. Hybrid intrusion detection system using machine learning techniques in cloud computing environments. In *2019 IEEE 17th International Conference on Software Engineering Research, Management and Applications (SERA)*, pages 84–89, 2019.

[33] Pandeeswari Nagarajan and Ganesh Kumar. Anomaly detection system in cloud environment using fuzzy clustering based ann. *Mobile Networks and Applications*, 21, 08 2015.

[34] Vinayaka Jyothi, Xueyang Wang, Sateesh K. Addepalli, and Ramesh Karri. Brain: Behavior based adaptive intrusion detection in networks: Using hardware performance counters to detect ddos attacks. In *2016 29th International Conference on VLSI Design and 2016 15th International Conference on Embedded Systems (VLSID)*, pages 587–588, 2016.

[35] Xiaoyong Yuan, Chuanhuang Li, and Xiaolin Li. Deepdefense: Identifying ddos attack via deep learning. In *2017 IEEE International Conference on Smart Computing (SMARTCOMP)*, pages 1–8, 2017.

[36] Chang-Jung Hsieh and Ting-Yuan Chan. Detection ddos attacks based on neural-network using apache spark. pages 1–4, 05 2016.

[37] Lorenzo Fernández Maimó, Félix J. García Clemente, Manuel Gil Pérez, and Gregorio Martínez Pérez. On the performance of a deep learning-based anomaly detection system for 5g networks. In *2017 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computed, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*, pages 1–8, 2017.

[38] Pengchuan Wang, Qianmu Li, Deqiang Li, Shunmei Meng, Muhammad Bilal, and Amrit Mukherjee. Security in defect detection: A new one-pixel attack for fooling dnns. *Journal of King Saud University - Computer and Information Sciences*, 35(8):101689, 2023.

[39] Weiwei Hu and Ying Tan. Generating adversarial malware examples for black-box attacks based on gan. In *International Conference on Data Mining and Big Data*, pages 409–423. Springer, 2022.

[40] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[41] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckes-

ser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[42] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. Cuda, release: 10.2.89, 2020.

[43] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[44] CIC. Intrusion detection evaluation dataset (cic-ids2017), 2017.

[45] Arash Habibi Lashkari. Cicflowmeter-v4.0 (formerly known as iscxflowmeter) is a network traffic bi-flow generator and analyser for anomaly detection. https://github.com/iscx/cicflowmeter, 08 2018.

[46] CIC and CSE. Cic-cse-ids2018, a collaborative project between the communications security establishment (cse) and the canadian institute for cybersecurity (cic), 2018.

[47] CIC. Ddos evaluation dataset (cic-ddos2019), 2019.

[48] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Huggingface's transformers: State-of-the-art natural language processing, 2020.