UNIVERSIDAD LA SALLE

CARRERA DE INGENIERÍA DE SISTEMAS

TESIS DE GRADO

COMPARACIÓN DE REDES NEURONALES PARA LA DETECCIÓN DE INTRUSOS EN REDES

Por: Mijaelha De los Rios Aliaga

Tutor: Ing. Kevin Marlon Soza Mamani

Tesis de Grado presentada para la obtención
del Título de Licenciatura en Ingeniería de Sistemas

La Paz - Bolivia

2024

DEDICATORIA

Este trabajo está dedicado a:

A mi mamá por todo su apoyo.

AGRADECIMIENTOS

Agradezco primero a Dios por haberme ayudado a terminar la carrera, a mi mamá Olga Aliaga, a mi papá Rigoberto De los Rios y a mis tíos Javier Aliaga, Maria Eugenia Aliaga y Rossy Chibana.

Agradezco a mi tutor, el Ing. Marlon Soza y a mis revisores, el Lic. Javier Heredia y el Ing. Osamu Yokosaki, por su contribución en el desarrollo de la investigación. Además, quiero agradecer a mis primas Etsuko y Rossy y a mi amigo Christopher.

Finalmente, quiero expresar mi gratitud a los docentes de las carreras de Ingeniería de Sistemas y Psicología de la Universidad La Salle Bolivia.

ABSTRACT

ABSTRACT

Anomaly-based network intrusion detection has a high false positive rate compared to signature-based detection, causing unnecessary alerts. This research proposes the modeling and comparison of convolutional, recurrent, and hybrid CNN-RNN neural networks based on the detection precision generated by trained models using a dataset of normal network traffic captures and the following threats: DoS, port scanning, and ARP spoofing.

The network traffic captures, obtained using Wireshark, consisting of 13 columns to represent the characteristics of replicated threats on Kali Linux and Ubuntu virtual machines were used to train the models for multiple classification, following the procedures of the methodology of the life cycle of a machine learning project and evaluating the models using confusion matrices and the parameters of precision, accuracy, and recall.

For the comparison of the neural networks, one-way factorial variance analysis was used with 30 samples of the precision of each model to determine that there is a significant difference between the results obtained by the three models and *t-tests* were conducted to validate that the trained neural networks present a high precision, greater than 95%.

The conclusions drawn from this research indicate that the CNN, RNN, and CNN-RNN neural networks exhibit different threat detection precision values in networks, greater than 95% based on the tests of the models.

RESUMEN

La detección de intrusos en redes basada en anomalías presenta un alto índice de falsos positivos frente a la detección basada en firmas, provocando alertas innecesarias. En esta investigación, se plantea el modelado y comparación de redes neuronales convolucionales, recurrentes e híbridas CNN-RNN basada en la precisión de detección que generan los modelos entrenados utilizando un *dataset* de capturas de tráfico de red normal y de las siguientes amenazas: DoS, escaneo de puertos y suplantación de ARP.

Las capturas de tráfico de red, obtenidas utilizando *Wireshark*, formadas por 13 columnas para representar las características de las amenazas replicadas en máquinas virtuales de *Kali Linux* y *Ubuntu* se utilizaron para el entrenamiento de los modelos para la clasificación múltiple, siguiendo los procedimientos de la metodología del ciclo de vida de un proyecto de aprendizaje automático y realizando la evaluación de los modelos mediante matrices de confusión y los parámetros de precisión, exactitud y exhaustividad.

Para la comparación de las redes neuronales, se utilizó el análisis de varianza factorial de una vía con 30 muestras de la precisión de cada modelo para determinar que existe una diferencia significativa entre los resultados de los tres modelos y se realizaron pruebas de *t* de *Student* para validar que las redes neuronales entrenadas presentan una alta precisión, mayor al 95%.

Las conclusiones obtenidas expresan que las redes neuronales CNN, RNN y CNN-RNN presentan diferentes valores de precisión de detección de amenazas en redes, mayores al 95% con base en las pruebas de los modelos.

ÍNDICE

ÍNDICE DE CONTENIDO

CAPÍTULO 1: GENERALIDADES

CAPÍTULO 2: MARCO TEÓRICO

CAPÍTULO 3: MARCO APLICATIVO

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

ÍNDICE DE ECUACIONES

ÍNDICE DE ANEXOS

CAPÍTULO 1

GENERALIDADES

CAPÍTULO 1
GENERALIDADES

1.1. INTRODUCCIÓN

Existe una creciente preocupación por la seguridad informática dentro de instituciones y organizaciones debido al incremento de amenazas en redes, como por ejemplo, los ataques de denegación de servicio distribuido (DDoS), capaces de colapsar con información la actividad de redes, servidores y diferentes servicios en línea (Erickson, 2008).

El uso de sistemas de detección de intrusos (IDS) y de sistemas de prevención de intrusos (IPS) se ha vuelto cada vez más frecuente para la identificación y reconocimiento del acceso no autorizado que una persona o un grupo externo puede obtener a una red o sistema, así como a información confidencial (Shimeall y Spring, 2014).

Los sistemas de detección de intrusos se pueden diferenciar, según su estrategia de análisis, en dos modelos: detección basada en firmas y detección basada en anomalías. El modelo de detección basado en firmas requiere que se conozca previamente los patrones de amenazas y actividad maliciosa para ser detectados (Shimeall y Spring, 2014). En contraste, la detección basada en anomalías compara la actividad en tiempo real con una línea base de lo que se considera como comportamiento normal para detectar el tráfico fuera de ese rango, logrando identificar ataques que no se conocían anteriormente (Shimeall y Spring, 2014).

La detección de intrusos basada en anomalías se beneficia de los modelos de redes neuronales para analizar el comportamiento de diferentes ataques e identificar patrones, lo que permite descubrir nuevas amenazas, a pesar de que puede producir un alto índice de falsos negativos y falsos positivos, al igual que otras técnicas de aprendizaje automático (Dua y Du, 2016).

Los modelos de redes neuronales convolucionales (CNN), reconocidos por sus diversas aplicaciones en tareas relacionadas con el procesamiento de imágenes y videos, presentan una capacidad de extracción de características que puede aplicarse en la clasificación de datos de tráfico de red (Meliboev, Alikhanov y Kim, 2020). Las redes neuronales recurrentes (RNN), mejor conocidas por su uso en el procesamiento y generación de textos, muestran un alto

rendimiento en la clasificación múltiple utilizada para la detección de anomalías, gracias a su capacidad de manejar información secuencial (Sun et al., 2020).

Con el propósito de combinar las fortalezas de los modelos de redes neuronales convolucionales y recurrentes, es posible entrenar un modelo híbrido CNN-RNN para la clasificación de tráfico de red utilizada para la detección de intrusos en redes, en donde se utilizan capas convolucionales de extracción de características y capas recurrentes con una arquitectura de memoria de largo corto plazo (LSTM) para el procesamiento de la información a lo largo del tiempo (Yao, Wang, Liu, Chen y Sheng, 2021).

1.2. ESTADO DEL ARTE

1.2.1. Modelo para la detección de escaneo de puertos

La tesis titulada “Modelo para la Detección de Escaneo de Puertos de la Computadora en una Red WLAN”, presentada en el año 2021 por Leonardo Julio Rios Aliaga, licenciado en informática de la Universidad Mayor de San Andrés, propone el uso del método de clasificación *Bagging,* con 50 árboles de decisión para la detección de escaneo de puertos en redes inalámbricas. El escaneo de puertos es un conjunto de técnicas que se realizan antes de ejecutar un ataque, enviando paquetes para identificar su estado y determinar si existen servicios que pueden ser explotados (Rios Aliaga, 2021).

El modelo elaborado obtuvo una precisión de detección del 99.96%, mostrando la factibilidad y las ventajas de la aplicación de algoritmos de *machine learning* para detectar el escaneo de puertos en redes inalámbricas (Rios Aliaga, 2021).

La presente investigación considerará al escaneo de puertos como una amenaza dentro de las diferentes clases de ataques que los modelos de redes neuronales podrán identificar, ya que es una técnica ampliamente utilizada que puede replicarse en un ambiente controlado.

1.2.2. Detección de malware en una red con machine learning

En la tesis presentada en el año 2020, por Fernando Roger Cornejo Tarqui, licenciado en informática de la Universidad Mayor de San Andrés, se describe el proceso de elaboración de

diferentes modelos para su aplicación en la detección de *malware* en redes, utilizando los siguientes algoritmos de *machine learning*: *Robust Covariance*, *Isolation Forest* y *One-Class SVM*. Para la evaluación de los modelos se utilizaron matrices de confusión, curvas de ROC y áreas bajo la curva (AUC), además de los siguientes parámetros: exactitud, precisión, sensibilidad y especificidad (Cornejo Tarqui, 2020).

Los resultados obtenidos mostraron que el modelo que tuvo el mejor desempeño fue *Isolation Forest*, con una probabilidad del 92% de clasificar correctamente el tráfico de red normal y el tráfico producido por *malware* en redes (Cornejo Tarqui, 2020). En contraste, la presente investigación no considerará la detección de *malware* en redes para el entrenamiento de los modelos, pero realizará una comparación de la precisión de diferentes redes neuronales para la detección de intrusos en redes.

1.2.3. Revisión sistemática de literatura sobre sistemas de detección de intrusos

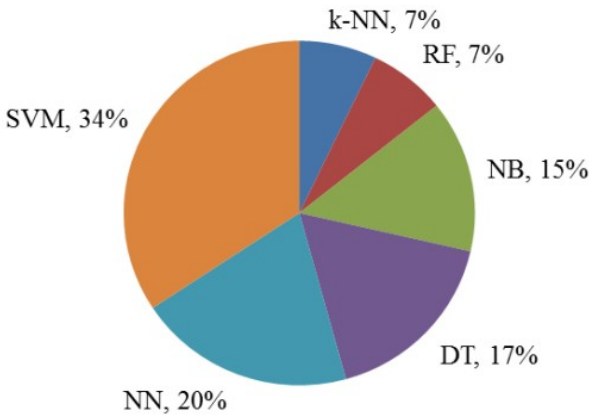
En el artículo de conferencia titulado *"A Systematic Literature Review of Intrusion Detection System for Network Security: Research Trends, Datasets and Methods"*, publicado en el año 2020, por Amarudin, Ferdiana y Widyawan, se lleva a cabo una revisión sistemática de la literatura sobre la detección de intrusos en redes, con base en el análisis de 62 estudios primarios publicados entre enero de 2016 y mayo de 2020. La revisión aborda los diferentes métodos, técnicas y las tendencias emergentes para la detección de intrusos en el ámbito de la seguridad en redes, al igual que el uso de *datasets* públicos y privados (Amarudin, Ferdiana y Widyawan, 2020).

Las técnicas de aprendizaje automático más utilizadas para la detección de intrusos en redes identificadas en el estudio son: clasificación, agrupación, análisis de *datasets*, estimación, predicción, asociación y técnicas estadísticas (Amarudin et al., 2020). La técnica más frecuente fue la clasificación, utilizada en el 81% de los artículos analizados y consiste en la asignación de dos etiquetas en la clasificación binaria o más de dos etiquetas para la clasificación múltiple, según el tipo de tráfico de red asociado, permitiendo diferenciar el tráfico normal de las anomalías en una red (Amarudin et al., 2020).

En el siguiente gráfico, se muestran los métodos y algoritmos de aprendizaje automático más utilizados para la detección de intrusos en redes.

Figura 1

Métodos más utilizados para la detección de intrusos en redes



Nota. El gráfico muestra los métodos más utilizados para la detección de intrusos en redes. Obtenido en *A Systematic Literature Review of Intrusion Detection System for Network Security: Research Trends, Datasets and Methods*, por Amarudin, Ferdiana y Widyawan, 2020.

Como se muestra en el gráfico de la Figura 1, el método más utilizado para la detección de intrusos en redes, según el análisis realizado en el artículo de conferencia, es el uso del algoritmo de máquinas de vectores de soporte (SVM) con el 34%, seguido por los modelos de redes neuronales (NN) con el 20%, árboles de decisión (DT) con el 17%, el algoritmo de *Naive Bayes* (NB) con un 15%, bosque aleatorio (RF) con 7% y el algoritmo *K-Nearest Neighbors* (k-NN) con el 7% (Amarudin et al., 2020).

El método de detección de amenazas seleccionado para la presente investigación es el entrenamiento de modelos de redes neuronales, utilizando la técnica de clasificación múltiple para el tráfico de red.

1.3. PLANTEAMIENTO DEL PROBLEMA

1.3.1. Identificación del problema

A pesar de la existencia de herramientas de seguridad comúnmente encontradas en sistemas informáticos como *firewalls*, también conocidos como cortafuegos, que proveen una barrera de

seguridad que filtra el tráfico proveniente de fuentes inseguras, o programas de antivirus que se encargan de los archivos que pueden contener programas maliciosos, todavía existe la necesidad de que las empresas y organizaciones utilicen herramientas capaces de analizar el comportamiento del tráfico de red para identificar amenazas en tiempo real, entre ellas se encuentran los sistemas de detección de intrusos, para alertar la existencia de anomalías y los sistemas de prevención de intrusos, capaces de tomar medidas para bloquear ataques (Cisco, 2022).

La detección de intrusos basada en anomalías continúa siendo considerada un tema relevante de investigación dentro de las áreas de seguridad informática y aprendizaje automático, debido a que a pesar de que este método ha sido ampliamente estudiado, todavía presenta cuestiones fundamentales sin resolver, entre las cuales se destaca el alto índice de falsos positivos y falsos negativos, que pueden generar el envío de alertas innecesarias, lo que resulta en el desperdicio de recursos y de tiempo, o que existan amenazas legítimas que no sean detectadas (Dua y Du, 2016).

La implementación de modelos de redes neuronales artificiales (ANN) para la detección de intrusos en redes facilita la identificación de patrones complejos de ataques desconocidos y permite inferir y clasificar en tiempo real el tráfico de red a partir de datos incompletos, ya que las redes neuronales pueden realizar el procesamiento de información no lineal (Dua y Du, 2016), es decir, que cambia constantemente y no sigue un proceso lógico de desarrollo (Collins English Dictionary, s.f.).

1.4. PREGUNTA DE INVESTIGACIÓN

¿Cómo difiere la precisión de detección de intrusos entre las redes neuronales convolucionales, redes neuronales recurrentes y redes neuronales híbridas convolucionales recurrentes?

1.5. OBJETIVOS

1.5.1. Objetivo general

Elaborar los modelos de redes neuronales convolucionales, recurrentes e híbridas convolucionales recurrentes y comparar su precisión para la detección de intrusos en redes.

1.5.2. Objetivos específicos

- Recolectar información sobre los tres modelos de redes neuronales para su aplicación en la detección de intrusos.
- Elaborar los tres modelos de redes neuronales y entrenarlos utilizando la base de datos K*DD Cup 1999* para obtener una evaluación preliminar.
- Establecer una línea base de tráfico de red normal y replicar los ataques que los modelos podrán clasificar para obtener los datos de tráfico de red normal y de las amenazas para el entrenamiento de las redes neuronales.
- Entrenar los tres modelos de redes neuronales con los datos recolectados para su evaluación en la detección de intrusos en redes.
- Determinar si existe una diferencia significativa de precisión de detección de intrusos en redes entre los tres modelos de redes neuronales para realizar la comparación.

1.6. JUSTIFICACIÓN

1.6.1. Justificación técnica

La presente investigación se justifica técnicamente por el modelado de redes neuronales convolucionales, recurrentes e híbridas convolucionales recurrentes, en el lenguaje de programación *Python,* utilizando la plataforma *TensorFlow* y la API para aprendizaje profundo *Keras*, inicialmente utilizando el *dataset KDD Cup 1999* para su entrenamiento y posteriormente empleando capturas de tráfico de red normal y de diferentes amenazas en redes, obtenidas utilizando la herramienta *Wireshark.*

Los ataques en redes se replicarán en un ambiente controlado utilizando máquinas virtuales de *Kali Linux* y *Ubuntu* en *Oracle VM VirtualBox* en un equipo anfitrión con un sistema operativo *Windows* 10.

1.6.2. Justificación social

La presente investigación se justifica socialmente debido a que el aumento de los ataques en redes dirigidos a empresas, organizaciones e instituciones, públicas y privadas, pueden afectar significativamente su credibilidad, provocando desconfianza por parte de los clientes y del

público en general, por lo cual es importante que los modelos de aprendizaje automático y aprendizaje profundo entrenados para la detección de amenazas en redes tengan una alta precisión de detección.

1.7. ALCANCES

1.7.1. Alcance temático

El alcance temático de la presente investigación comprenderá los siguientes temas.

- Líneas de investigación: ingeniería de sistemas, inteligencia artificial y seguridad en redes.
- Áreas temáticas: redes neuronales, detección de intrusos.

1.7.2. Alcance geográfico

La presente investigación se realizará en un ambiente controlado en la ciudad de La Paz, con *hotspot* móvil como punto de acceso a Internet para una red de área local (LAN) formada por un equipo anfitrión con *Windows 10* y tres máquinas virtuales, en donde la primera utilizará *Kali Linux* para realizar los ataques y las siguientes dos el sistema operativo *Ubuntu*, para simular el tráfico de una red.

1.7.3. Alcance temporal

La presente tesis de grado se realizará durante un período de once meses y su presentación se llevará a cabo según el reglamento establecido por la universidad.

1.8. LÍMITES

- Para la presente investigación se realizará la comparación exclusivamente basada en la precisión de los modelos de redes neuronales convolucionales, redes neuronales recurrentes y redes neuronales híbridas convolucionales recurrentes.
- Los modelos de redes neuronales se entrenarán utilizando capturas de tráfico de red, obtenidas en el ambiente controlado, de una línea establecida de tráfico normal y de ataques de denegación de servicio (DoS), escaneo de puertos y suplantación de ARP, por lo cual

solamente podrán realizar la clasificación múltiple de los ataques en redes mencionados y del tráfico normal.

- La finalidad de la presente investigación no incluye la implementación de los modelos de redes neuronales en un sistema de detección de intrusos en redes.

1.9. FORMULACIÓN DE LA HIPÓTESIS

1.9.1. Hipótesis de investigación

Las redes neuronales convolucionales, recurrentes e híbridas convolucionales recurrentes generan diferentes valores de precisión, mayores al 95%, permitiendo la detección de intrusos en redes.

1.9.2. Hipótesis nula

Las redes neuronales convolucionales, recurrentes e híbridas convolucionales recurrentes no generan diferentes valores de precisión y son menores al 95%, dificultando la detección de intrusos en redes.

1.10. IDENTIFICACIÓN Y ANÁLISIS DE VARIABLES

1.10.1. Variable independiente

Tipo de red neuronal.

1.10.2. Variable dependiente

Precisión de detección de intrusos en redes.

1.11. OPERACIONALIZACIÓN DE LAS VARIABLES

En la tabla que se muestra a continuación, se realiza la operacionalización de las variables identificadas para el desarrollo de la investigación: el tipo de red neuronal como variable independiente y la precisión de detección de intrusos en redes como variable dependiente.

Tabla 1

Operacionalización de las variables

Variable	Definición conceptual	Definición operacional	Indicadores	Escala de medición
Tipo de red neuronal	Modelos capaces de generalizar observaciones establecidas para predecir resultados (Gulli et al., 2019).	La variable se mide según el modelo de red neuronal que fue utilizado.	Red neuronal convolucional	Nominal
			Red neuronal recurrente	Nominal
			Red neuronal híbrida convolucional recurrente	Nominal
Precisión de detección de intrusos en redes	Cantidad de verdaderos positivos dividida entre la suma de verdaderos positivos y falsos positivos (Burkov, 2020).	La variable se mide según los resultados de precisión de las pruebas.	Precisión de la prueba	Porcentaje

Nota. La tabla muestra la operacionalización de las variables. Elaboración propia.

1.12. DISEÑO METODOLÓGICO

1.12.1. Enfoque de la investigación

Para el desarrollo de la presente investigación, se considerará un enfoque cuantitativo, dado que se realizará mediante la recopilación y el análisis de datos numéricos para la aplicación de pruebas de hipótesis, buscando obtener resultados empleando mediciones que permiten estudiar la relación existente entre las variables identificadas, con el propósito de presentar conclusiones objetivas y basadas en su totalidad en evidencia cuantificable (Creswell y Creswell, 2017).

1.12.2. Tipo de investigación

El tipo de investigación establecido es cuantitativo experimental, en donde se busca determinar si el tipo de red neuronal, que es la variable independiente, tiene un efecto en la variable dependiente, que se determinó como la precisión de detección de intrusos en redes, lo cual se realizará dentro de un ambiente controlado, manteniendo un control riguroso sobre los

procedimientos para aumentar la validez de los resultados y su replicabilidad (Creswell y Creswell, 2017).

1.12.3. Método de investigación

Se utilizarán métodos de investigación cuantitativos experimentales para determinar si los modelos de redes neuronales entrenados para la detección de intrusos en redes producen diferentes valores de precisión, mayores al 95%.

Para la primera parte de la prueba de hipótesis se utilizará el método de análisis de varianza factorial de una vía ANOVA entre los tres tipos de redes neuronales, que consiste en probar si existe una diferencia significativa entre las medias de dos o más grupos categorizados por una sola variable independiente (Witte y Witte, 2017). Para la segunda parte se realizarán pruebas de *t* de *Student* a cada uno de los modelos de redes neuronales para determinar si su precisión es superior al 95%.

CAPÍTULO 2

MARCO TEÓRICO

CAPÍTULO 2
MARCO TEÓRICO

2.1. INGENIERÍA DE SISTEMAS

La ingeniería de sistemas se puede definir como el conjunto de principios, prácticas y técnicas que se utilizan para la elaboración de sistemas complejos, con un enfoque multidisciplinario que ayuda a comprender las interrelaciones que conforman los sistemas (Haberfellner et al., 2019). Según Haberfellner et al. (2019), desde una perspectiva clásica, algunos de los procesos más importantes de la ingeniería de sistemas son los siguientes.

- Modelos de procesos: Hacen referencia a los diferentes modelos utilizados en la ingeniería de sistemas para facilitar el proceso de desarrollo.
- Procesos de resolución de problemas: Son los enfoques estructurados formados por la identificación, el análisis y la resolución de problemas.
- Gestión de proyectos: Es el proceso conformado por la planificación, organización, dirección y manejo de los recursos para cumplir con los objetivos planteados para el sistema.
- Arquitectura de sistemas: Consiste en el proceso de diseñar y construir sistemas complejos, considerado como el proceso general de la ingeniería de sistemas.
- Desarrollo de conceptos: Se conoce como la fase inicial para la ingeniería de sistemas, en donde se realiza la evaluación de la situación para identificar las posibles soluciones al problema planteado.
- Diseño de sistemas: Es el proceso de definición de la arquitectura, los componentes y las interfaces que un sistema debe tener para satisfacer los requerimientos establecidos.

2.2. METODOLOGÍA DE INVESTIGACIÓN

Para la aplicación del diseño de investigación cuantitativo experimental planteado anteriormente, se seguirán los procedimientos experimentales propuestos por Creswell y Creswell (2017), que se dividen en tres partes: diseño, procedimiento y medidas.

Dentro de la etapa de diseño se realiza la selección y descripción del ambiente experimental, se definen las variables independientes y dependientes y se determinan las técnicas y

herramientas que se utilizarán para la medición del efecto de las variables independientes sobre las variables dependientes (Creswell y Creswell, 2017).

En la etapa de procedimiento ocurre la manipulación de las variables mediante las técnicas seleccionadas para llegar a la fase de medidas, en la que se realizan las pruebas de hipótesis, análisis de los datos e interpretación de los resultados para poder redactar las conclusiones (Creswell y Creswell, 2017).

2.2.1. Pruebas de hipótesis

El método de análisis de varianza factorial de una vía, conocido como *One-way* ANOVA, se utiliza cuando se busca demostrar que existe una diferencia entre las medias de dos o más grupos y existe una única variable independiente (Witte y Witte, 2017). Para calcular la variabilidad entre las medias de más de tres grupos se utiliza el estadístico *F*, como se muestra en la siguiente ecuación.

Estadístico *F*

$$F = \frac{variabilidad\ entre\ los\ grupos}{variabilidad\ dentro\ de\ los\ grupos} \tag{1}$$

Fuente: Witte y Witte, 2017

La variabilidad entre los grupos se define como la diferencia entre las medias de cada grupo con respecto a la media de todos los grupos, mientras que la variabilidad dentro de los grupos consiste en la diferencia entre las muestras individuales dentro de un grupo con respecto a la media de cada grupo (Witte y Witte, 2017). Para calcular la variabilidad entre los grupos se utiliza la ecuación que se muestra a continuación.

Variabilidad entre los grupos

$$MS_{between} = \frac{SS_{between}}{df_{between}} \tag{2}$$

Fuente: Witte y Witte, 2017

Donde:

$MS_{between}$ = cuadrado medio entre los grupos

$SS_{between}$ = suma de cuadrados entre los grupos

$df_{between}$ = grados de libertad entre los grupos

La variabilidad dentro de los grupos se calcula de manera similar que la variabilidad entre los grupos, como se muestra a continuación.

Variabilidad dentro de los grupos

$$MS_{within} = \frac{SS_{within}}{df_{within}} \quad (3)$$

Fuente: Witte y Witte, 2017

Donde:

MS_{within} = cuadrado medio dentro de los grupos

SS_{within} = suma de cuadrados dentro de los grupos

df_{within} = grados de libertad dentro de los grupos

Para determinar si existe una diferencia significativa entre la media de un solo grupo y la media poblacional cuando la desviación estándar es desconocida y se debe estimar a partir de las muestras, se utiliza la prueba de *t* de *Student* (Witte y Witte, 2017), utilizando la fórmula que se muestra a continuación.

Estadístico *t*

$$t = \frac{\bar{X} - \mu_{hyp}}{S_{\bar{X}}} \quad (4)$$

Fuente: Witte y Witte, 2017

Donde:

$\bar{X}$ = media del grupo

μ_{hyp} = media hipotética de la población

$S_{\bar{X}}$ = error estándar estimado

Para rechazar una hipótesis nula se utilizan los resultados obtenidos de las pruebas, como el estadístico *F* y el estadístico *t*, para calcular el *p-value*, que se define como la probabilidad de alcanzar un valor igual o más extremo que el valor real cuando la hipótesis nula es verdadera, en otras palabras, es el nivel de significancia más bajo que se puede obtener para rechazar la hipótesis nula con base en a los resultados calculados a partir de las muestras (Newbold, Carlson y Thorne, 2022).

2.3. INTELIGENCIA ARTIFICIAL

Russell y Norvig (2021) establecen que la definición de inteligencia artificial puede verse desde diferentes enfoques. El primer enfoque propone el uso del Test de Turing, elaborado por Alan Turing en 1950, con el propósito de determinar si una máquina puede pensar demostrando las siguientes habilidades, como se conocen actualmente: procesamiento de lenguaje natural, representación del conocimiento, razonamiento automatizado y aprendizaje automático (Russell y Norvig, 2021).

El segundo enfoque se conoce como el modelo cognitivo y postula que para poder decir que un sistema piensa como un humano primero es necesario conocer el proceso de dichos pensamientos, utilizando técnicas de neuroimagen, introspección y diferentes experimentos psicológicos, para poder expresar dichas teorías en sistemas computacionales y afirmar que pueden realizar acciones humanas, siempre y cuando su comportamiento tenga cierto parecido con el comportamiento observado en humanos (Russell y Norvig, 2021).

El tercer enfoque, denominado el enfoque de las leyes del pensamiento, consiste en la creación de sistemas inteligentes utilizando la lógica basada en el conocimiento de la realidad, es decir, de lo que es verdadero (Russell y Norvig, 2021).

Este enfoque permite resolver problemas en los que se conocen las reglas, por lo cual queda un vacío que fue llenado posteriormente por las teorías de la probabilidad, permitiendo que se construyan modelos de pensamiento racional para hacer estimaciones o predicciones hacia el futuro (Russell y Norvig, 2021).

El último enfoque se basa en la idea de que el pensamiento racional del modelo anterior no es suficiente para definir la inteligencia artificial, por lo cual surge el enfoque del agente racional, que propone el uso de agentes inteligentes que realizan acciones de manera autónoma y que son capaces de percibir su entorno para adaptarse a cambios y así lograr obtener el mejor resultado incluso cuando haya incertidumbre (Russell y Norvig, 2021).

Desde un enfoque práctico, la inteligencia artificial se conoce como el área que se encarga del desarrollo de sistemas capaces de realizar, de manera automática, actividades de carácter intelectual que se creía que eran propias de los seres humanos (Chollet, 2017).

2.4. APRENDIZAJE AUTOMÁTICO

El aprendizaje automático es una especialidad de la inteligencia artificial, que se puede definir como la resolución de problemas a través de la construcción y entrenamiento de modelos estadísticos basados en datos recolectados anteriormente (Burkov, 2020).

En contraste con la programación tradicional, en la que se ingresan las reglas mediante un programa junto con los datos para obtener una respuesta, en la programación de modelos de aprendizaje automático se debe ingresar la respuesta esperada y los datos, con el objetivo de generar reglas que puedan ser aplicadas a nuevos datos para obtener como resultado la respuesta prevista (Chollet, 2017). Los modelos de aprendizaje automático se pueden dividir en cuatro tipos según su forma de entrenamiento: aprendizaje supervisado, aprendizaje no supervisado, aprendizaje semisupervisado y aprendizaje reforzado (Burkov, 2020). El aprendizaje supervisado utiliza datos etiquetados para la elaboración de modelos que puedan determinar la categoría de nuevos datos de entrada o predecir un valor numérico (Russell y Norvig, 2021).

En el aprendizaje no supervisado se utilizan datos sin etiquetas para el entrenamiento, con el objetivo de generar modelos que puedan identificar similitudes entre los datos de entrada para poder agruparlos en categorías de forma automática (Burkov, 2020). El aprendizaje semisupervisado utiliza una gran cantidad de datos no etiquetados, junto con un conjunto reducido de datos etiquetados para lograr un modelo mejorado en comparación a los modelos generados utilizando solo datos etiquetados, como ocurre con el aprendizaje supervisado (Burkov, 2019).

El aprendizaje reforzado se utiliza para resolver problemas secuenciales, con objetivos a largo plazo, empleando la idea de tomar el estado esperado como entrada y obtener como salida la acción óptima que se debe realizar para llegar a ese estado (Burkov, 2020).

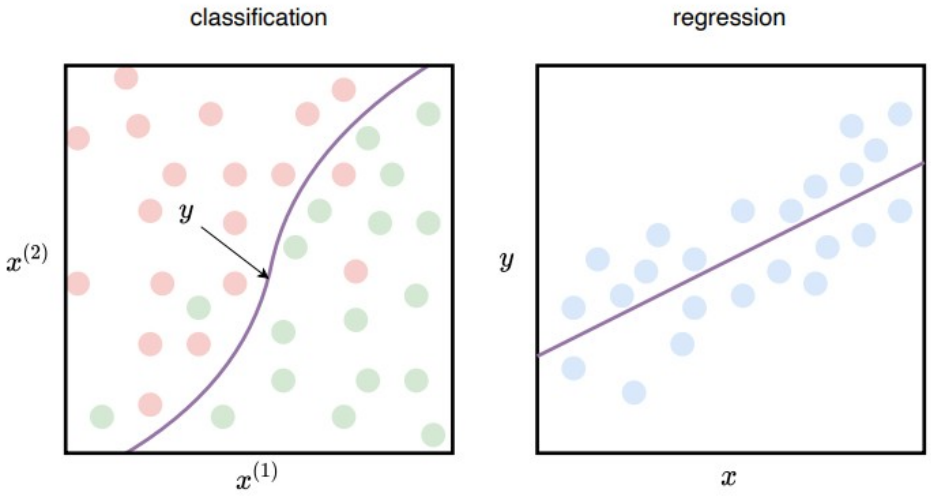
2.4.1. Aprendizaje supervisado

El aprendizaje supervisado consiste en que el modelo sea entrenado utilizando un conjunto de datos etiquetados, es decir, que toma el vector de datos como información de entrada para generar un modelo que pueda determinar, como dato de salida, una clase o un valor para el conjunto de datos nuevos de entrada (Burkov, 2020). Este tipo de aprendizaje puede aplicarse cuando se tienen conjuntos de datos etiquetados finitos, como por ejemplo seguro y no seguro, utilizando el método de clasificación para determinar la clase a la que pertenecen los datos y cuando los datos son valores numéricos y el resultado esperado es la predicción de un número, aplicando el método de regresión (Russell y Norvig, 2021).

En la siguiente imagen, se muestra la diferencia que puede observarse al graficar los resultados de los modelos de clasificación y de regresión.

Figura 2

Diferencia entre los gráficos de los modelos de clasificación y regresión



Nota. En la figura se muestran ejemplos de gráficos de modelos de clasificación y regresión. Obtenido en *Machine Learning Engineering. True Positive Incorporated*, por Andriy Burkov, 2020.

Como se observa en el gráfico, el algoritmo de aprendizaje de clasificación busca una separación entre las diferentes clases, mientras que en la regresión, debe existir una tendencia que se aproxime a los datos del entrenamiento (Burkov, 2020).

2.4.1.1. Regresión

Para resolver problemas de predicción de un valor real con base en un conjunto de datos, se utiliza el algoritmo de regresión lineal, que se utiliza para predecir un valor y desconocido, con un valor x asignado como entrada, calculando los valores óptimos para (w, b), con el propósito de obtener una predicción de alta precisión (Burkov, 2019).

A continuación, se muestra la ecuación utilizada por los algoritmos de regresión lineal para predecir valores.

Regresión lineal

$$f_{w,b} = wx + b \tag{5}$$

Fuente: Burkov, 2019

Donde:

$f_{w,b}$ = es la función de salida, que simboliza que f está parametrizado por w y b.

w = es un vector con una dimensión determinada

b = es un número real

2.4.1.2. Clasificación

Los modelos de clasificación utilizan la técnica de regresión logística, la cual no debe confundirse con un modelo de regresión, que emplea la función logística estándar, conocida como *sigmoid* para resolver problemas de clasificación binaria, cuando solo se tienen dos clases y se les puede asignar los valores de 0 y 1, en donde se utiliza una función continua de codominio (0, 1) para que cuando el resultado del modelo con el valor de entrada x se encuentre más cerca del 0 o del 1, tenga la etiqueta asignada a ese valor, (Burkov, 2019). A continuación, se muestra la fórmula de la función *sigmoid*.

Función *sigmoid*

$$f(x)=\frac{1}{1+\mathrm{e}^{-x}} \tag{6}$$

Fuente: Burkov, 2019

Donde:

$f(x)$ = es el valor de salida

x = es el valor de entrada

e = es la base del logaritmo natural

Para el modelo de regresión logística, se emplea la función *sigmoid*, como se muestra en la Ecuación 6, junto con la fórmula de regresión lineal de la Ecuación 5, como se observa a continuación.

Modelo de regresión logística

$$f_{w,b}(x)=\frac{1}{1+\mathrm{e}^{-(wx+b)}} \tag{7}$$

Fuente: Burkov, 2019

Donde:

$f_{w,b}(x)$ = es el valor de salida, que simboliza que f está parametrizado por w y b.

x = es el valor de entrada

w = es un vector con una dimensión determinada

b = es un número real

Para un modelo de clasificación múltiple se utiliza la función *softmax*, que es una forma generalizada de la función *sigmoid* para obtener varias salidas (Burkov, 2019).

A continuación, se muestra la función *softmax*, utilizada para la clasificación múltiple por su capacidad de calcular la distribución de varias clases (Burkov, 2019).

Función *softmax*

$$\sigma(z)=[\sigma^{(1)},\ldots,\sigma^{(D)}], \text{ donde } \sigma^{j}=\frac{e^{(z^{(j)})}}{\sum_{k=1}^{D} e^{(z^{(k)})}} \quad (8)$$

Fuente: Burkov, 2019

Donde:

$\sigma(z)$ = es la función de salida *softmax*

z = es un vector de números reales que representan el valor asociado con cada clase

D = es el número de clases en el vector de entrada

k = es un índice para referirse a una clase específica

j = es un índice adicional de referencia a una clase específica del vector de entrada

e = es la base del logaritmo natural

El uso de la función *softmax* genera como resultado las distribuciones de probabilidad de las clases, que son mutuamente excluyentes (Patterson y Gibson, 2017).

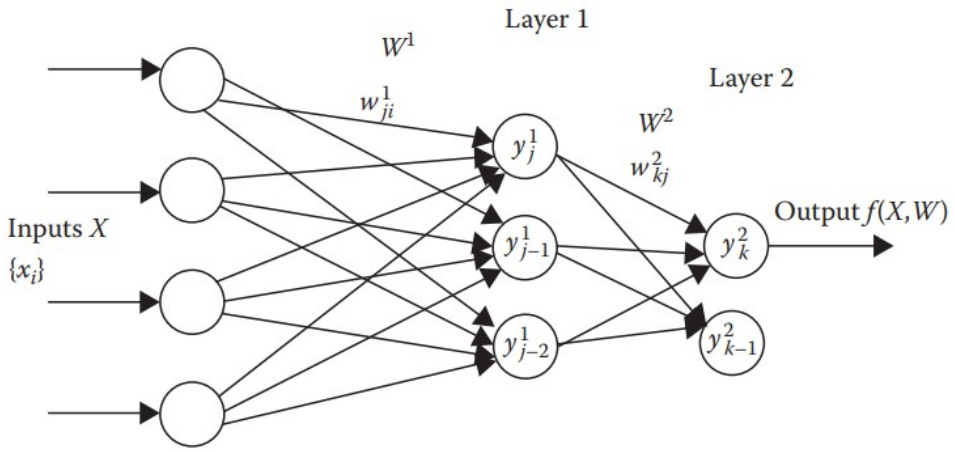
2.5. REDES NEURONALES

Las redes neuronales artificiales (ANN) son modelos de aprendizaje automático basados inicialmente en estudios de la estructura y el funcionamiento del sistema nervioso central, ya que cada red puede formarse por diferentes neuronas interconectadas, organizadas por capas en donde las neuronas de una capa pueden enviar información a las neuronas de la siguiente capa (Gulli et al., 2019). El comportamiento de una red neuronal se determina a partir de la estructura de la red, que está conformada por el número de neuronas, el número de capas y el tipo de conexión entre las capas (Patterson y Gibson, 2017).

Las redes neuronales realizan el procesamiento de información no lineal en un grupo conectado de neuronas que conforman capas ocultas para transformar los datos de entrada en datos de salida, con el objetivo de minimizar el error entre el resultado real calculado y la salida esperada (Dua y Du, 2016). A continuación, se muestra la estructura de una red neuronal con cuatro entradas, una salida y dos capas ocultas.

Figura 3

Estructura de una red neuronal de dos capas



Nota. En la figura se muestra la estructura de una red neuronal. Obtenido en *Data Mining and Machine Learning in Cybersecurity*, por Sumeet Dua y Xian Du, 2016.

En la Figura 3, la primera capa tiene tres neuronas y un vector de pesos W^1 y la segunda capa está formada por dos neuronas y el vector de pesos W^2. El funcionamiento óptimo de una red neuronal depende de la selección de la función de activación y de los pesos, los cuales deben especificarse para la conexión entre neuronas (Dua y Du, 2016).

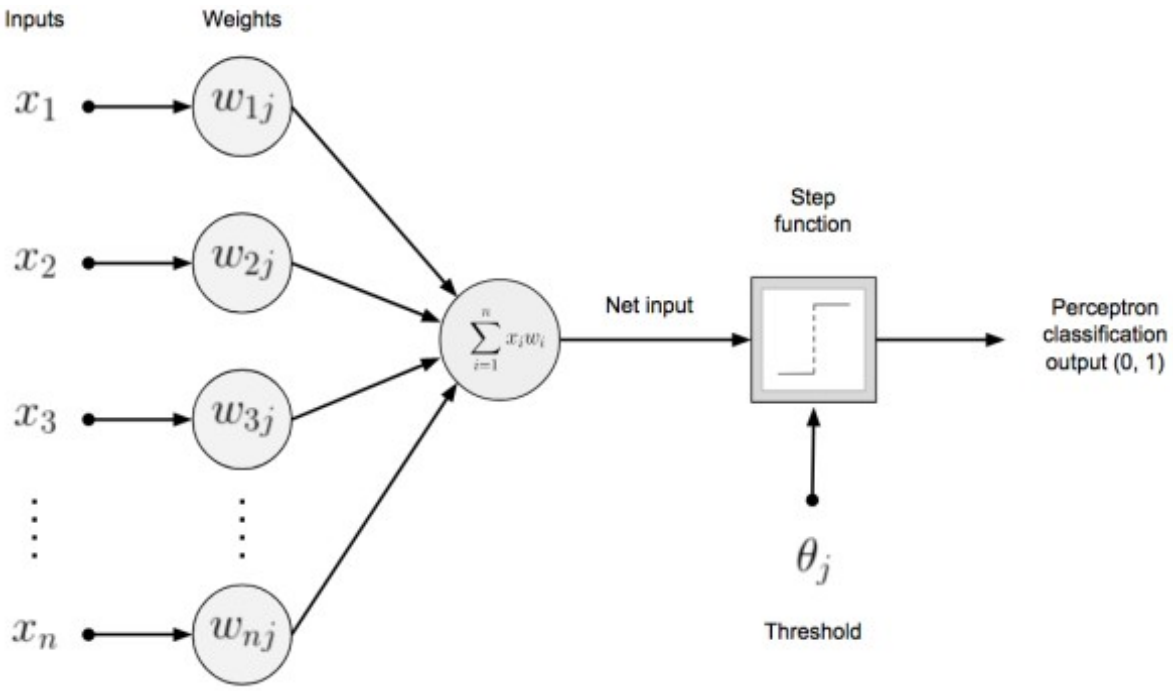
Las funciones de activación, introducen transformaciones no lineales a las redes neuronales para que sean capaces de aprender de los patrones complejos que existen en los datos, ya que pueden cambiar constantemente (Ravichandiran, 2019).

La estructura más simple que una red neuronal puede tener se denomina perceptrón de una capa y es un modelo lineal que se utiliza como clasificador binario, que funciona al sumar los datos de entrada por sus respectivos pesos, para enviarlos a la función de activación llamada función de paso que tiene un límite definido anteriormente, con el objetivo de que el modelo genere como salida un valor único binario (Patterson y Gibson, 2017).

En la figura que se muestra a continuación, se observa la estructura básica de un perceptrón de una capa.

Figura 4

Perceptrón de una capa



Nota. En la figura se muestra un perceptrón de una capa. Obtenido en *Deep Learning*. *A Practitioner's Approach*, por Josh Patterson y Adam Gibson, 2017.

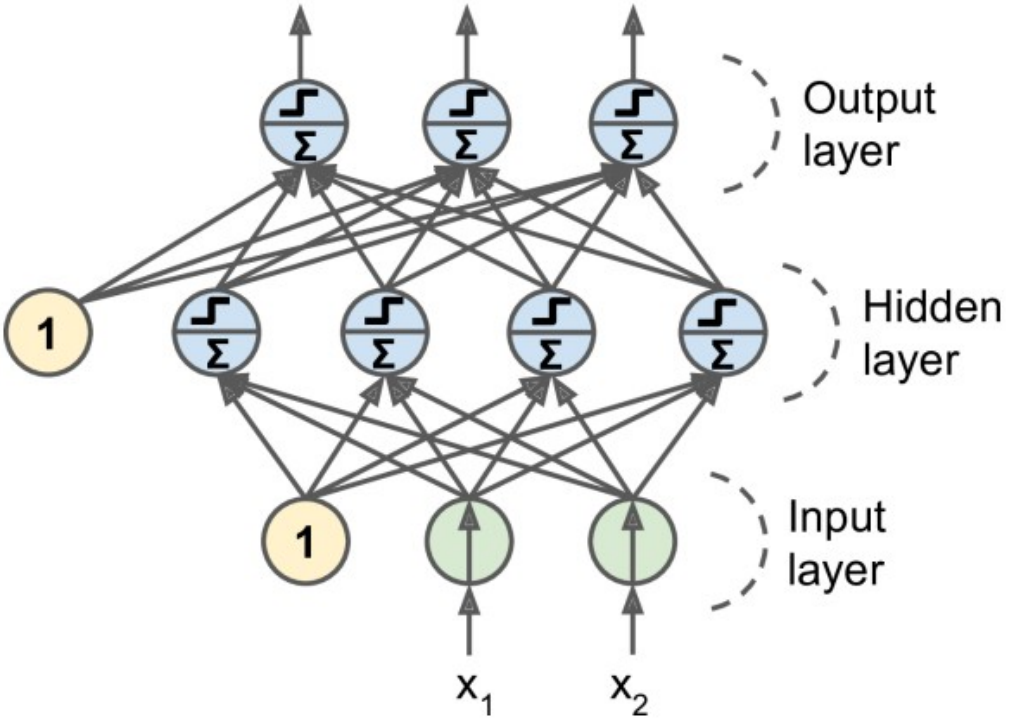
El término perceptrón se utiliza comúnmente para los modelos de redes neuronales que tienen una sola capa lineal, por lo cual, para los modelos que emplean más capas individuales, una después de la otra, se aplica el término de perceptrón multicapa (Gulli et al., 2019).

La ventaja de utilizar perceptrones multicapa es la flexibilidad en el tipo de función de activación que se puede aplicar, ya que dichas funciones se encargan de transformar las entradas y los pesos para que su resultado pasa a la siguiente capa (Patterson y Gibson, 2017).

En el algoritmo del perceptrón multicapa las neuronas de la primera capa oculta reciben una entrada para enviar los resultados de la función de activación a la siguiente capa oculta, donde se aplica la función respectiva hasta llegar a la capa de salida y obtener el resultado del entrenamiento de la red neuronal (Gulli et al., 2019). En la figura que se muestra a continuación, se observa la estructura un perceptrón multicapa.

Figura 5

Perceptrón multicapa



Nota. En la figura se muestra un perceptrón multicapa. Obtenido en *Hands-On Machine Learning with Scikit-Learn and TensorFlow*, por Aurélien Géron, 2017.

Como se muestra en la Figura 5, el perceptrón multicapa está formado por una capa de entrada, una o más capas ocultas y una capa final de salida.

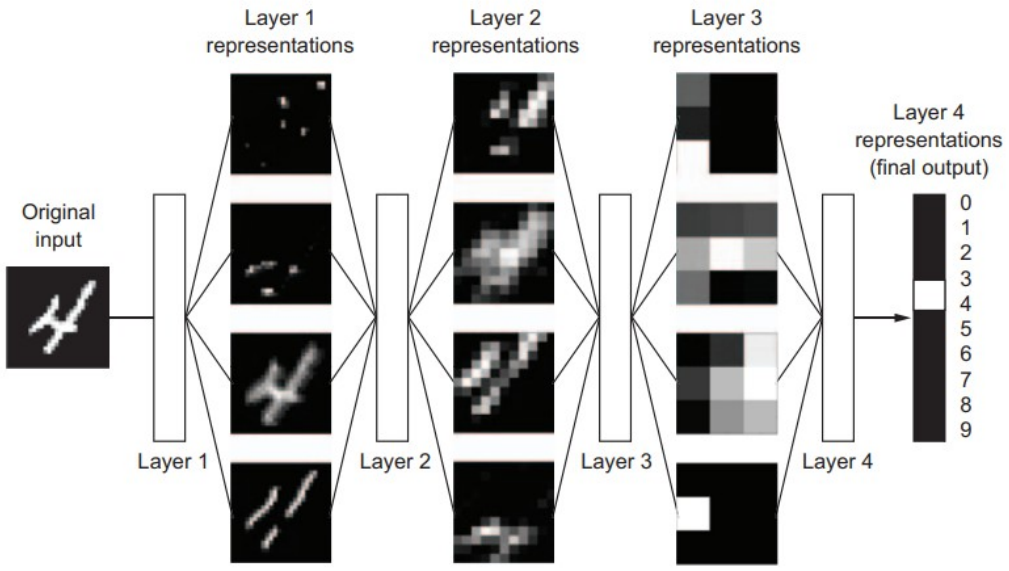
2.5.1. Aprendizaje profundo

El aprendizaje profundo o *Deep Learning* se considera el entrenamiento de redes neuronales con más de dos capas ocultas y puede definirse como la aplicación del conjunto de algoritmos y funciones para el entrenamiento de redes neuronales, independientemente de su profundidad y de la cantidad de parámetros que utiliza (Burkov, 2019).

En la siguiente figura, se muestra la estructura y la representación de un modelo de clasificación, utilizando aprendizaje profundo.

Figura 6

Estructura y representación de un modelo de clasificación utilizando aprendizaje profundo



Nota. En la figura se muestra la estructura y representación de un modelo de clasificación utilizando aprendizaje profundo. Obtenido en *Deep Learning with Python*, por François Chollet, 2017.

En la Figura 6, la red neuronal transforma la imagen original de entrada, que representa un dígito, dentro de las capas ocultas sucesivas para que mediante el entrenamiento pueda identificar el dígito en la capa de salida.

El entrenamiento de redes neuronales utilizando aprendizaje profundo logró generar modelos de clasificación de imágenes, reconocimiento de escritura manual y reconocimiento de voz casi a nivel humano (Chollet, 2017).

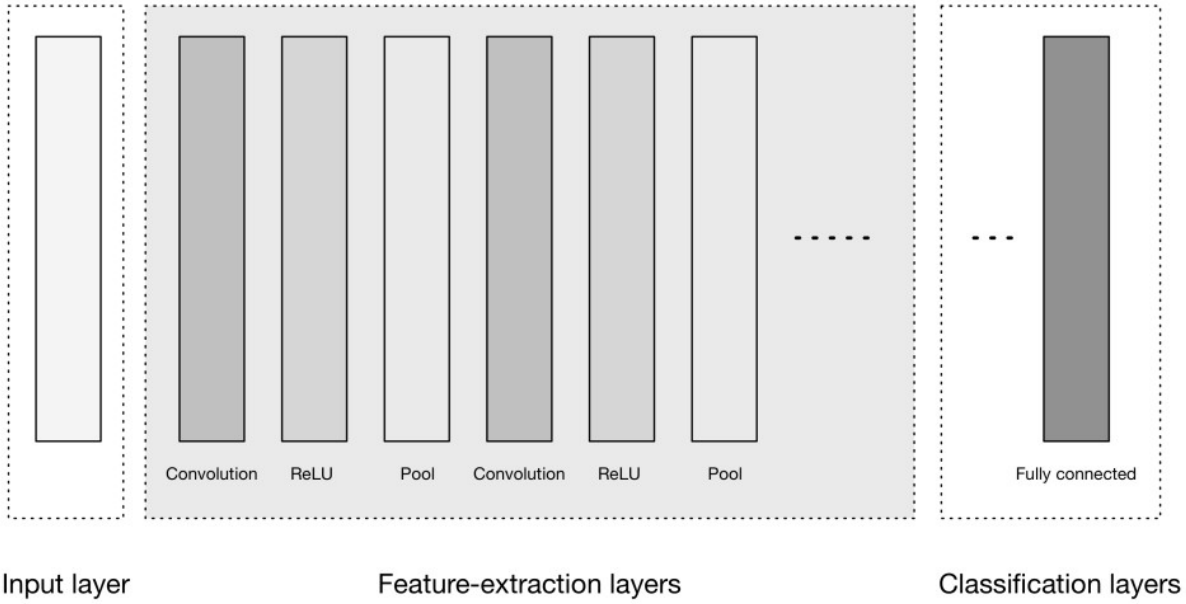
2.5.2. Redes neuronales convolucionales

Las redes neuronales convolucionales se utilizan ampliamente para la clasificación y reconocimiento de imágenes, pero también son efectivas para el análisis de texto y sonido (Patterson y Gibson, 2017). Las CNNs generalmente funcionan transformando los datos de entrada dentro de cada una de las capas ocultas conectadas en un conjunto de clases,

asignadas por la capa de salida (Patterson y Gibson, 2017). A continuación, se muestra la arquitectura más utilizada en tareas de clasificación utilizando redes neuronales convolucionales.

Figura 7

Arquitectura de una red neuronal convolucional



Nota. En la figura se muestra la arquitectura de una red neuronal convolucional. Obtenido en *Deep Learning. A Practitioner's Approach*, por Josh Patterson y Adam Gibson, 2017.

Como se muestra en la Figura 7, la arquitectura de una red neuronal convolucional presenta tres grupos principales: la capa de entrada, la capa de extracción de características y la capa de clasificación.

La capa de entrada es en donde se introducen los datos que serán procesados, en la capa de extracción de características existe un patrón que se repite de una capa de convolución, en donde se aplica la función de activación que en el caso del gráfico es la unidad lineal rectificada (ReLU) y una capa de agrupación, mejor conocida como *pooling*, en donde se construyen progresivamente las características de orden superior de los datos de entrada (Patterson y Gibson, 2017).

En la siguiente ecuación, se muestra la función de activación de unidad lineal rectificada o ReLU.

Función ReLU

$$relu(z) = max(0, z) \tag{9}$$

Fuente: Patterson y Gibson, 2017

Donde:

$relu(z)$ = es la función de salida de unidad lineal rectificada

z = es el valor de entrada

$max(0, z)$ = es la operación que devuelve 0 si z es negativo y z si el valor de entrada es mayor o igual a 0

En la capa de clasificación, que se encuentra conectada o una o más capas, se toman las características de orden superior para generar probabilidades o puntajes de clase de las salidas (Patterson y Gibson, 2017), utilizando la fórmula que se muestra a continuación.

Producto de las capas de clasificación

$$b \times N \tag{10}$$

Fuente: Patterson y Gibson, 2017

Donde:

b = es el número de ejemplos obtenidos

N = es la cantidad de clases que existen

Las CNNs aplican operaciones denominadas convoluciones, que se utilizan para extraer características y formar una nueva matriz, conocida como mapa de características (Ravichandiran, 2019). Las convoluciones consisten en reglas para unir dos conjuntos de datos, utilizando los datos de entrada que pueden ser los datos originales o el resultado de una convolución anterior dentro de una matriz multidimensional y un *kernel* que también es una

matriz multidimensional de parámetros adaptados para la red neuronal (Goodfellow et al., 2016). A continuación, se muestra la fórmula para realizar convoluciones.

Convolución

$$s(t)=(x*w)(t) \tag{11}$$

Fuente: Goodfellow, Bengio y Courville, 2016

Donde:

$s(t)$ = es la función de salida

t = es un valor real de las funciones

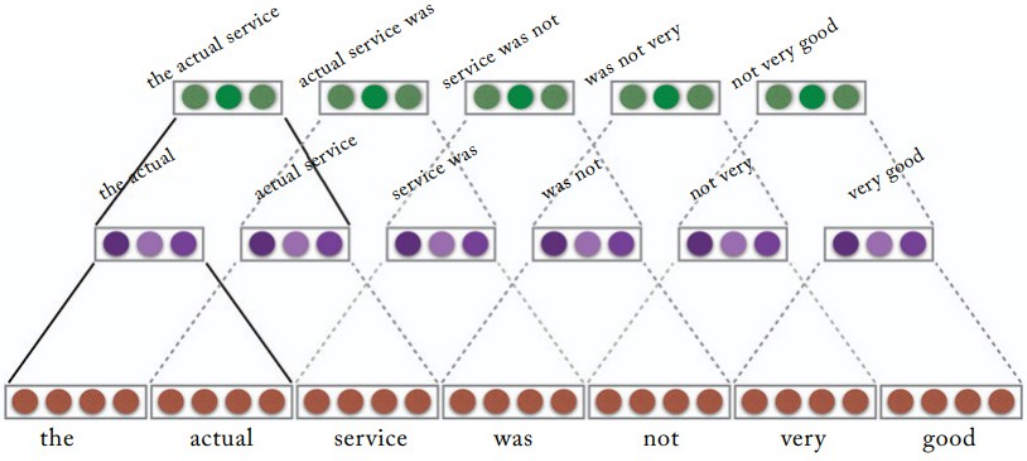
x = es la función de entradas

w = es la función de kernel

A continuación, se muestra un ejemplo de una convolución aplicada para el análisis de una secuencia de texto.

Figura 8

Convolución para una secuencia de texto



Nota. En la figura se muestra una convolución para una secuencia de texto. Obtenido en *Neural Network Methods in Natural Language Processing*, por Yoav Goldberg, 2017.

Al aplicar convoluciones para una secuencia de texto, como se muestra en la Figura 8, se emplean funciones no lineales, consideradas como filtros, para transformar las palabras en valores escalares, obteniendo como resultado un vector donde cada dimensión corresponde respectivamente a un filtro (Goldberg, 2017). Cuando se aplican redes convoluciones para el análisis de textos, su arquitectura se describe como una jerarquía de capas de convolución para la extracción de información significativa y la identificación de patrones (Goldberg, 2017).

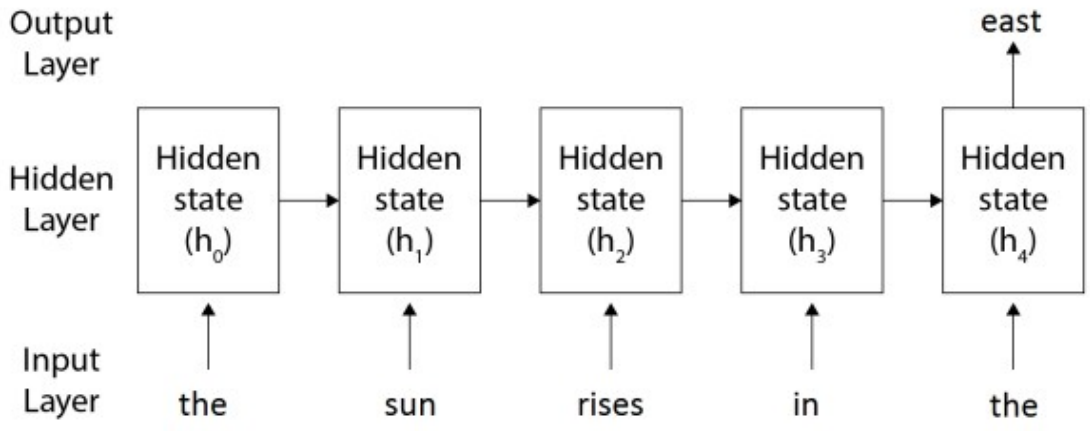
2.5.3. Redes neuronales recurrentes

Las redes neuronales recurrentes se diferencian principalmente por su capacidad de predecir salidas basadas en un estado previo oculto, además de los datos de entrada y tienen varias aplicaciones dentro del campo del procesamiento de lenguaje natural y en el procesamiento de datos secuenciales, como texto, audio, video y series de tiempo entre otros (Ravichandiran, 2019).

En la siguiente figura, se muestra la estructura de una red neuronal recurrente para el procesamiento de texto.

Figura 9

Estructura de una red neuronal recurrente



Nota. En la figura se muestra la estructura de una red neuronal recurrente. Obtenido en *Hands-On Deep Learning Algorithms with Python*, por Sudharsan Ravichandiran, 2019.

Como se observa en la Figura 9, las redes neuronales recurrentes utilizan los estados ocultos h_i como memoria para poder almacenar la información contextual que ha sido analizada, con el objetivo de predecir la siguiente palabra (Ravichandiran, 2019).

El valor de un estado oculto h_t en un tiempo específico, es el resultado de la función del valor del estado oculto en un tiempo anterior h_{t-1} y el valor de entrada en el tiempo actual (Gulli et al., 2019), como se muestra en la siguiente fórmula.

Función recurrente

$$h_t = \phi(h_{t-1},\ X_t) \tag{12}$$

Fuente: Gulli, Kapoor y Pal, 2019

Donde:

h = es el estado recurrente

t = es la representación del paso del tiempo

ϕ = es la función para actualizar el estado recurrente

X = es el valor de entrada

Es importante tomar en cuenta que la Ecuación 8 es recursiva y el estado oculto h_{t-1} puede reemplazarse por h_{t-2} y la función X_t por X_{t-1} cuando corresponda (Gulli et al., 2019).

La función de activación comúnmente utilizada en las capas ocultas de una red neuronal recurrente se denomina tangente hiperbólica (TanH), como se muestra en la siguiente ecuación.

Función TanH

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{13}$$

Fuente: Antonio Gulli, Kapoor y Pal, 2019

Donde:

$\tanh(z)$ = es la función de activación de tangente hiperbólica

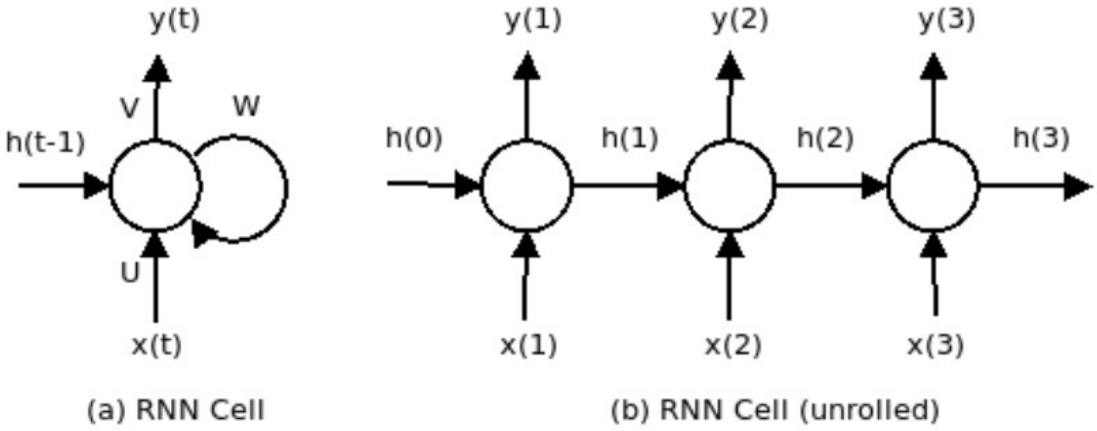
z = es el valor de entrada

e = es la base del logaritmo natural

A continuación, se muestra la representación gráfica reducida de una red neuronal recurrente y la representación de la secuencia completa.

Figura 10

Representación de una RNN



Nota. En la figura se muestran las representaciones de una red neuronal recurrente. Obtenido en *Deep Learning with TensorFlow 2 and Keras. Second Edition*, por Antonio Gulli, Amita Kapoor y Sujit Pal, 2019.

En la primera representación de la Figura 10, $x(t)$ simboliza la entrada y $y(t)$ la salida en un tiempo t determinado, en donde los parámetros del entrenamiento se almacenan en las matrices de pesos U para la entrada, V para la salida y W para las capas ocultas (Gulli et al., 2019).

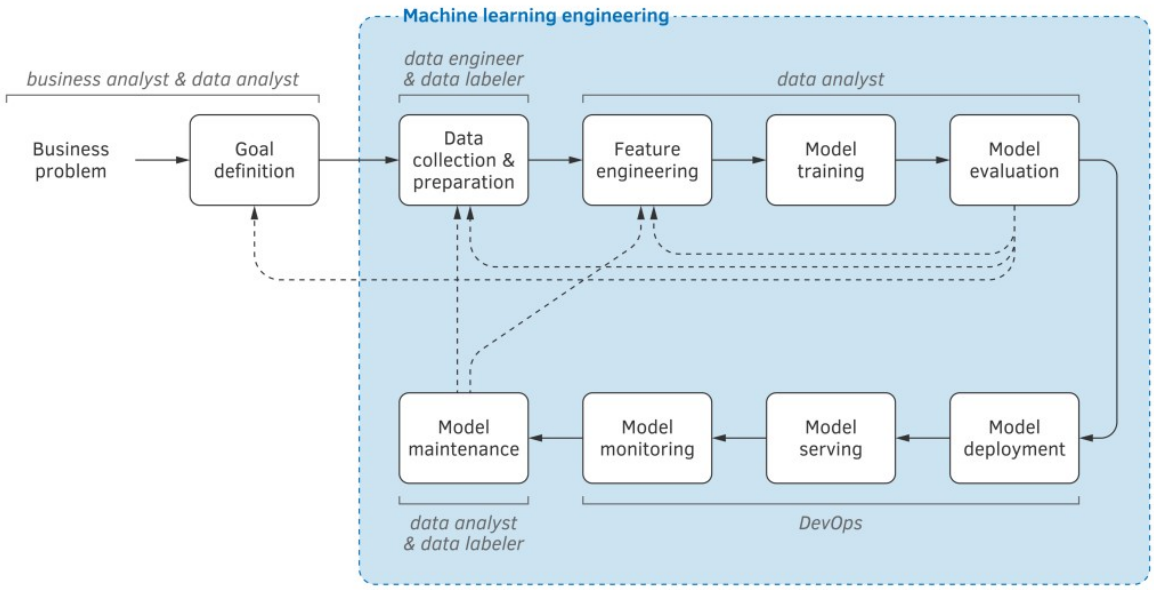
En la representación de la secuencia completa de la red neuronal recurrente, las matrices de pesos U, V y W se mantienen durante cada uno de los tres pasos de tiempo, ya que se aplica la misma operación, pero con diferentes valores de entrada, permitiendo que el número de parámetros necesarios para el entrenamiento se reduzca (Gulli et al., 2019).

2.6. CICLO DE VIDA DE UNA RED NEURONAL

Burkov (2020) propone un ciclo de vida de nueve etapas para la elaboración de proyectos de aprendizaje automático, el cual se aplicará para realizar los modelos de redes neuronales, como se puede observar en la siguiente figura.

Figura 11

Ciclo de vida de un proyecto de aprendizaje automático



Nota. En la figura se muestran las etapas del ciclo de vida de un proyecto de aprendizaje automático. Obtenido en *Machine Learning Engineering. True Positive Incorporated*, por Andriy Burkov, 2020.

La primera etapa consiste en la definición de los objetivos del proyecto de aprendizaje automático, los cuales deben resolver el problema identificado anteriormente utilizando un modelo para el cual primero se describen las estructuras de las entradas y salidas y el nivel de aceptación de rendimiento, el cual se mide utilizando parámetros de rendimiento, que se definen en la etapa de entrenamiento del modelo (Burkov, 2020).

En la etapa de recolección y preparación se debe considerar la calidad, usabilidad, comprensibilidad y confiabilidad de los datos para el entrenamiento, por lo cual en esta etapa

se deben realizar las acciones necesarias para garantizar que los datos obtenidos son adecuados para el modelo (Burkov, 2020).

Una parte importante de esta etapa es la partición de los datos en tres grupos: datos de entrenamiento que se utilizan para el algoritmo de aprendizaje, datos de validación para obtener los valores ideales de los parámetros y datos de prueba, que se utilizan para comprobar el rendimiento del modelo terminado y obtener los resultados (Burkov, 2020).

En la siguiente etapa, llamada ingeniería de características, se realizan los procedimientos necesarios de transformación de los datos para que puedan ser utilizados en modelos de aprendizaje automático, debido a que usualmente no es posible ingresar datos sin procesar y obtener los resultados deseados (Burkov, 2020).

Uno de los métodos más utilizados en la ingeniería de características se denomina normalización y consiste en convertir los rangos de valores reales en un rango estándar, usualmente en un intervalo de $[-1,1]$ o $[0,1]$ para que todos los datos se encuentren relativamente dentro del mismo rango (Burkov, 2019), como se muestra en la siguiente ecuación.

Normalización

$$\bar{x}^{(j)} = \frac{x^{(j)} - min^{(j)}}{max^{(j)} - min^{(j)}} \tag{14}$$

Fuente: Burkov, 2019

Donde:

$\bar{x}$ = es el valor de salida

x = es el valor de entrada

j = es el índice que hace referencia a una dimensión o a un conjunto de valores

$min^{(j)}$ = es el valor mínimo de j

$max^{(j)}$ = es el valor máximo de j

Otro método utilizado en la ingeniería de características es la estandarización o normalización de puntuación z, que es el proceso de redimensionar los valores para que obtengan las propie-

dades de una distribución estándar, en donde la media $\mu = 0$ y la desviación estándar $\sigma = 1$ (Burkov, 2019), como se muestra en la siguiente ecuación.

Estandarización

$$\hat{x}^{(j)} = \frac{x^{(j)} - \mu^{(j)}}{\sigma^{(j)}} \tag{15}$$

Fuente: Burkov, 2019

Donde:

$\hat{x}$ = es el valor de salida

x = es el conjunto de valores de entrada

j = es el índice que hace referencia al conjunto de valores

μ = es la media del conjunto de valores de entrada

σ = es la desviación estándar del conjunto de valores de entrada

Para entrenar modelos de aprendizaje automático utilizando como datos de entrada valores de texto, es necesario el uso de técnicas de ingeniería de características para convertirlos en números como *One Hot Encoding*, utilizada para problemas de clasificación múltiple, en donde se transforman todas las diferentes clases a vectores binarios únicos (Burkov, 2020). Una variación de esta técnica, conocida como el uso de variables ficticias o *dummy*, consiste en asignarle a cada clase un valor numérico diferente, logrando el mismo objetivo (Raschka, 2015).

En la cuarta etapa se realiza el entrenamiento del modelo, que contempla la selección del algoritmo de aprendizaje, la elaboración de la secuencia de transformaciones por las que pasan los datos de entrenamiento antes de llegar al modelo, la selección de los parámetros de rendimiento y la aplicación de técnicas de ajuste de los parámetros, cuando el modelo lo requiere o no se tiene una gran cantidad de datos para la validación, como *Cross Validation* o validación cruzada, que consiste en la división del conjunto de datos para el entrenamiento en varios subconjuntos del mismo tamaño, denominados pliegues, entrenando el modelo de forma iterativa y calculando el promedio de los resultados de los parámetros seleccionados, logrando obtener valores del rendimiento del modelo (Burkov, 2020).

Para evaluar el rendimiento de un modelo de aprendizaje automático entrenado para la clasificación, ya sea binaria o múltiple, se utilizan matrices de confusión que consisten en tablas que representan el éxito del modelo en la predicción de las clases, en donde el número de filas y columnas de la matriz es igual a la cantidad de clases reales (Burkov, 2020). Dicho de otra manera, si se tienen 4 clases en total para la clasificación, la matriz de confusión tendría 4 filas y 4 columnas.

El primer parámetro analizado será la precisión, que se define como la relación entre los verdaderos positivos y el total de las predicciones positivas (Burkov, 2020), como se muestra en la siguiente ecuación.

Precisión

$$precision \stackrel{\text{def}}{=} \frac{TP}{TP+FP} \tag{16}$$

Fuente: Burkov, 2020

Donde:
TP = verdaderos positivos
TN = verdaderos negativos
FP= falsos positivos
FN = falsos negativos

El siguiente parámetro es la exhaustividad o *recall*, que se puede definir como la relación que existe entre las predicciones correctas de verdaderos positivos sobre la suma de todos los verdaderos positivos y falsos negativos (Burkov, 2020), como se puede observar en la siguiente ecuación.

Exhaustividad

$$recall \stackrel{\text{def}}{=} \frac{TP}{TP+FN} \tag{17}$$

Fuente: Burkov, 2020

El último parámetro es la exactitud, que se define como la suma de todos los ejemplos clasificados correctamente entre en número total de ejemplos clasificados (Burkov, 2020), como se muestra a continuación.

Exactitud

$$accuracy \stackrel{\text{def}}{=} \frac{TP + TN}{TP + TN + FP + FN} \tag{18}$$

Fuente: Burkov, 2020

La siguiente etapa consiste en la evaluación del modelo, en donde se monitorea y determina el rendimiento del modelo antes de pasar a la sexta etapa, denominada implementación del modelo, que puede ser de manera estática como parte de un paquete de software que debe ser instalado o dinámica, dentro del dispositivo de los usuarios o en un servidor o a través de una transmisión de modelos (Burkov, 2020).

Las últimas tres etapas en donde se realizan los servicios, monitoreo y mantenimiento del modelo se realizan una vez que se implementó el modelo en un entorno real (Burkov, 2020), por lo cual no se tomarán en cuenta en el desarrollo de la investigación.

2.7. SEGURIDAD INFORMÁTICA

La seguridad informática se define como el conjunto de actividades que se realizan para proteger todos los elementos que conforman un sistema informático, es decir, sistemas operativos, hardware y software (Kim y Solomon, 2016).

Existen tres principios fundamentales de la seguridad informática que garantizan que solo los usuarios autorizados puedan realizar ciertas acciones. El primer principio es la confidencialidad, que consiste en que la información solo sea visible cuando sea estrictamente necesario y solo para el personal autorizado (Kim y Solomon, 2016).

El segundo principio es la integridad, que garantiza que los datos solo puedan ser modificados por los usuarios que legítimamente tengan la autoridad para hacerlo, para mantener la

exactitud y validez de la información (Kim y Solomon, 2016). El último principio es la disponibilidad, que se encarga de que los datos solo se encuentren disponibles para el personal autorizado que requiere su acceso (Kim y Solomon, 2016).

Para proteger los sistemas informáticos y tomar las medidas necesarias de prevención o mitigación, primero es necesario analizar los conceptos y diferencias entre riesgo, amenaza y vulnerabilidad (Kim y Solomon, 2016).

El riesgo se define como la probabilidad de que un evento incierto pueda ocurrir, afectando negativamente a uno o más elementos de un sistema informático, una amenaza es toda acción que pueda comprometer o dañar al sistema y las vulnerabilidades son todas las debilidades internas que existen en el sistema o que pueden ser producidas por los usuarios que lo utilizan y la infraestructura en la que se encuentra (Kim y Solomon, 2016).

2.8. SEGURIDAD EN REDES

La seguridad en redes se puede definir como el conjunto de tecnologías, procesos, modelos y configuraciones que tienen el objetivo de proteger la confidencialidad, integridad y accesibilidad de todos los dispositivos y elementos, tanto físicos como lógicos, de una red (Forcepoint, 2022).

2.8.1. Amenazas en redes

En el ámbito de la seguridad en redes, una amenaza es cualquier acción que tiene la capacidad de causar daños a los dispositivos que forman parte una determinada red (Kim y Solomon, 2016).

2.8.1.1. Ataques de denegación de servicio

Los ataques de denegación de servicio, conocidos como DoS por sus siglas en inglés, pueden ser de tipo lógico, cuando se explotan las fallas de un programa para impedir el funcionamiento de servidores remotos, o de inundación, cuando se envía una gran cantidad de solicitudes innecesarias a un sistema o a un recurso de red, provocando que los usuarios sean incapaces de acceder (Kim y Solomon, 2016).

Un ataque de denegación de servicio distribuido o DDoS puede definirse como un ataque de DoS a gran escala, debido a que se realizan con el mismo propósito, utilizando millones de dispositivos que, sin el conocimiento de los usuarios, ejecutan ataques automatizados para enviar solicitudes que producen que el tráfico legítimo sea bloqueado, causando que un sistema, red o un servicio sea inaccesible para los usuarios (Kim y Solomon, 2016).

2.8.1.2. Escaneo de puertos

El escaneo de puertos es considerado un ataque cuando se realiza sin autorización y consiste en utilizar herramientas para descubrir puertos abiertos o servicios y aplicaciones que se encuentran habilitadas utilizando la IP del dispositivo (Kim y Solomon, 2016).

2.8.1.3. Suplantación de ARP

Los ataques de suplantación o *spoofing* ocurren cuando un usuario, programa o sistema obtiene acceso no autorizado a un recurso mediante cambios para ocultar su origen e identidad, como sucede en los ataques de suplantación de ARP o protocolo de resolución de direcciones, en los cuales se envían falsas respuestas con diferentes direcciones MAC, provocando que se envíe el tráfico de red y sea interceptado (Kim y Solomon, 2016).

2.9. REDES DE ÁREA LOCAL

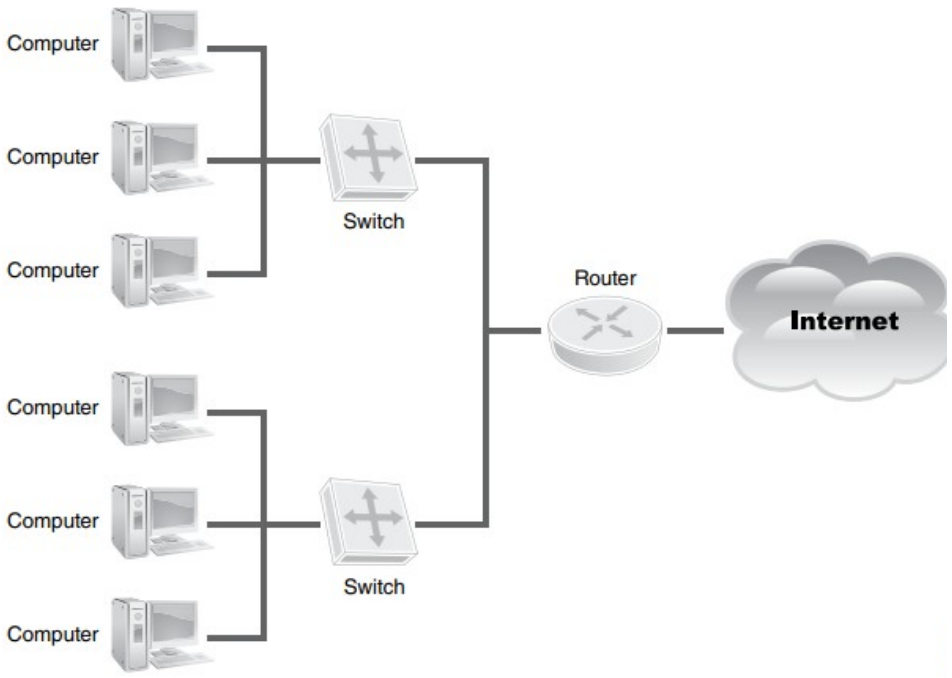
Las redes de área local (LAN) proporcionan conectividad de red a diferentes dispositivos que se encuentran en el mismo espacio geográfico, utilizando dispositivos especializados para la conexión entre diferentes redes, llamados *routers* y para la conexión entre dispositivos de una red, denominados *switches* (Kim y Solomon, 2016).

Las redes de área local inalámbricas (WLAN) emplean transmisiones de radio para la conectividad de red, en lugar de utilizar conexiones por cable, lo cual hace que sean más vulnerables que las redes LAN, debido a que es posible obtener acceso solo estando dentro del alcance de la red, sin la necesidad de obtener acceso físico (Cisco, 2022).

En la figura que se muestra a continuación, se observa una red de área local formada por un *router* y dos *switches.*

Figura 12

Red de área local



Nota. En la figura se muestra una red de área local. Obtenido en *Fundamentals of Information Systems Security,* por David Kim y Michael G. Solomon, 2016.

2.10. SISTEMAS DE DETECCIÓN DE INTRUSOS

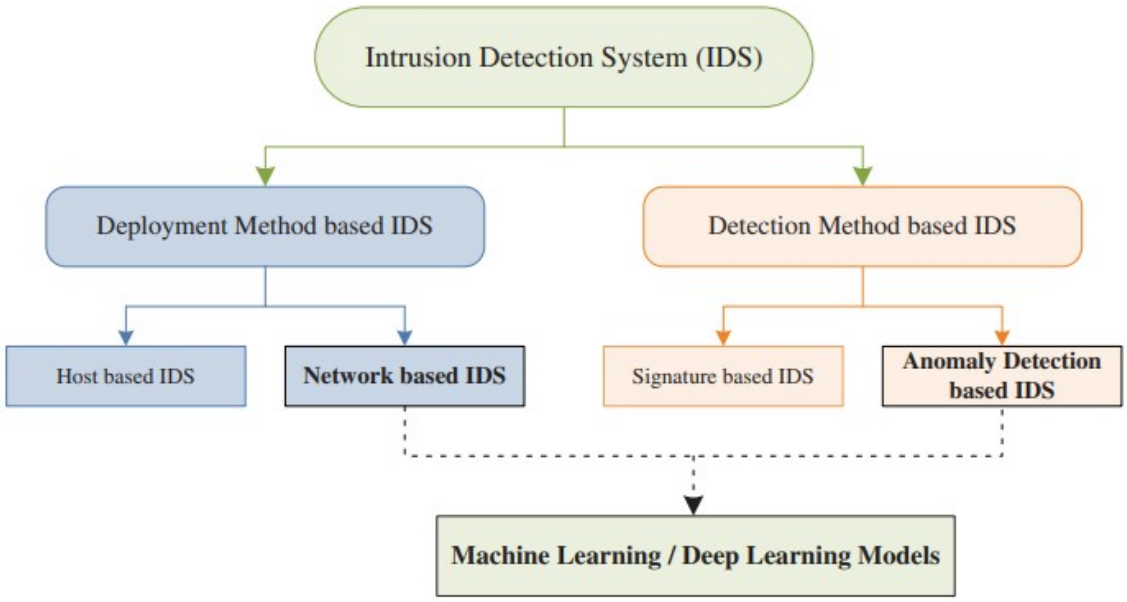
Un IDS es un sistema que se encarga de monitorear y analizar en tiempo real la actividad de una red o de un sistema informático para alertar la posible existencia de amenazas hacia las prácticas de seguridad estándar, políticas de seguridad y políticas del uso aceptable (Shimeall y Spring, 2014).

Los sistemas de detección basados en redes, conocidos como NIDS, analizan el tráfico de red desde un punto de congestión, también llamado *choke point*, en el que fluye la mayor parte del tráfico de la red (Shimeall y Spring, 2014). En contraste, los sistemas de detección basados en el *host*, llamados HIDS, monitorean los procesos de un solo sistema informático, denominado *host* (Kim y Solomon, 2016).

En la siguiente figura, se muestran las dos formas principales de clasificar los sistemas de detección de intrusos y sus respectivas divisiones.

Figura 13

Clasificación de los sistemas de detección de intrusos



Nota. En la figura se muestra la clasificación de los sistemas de detección de intrusos según el método de despliegue y de detección. Obtenido en *Network intrusion detection system: A systematic study of machine learning and deep learning approaches. Transactions on Emerging Telecommunications Technologies*, por Ahmad, Z., Khan, A., Shiang, C. W., Abdullah, J. y Ahmad, F. J., 2021.

Dentro de la clasificación según la estrategia de análisis, los sistemas de detección basados en firmas (SIDS) dependen de bases de datos de firmas y conocimiento de patrones de ataques analizados previamente y la detección basada en anomalías (AIDS) compara una línea base definida de comportamiento normal con la actividad analizada para identificar acciones fuera del rango definido (Ahmad et al., 2021).

Los sistemas de detección de intrusos basados en redes y que utilizan el análisis basado en anomalías se benefician del uso de modelos de aprendizaje automático y aprendizaje profundo (Ahmad et al., 2021).

2.11. HERRAMIENTAS

2.11.1. Wireshark

La herramienta de análisis de paquetes de red *Wireshark* se utiliza para realizar la captura de paquetes de diferentes medios de red tiempo real, de manera que sea posible visualizar los datos de forma detallada para poder diagnosticar y solucionar problemas de red, identificar problemas de seguridad y aprender sobre el funcionamiento de los protocolos de red (Wireshark, s.f.).

2.11.2. Máquinas virtuales

La herramienta de código abierto de virtualización de *Oracle VM VirtualBox* permite la ejecución de diferentes sistemas operativos desde *hosts* de *Windows*, logrando obtener una red con varios dispositivos conectados utilizando un solo equipo físico (Oracle VM VirtualBox, s.f.).

2.11.2.1. Kali Linux

Kali Linux es una distribución de *Linux* de código abierto basada en *Debian*, ampliamente utilizada en el ámbito de la seguridad informática, ya que contiene una gran variedad de herramientas instaladas para realizar pruebas de penetración o *Penetration Testing*, auditorías de seguridad, investigaciones forenses y pruebas de red entre otras aplicaciones en el ámbito académico y profesional (Kali Linux Documentation, s.f.).

2.11.2.2. Ubuntu

El sistema operativo de código abierto *Ubuntu*, basado en *Linux*, cuenta con un *firewall* integrado, programas de protección y actualizaciones de seguridad y soporte, convirtiéndolo en una opción para su uso en investigaciones dado sus requerimientos accesibles (Ubuntu, s.f.).

2.11.3. Python

El lenguaje de programación de código abierto *Python* permite la integración de diferentes módulos de terceros y librerías que permiten su uso en el desarrollo de interfaces gráficas,

desarrollo web, desarrollo de software, análisis de datos, desarrollo de modelos de aprendizaje automático y aprendizaje profundo entre otras aplicaciones (Python.org, s.f.).

2.11.3.1. TensorFlow

TensorFlow es una plataforma de código abierto para simplificar el desarrollo de modelos de aprendizaje automático, mediante su ecosistema flexible que facilita el uso de librerías complementarias para el procesamiento y validación de los datos y la API de aprendizaje profundo *Keras*, logrando la creación de modelos de aprendizaje automático independiente del lenguaje y del entorno (TensorFlow, s.f.).

2.11.3.2. Keras

La API para el desarrollo de alto nivel de modelos de aprendizaje profundo *Keras* permite facilitar la elaboración de modelos mediante interfaces intuitivas y fáciles de utilizar, minimizando la cantidad de pasos necesarios para realizar acciones comunes y ofreciendo una amplia documentación y guías (Keras, s.f.).

2.11.4. Anaconda Navigator

La herramienta de escritorio *Anaconda Navigator* cuenta con una interfaz gráfica de usuario que permite utilizar y gestionar diferentes entornos de desarrollo, paquetes y utilidades, mayormente utilizados para la ciencia de datos, como *Jupyter Notebook* y *Spyder*, entre otros (Anaconda Navigator, s.f.).

2.11.4.1. Jupyter Notebook

Jupyter Notebook es una aplicación de computación interactiva mediante *notebooks*, que permiten ejecutar código, visualizar datos y exportar el contenido (Jupyter Notebook, s.f.).

2.11.4.2. Spyder

El entorno científico de código abierto *Spyder* es una herramienta de desarrollo con funciones avanzadas para la exploración, análisis y visualización de los datos (Spyder, s.f.).

CAPÍTULO 3

MARCO APLICATIVO

CAPÍTULO 3
MARCO APLICATIVO

3.1. RECOLECCIÓN DE INFORMACIÓN

Las redes neuronales convolucionales y recurrentes presentan diferentes aplicaciones, las cuales pueden variar según su arquitectura y el tipo de dato de entrada que aceptan, por lo cual, es necesario comparar sus características para determinar su utilidad en la detección de intrusos en redes, como se muestra en la siguiente tabla.

Tabla 2

Características de las redes neuronales

Modelo	Arquitecturas	Funciones de activación	Datos de entrada	Aplicaciones
CNN	GAN ResNet Inception Network	ReLU Leaky ReLU ELU Softmax	Imágenes Videos Audio Texto Números	Clasificación Detección Reconocimiento Transferencia de estilo
RNN	GRU LSTM BRNN DRNN	ReLU TanH Sigmoid Softmax	Texto Audio Series de tiempo Imágenes Números	Clasificación Generación de secuencias Traducción Procesamiento de lenguaje natural

Nota. En la tabla se realiza la comparación de características y aplicaciones de las redes neuronales convolucionales y recurrentes. Elaborado con base en *Convolutional Neural Networks cheatsheet* y *Recurrent Neural Networks cheatsheet* por Amidi, A.y Amidi, S.

Como se muestra en la Tabla 2, en donde se listan las arquitecturas, funciones de activación y datos de entrada comúnmente utilizados para cada tipo de red neuronal, al igual que sus aplicaciones habituales, las redes neuronales convolucionales y recurrentes pueden realizar tareas de clasificación, binaria o múltiple, basadas en datos de entrada numéricos y de texto, por lo cual, es posible entrenar los modelos utilizando capturas de tráfico de red para su aplicación en la detección de intrusos en redes, ya sea en modelos separados o en una red híbrida CNN-RNN.

En la tabla que se muestra a continuación, se resumen las principales características y conclusiones de tres investigaciones, realizadas en los últimos años, sobre la aplicación de modelos híbridos convolucionales recurrentes en la detección de intrusos según el tipo de clasificación, el *dataset* utilizado para el entrenamiento, los parámetros para la evaluación y los resultados obtenidos.

Tabla 3

Investigaciones de redes híbridas convolucionales recurrentes

Referencia	Modelos	Clasificación	Dataset	Parámetros	Resultados
Intrusion detection system in the advanced metering Infrastructure: a Cross-Layer Feature-Fusion CNN-LSTM-Based approach, por Yao, R., Wang, N., Liu, Z., Chen, P. y Sheng, X. (2021).	CNN-LSTM	Múltiple	KDD Cup 1999	Exactitud	99.70
				DR	99.60
DL-IDS: Extracting features using CNN-LSTM hybrid network for Intrusion Detection System, por Sun, P., Liu, P., Li, Q., Liu, C., Lu, X., Hao, R. y Chen, J. (2020).	CNN-LSTM	Múltiple	CICIDS2017	Exactitud	98.67
				TPR	97.21
				FPR	0.47
				F1-score	93.32
1D CNN based network intrusion detection with normalization on imbalanced data, por Meliboev, A., Alikhanov, J. y Kim, W. (2020).	CNN-LSTM	Binaria	UNSW NB15 (Balanced)	Exactitud	89.93
				Precisión	86.15
				Exhaustividad	95.15
				F1-Score	90.43
			UNSW NB15 (Imbalanced)	Exactitud	85.11
				Precisión	79.57
				Exhaustividad	98.16
				F1-Score	87.89

Nota. En la tabla se detallan tres investigaciones sobre modelos híbridos convolucionales recurrentes para la detección de intrusos según su tipo de clasificación, *dataset* utilizado, parámetros y resultados. Elaborado con base en las siguientes investigaciones: *Intrusion detection system in the advanced metering Infrastructure: a Cross-Layer Feature-Fusion CNN-LSTM-Based approach*, por Yao, R., Wang, N., Liu, Z., Chen, P. y Sheng, X., 2021, *DL-IDS: Extracting features using CNN-LSTM hybrid network for Intrusion Detection System*, por Sun, P., Liu, P., Li, Q., Liu, C., Lu, X., Hao, R. y Chen, J., 2020 y *1D CNN based network intrusion detection with normalization on imbalanced data*, por Meliboev, A., Alikhanov, J. y Kim, W., 2020.

En la primera investigación de la Tabla 3, se utilizaron los *datasets KDD Cup 1999* y *NSL-KDD* para el entrenamiento de diferentes modelos de aprendizaje profundo basados en redes convolucionales y en la arquitectura LSTM de las redes recurrentes, en donde solo se tomaron en cuenta los resultados de clasificación múltiple del 99.7% de exactitud y una tasa de detección (DR) del 99.6% de la red híbrida CNN-LSTM entrenada con el *dataset KDD Cup 1999*, debido a que no se obtuvieron resultados concluyentes para la red con los demás parámetros ni con el segundo *dataset* utilizado (Yao et al., 2021).

En la segunda investigación se realizó la clasificación múltiple utilizando el *dataset CICIDS2017* para el entrenamiento de tres modelos de redes neuronales, una red convolucional, una red recurrente con arquitectura LSTM y una red híbrida CNN-LSTM, obteniendo como resultados para la última red una exactitud del 98.67%, una tasa de verdaderos positivos (TPR) del 97.21%, una tasa de falsos positivos (FPR) del 0.47% y una puntuación *F1-score* del 93.32% (Sun et al., 2020).

La última investigación utiliza el *dataset UNSW NB15* para realizar la clasificación binaria entre tráfico normal y el tráfico de ataques sin diferenciarlos, utilizando redes neuronales de una capa, dos capas, tres capas, una red híbrida CNN-LSTM, bosque aleatorio (RF) y máquinas de vectores de soporte (SVM) para realizar la evaluación de los modelos utilizando datos balanceados, cuando el número de datos de ambas clases se asemeja y no balanceados, en donde existe una gran diferencia en la cantidad de datos de tráfico normal y de los ataques (Meliboev et al., 2020).

Los resultados mostraron que se obtuvo una exactitud del 89.93%, precisión del 86.15%, exhaustividad del 95.15% y *F1-score* del 90.43% con los datos balanceados, mientras que con los datos no balanceados se alcanzó una exactitud del 85.11%, precisión del 79.57%, exhaustividad del 98.16% y *F1-score* del 87.89% (Meliboev et al., 2020).

La información recolectada sobre los modelos de redes neuronales seleccionados y su aplicación en la detección de intrusos permite determinar que es posible obtener una alta precisión de detección de anomalías en redes aplicando los modelos de redes neuronales convolucionales, recurrentes e híbridos convolucionales recurrentes para la clasificación múltiple de diferentes tipos de ataques provenientes de un *dataset* no balanceado de tráfico de red, conformado por datos numéricos y de texto.

3.2. EVALUACIÓN PRELIMINAR DE LAS REDES NEURONALES

La evaluación preliminar se realizó en el lenguaje de programación *Python* en *Jupyter Notebook*, utilizando el 10% del *dataset KDD Cup 1999*, que contiene un total de 145,586 datos obtenidos a partir de intrusiones simuladas en una red, con 41 columnas para los datos de entrada y 23 clases en la columna de etiquetas, de las cuales 22 representan diferentes tipos de ataques y la última el tráfico de red normal (*KDD Cup 1999 Data*, 2018).

Para elaborar los tres modelos de redes neuronales se comenzó por cargar los datos, asignar los nombres de las columnas y eliminar los valores duplicados y filas con valores faltantes para evitar errores. Luego, se realizaron los mismos procedimientos de preparación de los datos e ingeniería de características para los tres modelos, comenzando por codificar los valores numéricos para optimizar los modelos utilizando la normalización de puntuación Z, como se muestra en la siguiente figura.

Figura 14

Normalización por puntuación Z

```
function EncodeNumeric(dataframe, name):

    if mean is null:
        mean = CalculateMean(dataframe(name))

    if sd is null:
        sd = CalculateSD(dataframe(name))

    dataframe(name) = (dataframe(name)-mean)/sd
```

Nota. Pseudocódigo de la función de ingeniería de características para la normalización por puntuación Z. Elaboración propia en Carbon.now.sh.

En la tabla que se muestra a continuación, se listan todas las columnas del *dataset* que contienen valores numéricos y que requirieron la aplicación de la función de normalización por puntuación Z de la Figura 14.

Tabla 4

Columnas del dataset KDD Cup 1999 con valores numéricos

Columnas con datos numéricos	
duration	serror_rate
src_bytes	srv_serror_rate
dst_bytes	rerror_rate
wrong_fragment	srv_rerror_rate
urgent	same_srv_rate
hot	diff_srv_rate
num_failed_logins	srv_diff_host_rate
num_compromised	dst_host_count
root_shell	dst_host_srv_count
su_attempted	dst_host_same_srv_rate
num_root	dst_host_diff_srv_rate
num_file_creations	dst_host_same_src_port_rate
num_shells	dst_host_srv_diff_host_rate
num_access_files	dst_host_serror_rate
num_outbound_cmds	dst_host_srv_serror_rate
count	dst_host_rerror_rate
srv_count	dst_host_srv_rerror_rate

Nota. En la tabla se listan las columnas del *dataset* que contienen valores numéricos y utilizaron la función de normalización por puntuación Z. Elaboración propia basada en las columnas del *dataset KDD Cup 1999*, obtenido en Kaggle, 2018.

Como se muestra en la Tabla 4, se le aplicó la función de normalización por puntuación Z a 34 de las 41 columnas de entrada, con el objetivo de mejorar el rendimiento de los tres modelos de redes neuronales al regularizar los valores extremos identificados en las columnas de características.

Dado que no es posible entrenar los modelos de redes neuronales utilizando valores de texto sin procesar, se aplicaron técnicas de codificación a las columnas categóricas restantes para transformarlas en valores numéricos, utilizando *One Hot Encoding* y variables ficticias como se muestra en la siguiente figura, en donde se observa el pseudocódigo de la función para realizar la codificación de texto.

Figura 15

Ingeniería de características para valores de texto

```
function EncodeText(dataframe, name):

    DummyVar = dataframe(name)

    for i in DummyVar:
        dummyname = name + "-" + i
        dataframe(dummyname) = DummyVar(i)

    dataframe.drop(name)
```

Nota. Pseudocódigo de la función de codificación para valores de texto utilizando *One Hot Encoding* y variables ficticias. Elaboración propia en Carbon.now.sh.

En la siguiente tabla, se muestran las siete columnas categóricas restantes, que utilizaron la función de codificación para valores de texto de la Figura 15.

Tabla 5

Columnas del dataset KDD Cup 1999 con valores de texto

Columnas con valores de texto
protocol_type
service
flag
land
logged_in
is_host_login
is_guest_login

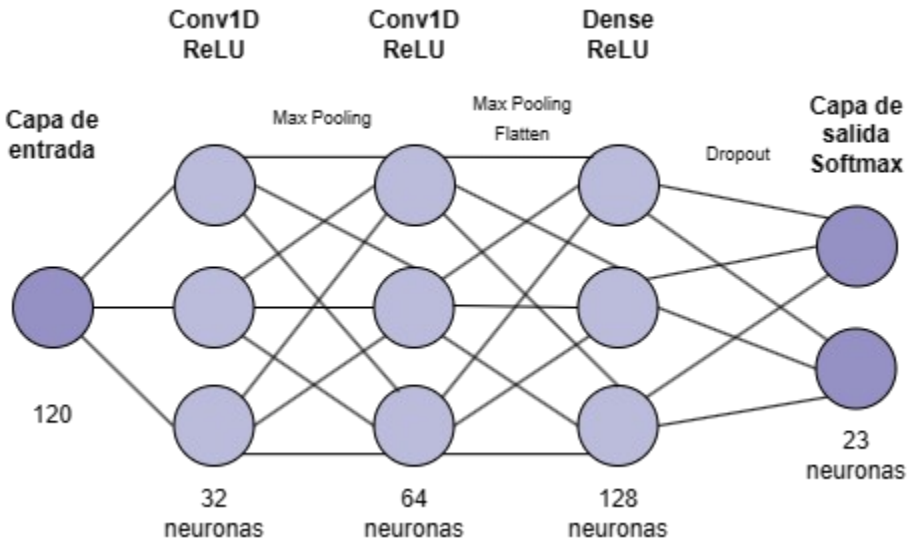
Nota. En la tabla se listan las columnas que contienen valores de texto y utilizaron la función de codificación, excluyendo a la columna que guarda las etiquetas para la clasificación. Elaboración propia basada en las columnas del *dataset KDD Cup 1999*, obtenido en Kaggle, 2018.

La función de codificación de texto consiste en la creación de variables ficticias con valores numéricos y la aplicación de la técnica de *One Hot Encoding*, de manera que se genere una nueva columna por cada valor diferente existente, provocando que se designen un total de 120 columnas de entrada. Posteriormente, se realizó la definición de las variables y se aplicó la codificación por variables ficticias a la columna *outcome* de manera individual para que se mantenga como una sola columna y se pasó a la división de los datos en entrenamiento y prueba, que se realizó utilizando la librería *Scikit-Learn*, en donde se utilizó un tamaño de prueba del 25%, lo cual implica que el restante 75% de los datos se utilizará para el entrenamiento de los modelos.

La siguiente etapa consiste en el modelado de las redes neuronales, comenzando por la red convolucional, como se muestra en el siguiente diagrama.

Figura 16

Diagrama de la red neuronal convolucional preliminar



Nota. La figura muestra el diagrama de la red neuronal convolucional preliminar. Elaboración propia utilizando la herramienta en línea draw.io.

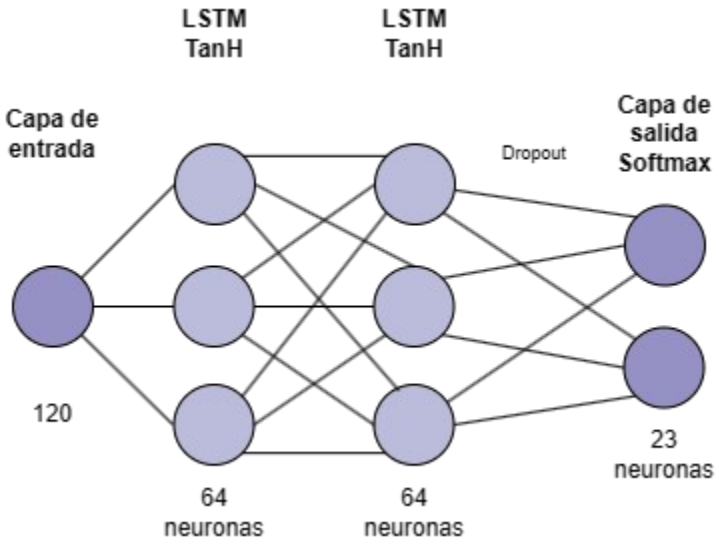
En la Figura 16, se observa que el modelo tiene 120 categorías de entrada, una capa oculta convolucional de una dimensión con 32 neuronas, seguida de una capa de *Max Pooling* para reducir las dimensiones espaciales, otra capa oculta convolucional con 64 neuronas y sus

respectivas capas de *Max Pooling* y *Flatten,* una capa densa con 128 neuronas y 0.5 de *dropout* antes de la capa de salida, que obtiene las probabilidades de cada clase.

El siguiente modelo elaborado fue la red neuronal recurrente con memoria de largo corto plazo (LSTM), como se muestra a continuación.

Figura 17

Diagrama de la red neuronal recurrente preliminar



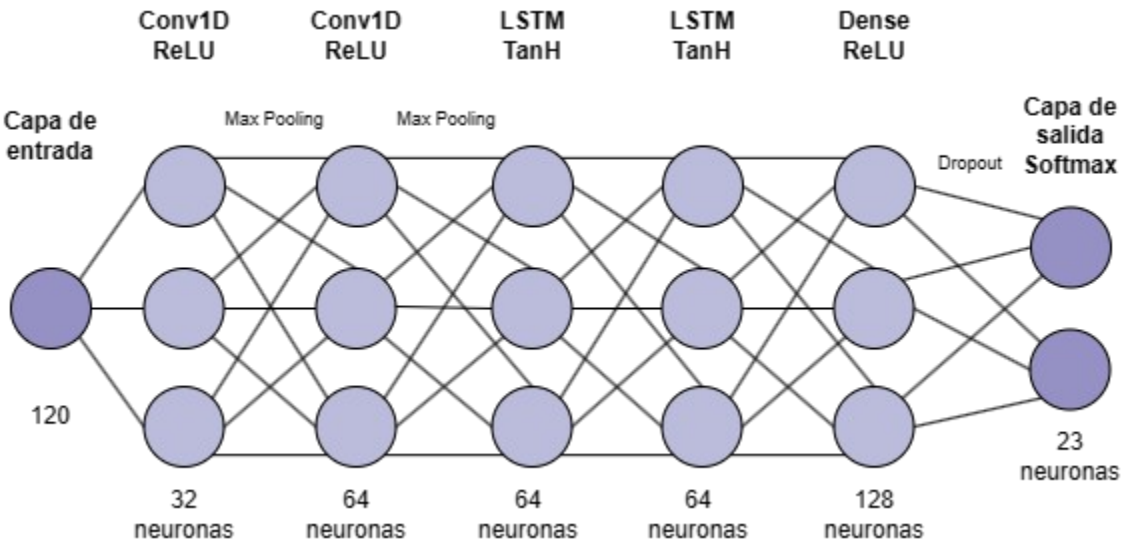
Nota. La imagen muestra el diagrama de la red neuronal recurrente con arquitectura LSTM preliminar. Elaboración propia utilizando la herramienta en línea draw.io.

Como se observa en la Figura 17, el modelo está formado por la capa de entrada de 120 categorías, dos capas ocultas de 64 neuronas cada una, que utilizan la función de activación de tangente hiperbólica (TanH) y 0.5 de *dropout* antes de la capa de salida, que tiene 23 salidas expresadas en la probabilidad de cada clase, mediante la función *softmax.*

El modelo híbrido convolucional recurrente, con 120 categorías de entrada, comienza con dos capas ocultas convolucionales de 32 y 64 neuronas cada una y sus respectivas capas de *Max Pooling*, seguidas de dos capas recurrentes de 64 neuronas cada una, una capa densa de 128 neuronas con 0.5 de *dropout* y la capa de salida con la función de activación *softmax* para la clasificación múltiple de las 23 salidas, como se muestra en el siguiente diagrama.

Figura 18

Diagrama de la red neuronal CNN-RNN preliminar



Nota. Diagrama de la red neuronal híbrida convolucional recurrente preliminar. Elaboración propia utilizando la herramienta en línea draw.io.

En la siguiente figura, se muestra el pseudocódigo de la función elaborada para calcular la precisión de las pruebas de los modelos, en donde TP representa los verdaderos positivos y FP los falsos positivos.

Figura 19

Función para calcular la precisión

```
function CalcPrecision(y_true, y_pred):

    TP = sum(y_true * y_pred)
    FP = sum((1 - y_true) * y_pred)
    precision = TP / (TP + FP + epsilon)
```

Nota. La figura muestra el pseudocódigo de la función para calcular la precisión. Elaboración propia en Carbon.now.sh.

En la siguiente imagen, se muestra el pseudocódigo de la función para calcular la exactitud o *accuracy* de las pruebas, en donde TN representa los verdaderos negativos y FN los falsos negativos.

Figura 20

Función para calcular la exactitud

```
function CalcAccuracy(y_true, y_pred):

    TP = sum(y_true * y_pred)
    TN = sum((1 - y_true) * (1 - y_pred))
    FP = sum((1 - y_true) * y_pred)
    FN = sum(y_true * (1 - y_pred))
    accuracy = TP + TN / (TP + TN + FP + FN + epsilon)
```

Nota. La figura muestra el pseudocódigo de la función para calcular la exactitud. Elaboración propia en *Carbon.now.sh.*

En la figura que se muestra a continuación, se observa el pseudocódigo de la función para calcular la exhaustividad o *recall* de las pruebas.

Figura 21

Función para calcular la exhaustividad

```
function CalcRecall(y_true, y_pred):

    TP = sum(y_true * y_pred)
    FN = sum(y_true * (1 - y_pred))
    recall = TP / (TP + FN + epsilon)
```

Nota. La figura muestra el pseudocódigo de la función para calcular la exhaustividad. Elaboración propia en Carbon.now.sh.

Posteriormente al modelado de las redes neuronales y la definición de las funciones para calcular los parámetros de evaluación seleccionados, se realizó la inicialización, compilación y entrenamiento de los tres modelos, utilizando la plataforma de código abierto de desarrollo de modelos de aprendizaje automático *TensorFlow,* junto con la API para aprendizaje profundo de alto nivel *Keras*, importando las librerías necesarias para cada modelo por separado en *Jupyter Notebook.*

Se comenzó por definir y construir los modelos secuenciales, agregar las capas con sus respectivas funciones de activación y utilizar el algoritmo de optimización de gradiente descendente *Adam* (*Adaptive Moment Estimation*), para finalmente pasar al entrenamiento de los tres modelos de redes neuronales, mediante la ejecución de 20 épocas para que los modelos ajusten sus pesos de manera iterativa y se minimice la pérdida, con un tamaño de lote o *batch size* de 256, para garantizar que el entrenamiento se realice de manera eficiente, considerando el tamaño del *dataset* y de los datos de entrada (ver Anexo A, Anexo B y Anexo C).

Una vez finalizado el entrenamiento de los tres modelos, se pasó a la evaluación preliminar de los parámetros calculados por las redes neuronales, comenzando por la elaboración de matrices de confusión, mediante el uso de la librería *Scikit-Learn,* para crear una función que permita personalizar la figura, de manera que se muestren todas las etiquetas de las clases existentes y sus resultados.

Debido a que el *dataset* utilizado para el entrenamiento presenta una distribución no balanceada y contiene clases con menos de diez muestras, provocando un desequilibrio significativo de clases frente a las que contienen más ejemplares como la clase normal con un total de 87,832 filas, solamente se agregaron las etiquetas, dentro de las matrices de confusión, para las clases que formaron parte de los conjuntos de prueba en el entrenamiento de cada modelo, dado que la partición aleatoria de los datos provocó que diferentes clases no se tomen en cuenta en cada modelo.

En la tabla que se muestra a continuación, se observa el conteo de las clases dentro de la columna *outcome,* realizado con el propósito de demostrar la distribución no balanceada de las clases y asistir en la verificación de las clases que fueron excluidas en el entrenamiento de cada modelo.

Tabla 6

Conteo de clases de la columna outcome

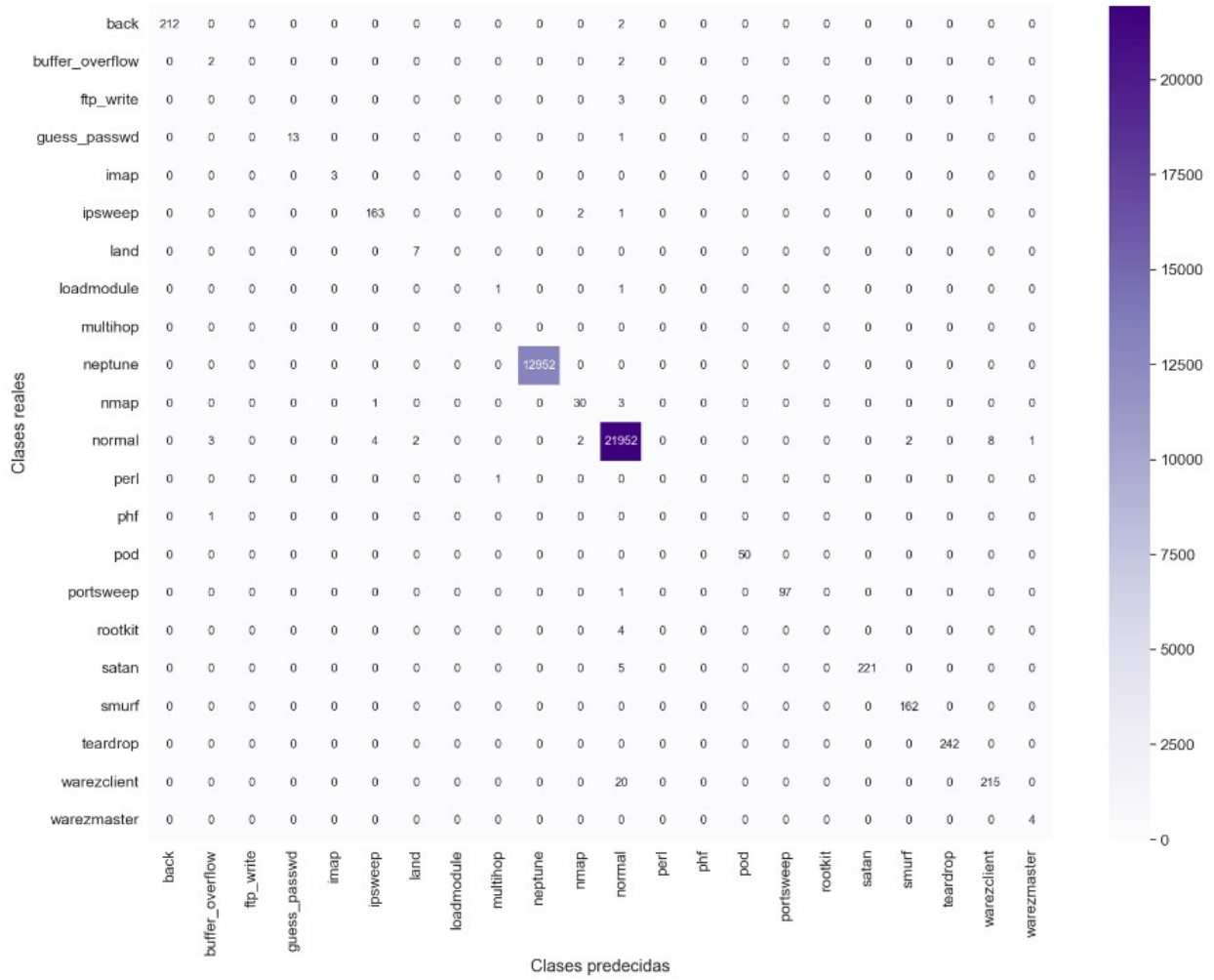
Clase	Total
back	968
buffer_overflow	30
ftp_write	8
guess_passwd	53
imap	12
ipsweep	651
land	19
loadmodule	9
multihop	7
neptune	51820
nmap	158
normal	87832
perl	3
phf	4
pod	206
portsweep	416
rootkit	10
satan	906
smurf	641
spy	2
teardrop	918
warezclient	893
warezmaster	20

Nota. En la tabla se muestra el conteo de clases de la columna *outcome*, que contiene las etiquetas para la clasificación múltiple. Elaboración propia basada en la columna de clases del 10% del *dataset KDD Cup 1999*, obtenido en Kaggle, 2018.

En la siguiente figura, se muestra la matriz de confusión con las etiquetas que formaron parte del conjunto de prueba durante el entrenamiento del modelo de red neuronal convolucional.

Figura 22

Matriz de confusión de la red CNN preliminar



Nota. En la figura se muestra la matriz de confusión de la red CNN preliminar, generada con *Scikit-Learn*. Elaboración propia en *Jupyter Notebook.*

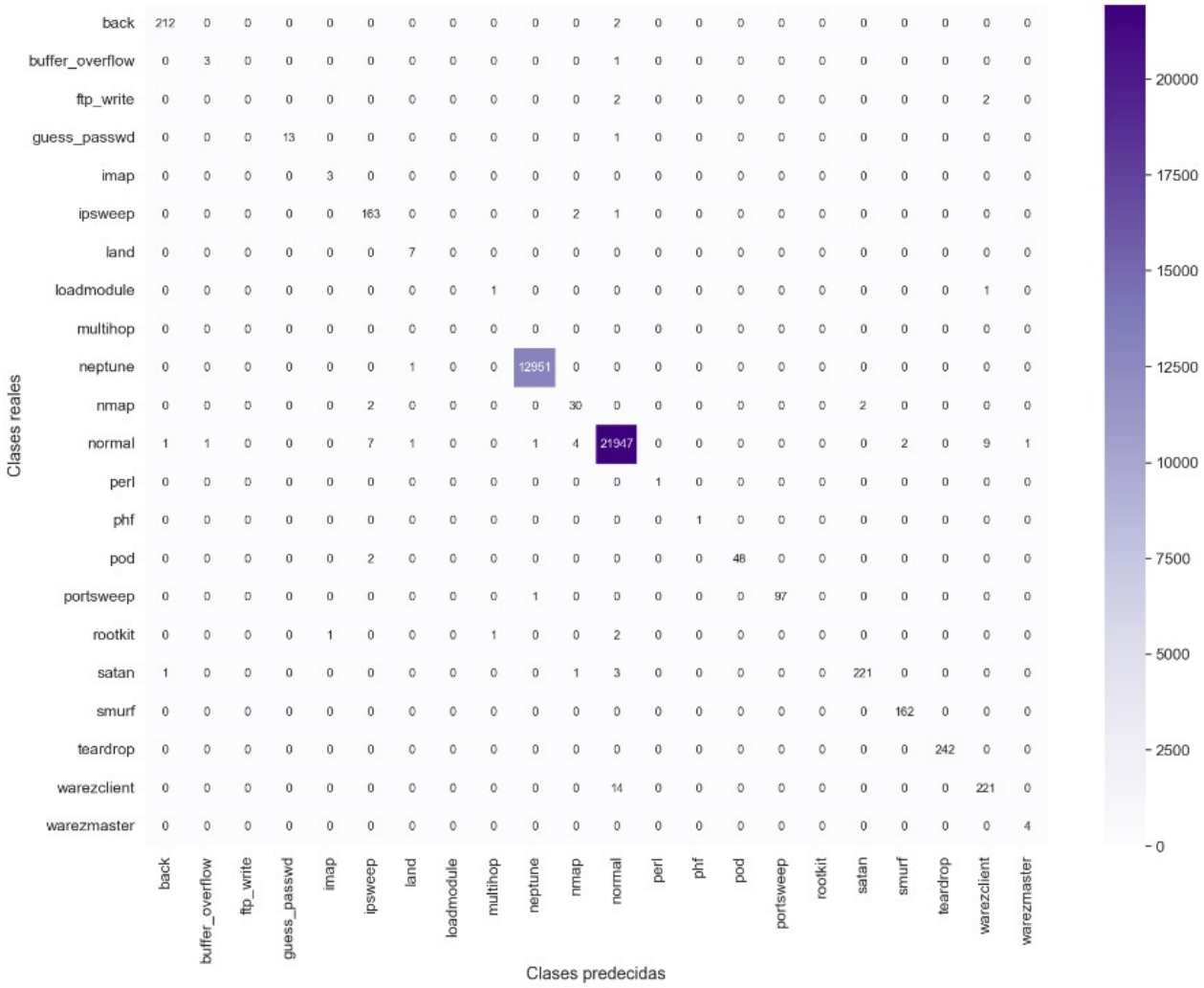
Como se observa en la Figura 22, los colores más oscuros en la matriz indican una mayor cantidad de ejemplares y predicciones por clase, en donde se muestra que las etiquetas más representadas dentro del conjunto de prueba fueron la clase normal con 21,952 y el ataque de

neptune con 12,952 aciertos. La clase que no formó parte del conjunto de prueba en la división de los datos de la red neuronal convolucional fue *spy,* con dos ejemplares en total.

A continuación, se muestra la matriz de confusión de la red neuronal recurrente para la evaluación preliminar.

Figura 23

Matriz de confusión de la red RNN preliminar



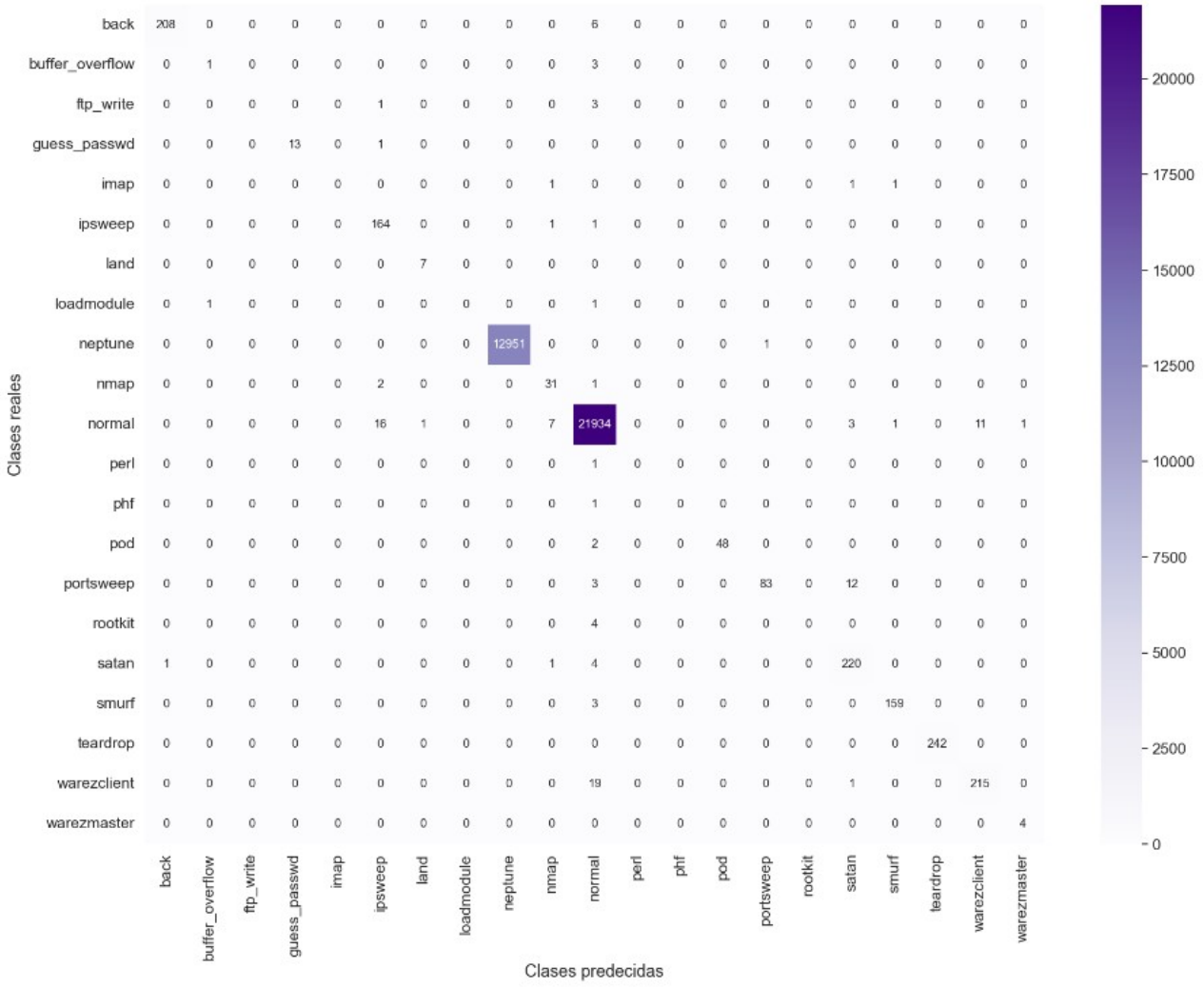
Nota. En la figura se muestra la matriz de confusión de la red RNN preliminar, generada con *Scikit-Learn*. Elaboración propia en *Jupyter Notebook*.

En la Figura 23, se muestra que las clases más representadas fueron *neptune* con 12,951 y la clase normal con 21,947 aciertos. Al igual que en el modelo convolucional, la clase que no formó parte del entrenamiento fue *spy,* por lo cual no se incluyó la etiqueta en la figura.

En la siguiente figura, se muestra la matriz de confusión de la red neuronal híbrida convolucional recurrente.

Figura 24

Matriz de confusión de la red híbrida CNN-RNN preliminar



Nota. En la figura se muestra la matriz de confusión de la red híbrida CNN-RNN preliminar generada utilizando *Scikit-Learn*. Elaboración propia en *Jupyter Notebook.*

Como se muestra en la matriz de la Figura 24, las clases que no formaron parte del entrenamiento fueron *spy,* con dos muestras y *multihop* con 7, mientras que las clases con una mayor cantidad de aciertos fueron, de manera similar que en los modelos anteriores, *neptune* y normal, con 12,951 y 21,934 muestras respectivamente.

Para la evaluación preliminar de los tres modelos de redes neuronales se tomaron en cuenta los parámetros de exactitud, precisión y exhaustividad de las pruebas, como se muestra en la siguiente tabla.

Tabla 7

Evaluación preliminar de las redes neuronales

Red neuronal	Exactitud	Precisión	Exhaustividad
CNN	0.9998	0.9982	0.9979
RNN	0.9998	0.9984	0.9980
CNN-RNN	0.9997	0.9972	0.9965

Nota. En la tabla se muestran los resultados de los tres modelos de redes neuronales para la evaluación preliminar. Elaboración propia.

Como se muestra en la Tabla 7, inicialmente se puede observar que los resultados obtenidos de los parámetros de exactitud, precisión y exhaustividad de las pruebas, expresados utilizando cuatro decimales, muestran cifras menores en el modelo híbrido CNN-RNN, en comparación con los valores obtenidos por los modelos CNN y RNN, que presentan estructuras más simples.

3.3. RECOLECCIÓN DE LOS DATOS

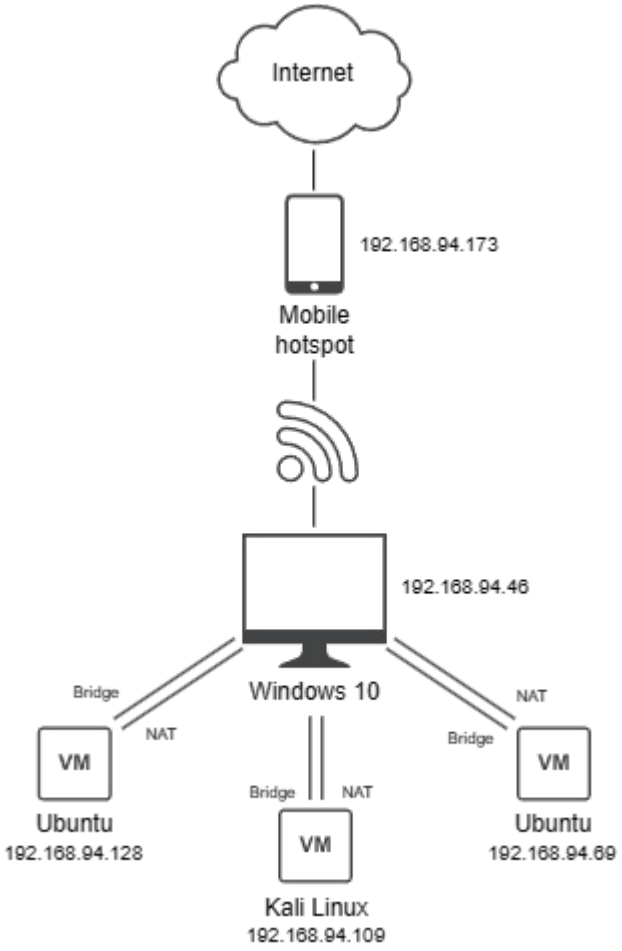
Para la fase de recolección de los datos es necesario comenzar con la configuración del entorno controlado, que consiste en una red de área local (LAN) que utiliza *hotspot* móvil como punto de acceso a Internet para establecer una conexión Wi-Fi con un equipo con *Windows 10* como sistema operativo, que actúa como anfitrión para las tres máquinas virtuales en *Oracle VM VirtualBox*, versión 7.0.10.

La primera máquina virtual se basa en una imagen prefabricada de *Kali Linux* versión 2023.3, mientras que las dos siguientes utilizan una imagen ISO de *Ubuntu* versión 22.04.3 LTS.

Dado que el equipo de *Windows 10* y las tres máquinas virtuales se encuentran en el mismo segmento de red, es posible utilizar el modo promiscuo de *Wireshark,* versión 4.0.8, para capturar los paquetes de los dispositivos que conforman la red LAN, como se muestra en el siguiente diagrama de red.

Figura 25

Diagrama de la red LAN



Nota. La figura muestra el diagrama de red del entorno controlado, que consiste en una red LAN con *hotspot* móvil como punto de acceso a Internet. Elaboración propia utilizando la herramienta en línea draw.io.

Como se muestra en la Figura 25, se configuraron dos adaptadores de red en cada una de las máquinas virtuales, con el objetivo de que tengan una dirección IP privada de clase C al igual que el equipo anfitrión. El primero es un adaptador puente o *bridge*, que debe tener el modo promiscuo habilitado en la opción de “Permitir todo” para la comunicación entre las máquinas virtuales con el anfitrión y el segundo adaptador es de tipo NAT, para que los dispositivos tengan acceso a Internet.

Luego de configurar el entorno, se ejecutó el comando *ping* desde el equipo anfitrión hacia cada una de las máquinas virtuales y entre ellas, comprobando que se encuentran conectadas entre sí. Sin embargo, no es posible la ejecución exitosa del comando desde las máquinas virtuales hacia el equipo de *Windows 10*, debido a las configuraciones de *firewall*.

En la siguiente tabla, se muestran las trece columnas utilizadas para las capturas en el programa *Wireshark.*

Tabla 8

Columnas en Wireshark

Columna	Descripción
No.	Número del paquete
Delta time	Tiempo transcurrido entre los paquetes
Src add	Dirección de origen del paquete
Dst add	Dirección de destino del paquete
Protocol	Nombre del protocolo
Length	Longitud del paquete en bytes
Src port	Puerto de origen sin resolver del paquete
Dst port	Puerto de destino sin resolver del paquete
Flags	Estado de la conexión TCP
Duplicate IP add	Existencia de dos o más dispositivos con la misma dirección IP
Severity	Nivel de severidad de la información experta del paquete
Info	Información del contenido del paquete
Window size	Tamaño de ventana TCP

Nota. Columnas predeterminadas y personalizadas utilizadas para las capturas de tráfico de red en el programa *Wireshark.* Elaboración propia.

Para que los modelos de redes neuronales puedan clasificar el tráfico de red a partir de las características extraídas de las capturas se agregaron columnas personalizadas mediante la configuración de preferencias en el programa, obteniendo un total de trece columnas junto con las columnas predeterminadas que se mantuvieron dentro de la lista de paquetes.

Las columnas de dirección de origen, dirección de destino y dispositivos con la misma dirección IP se procesarán para convertir su contenido en valores booleanos, mientras que las columnas de número del paquete e información adicional solo se utilizarán durante la recolección de los datos para verificar el envío y recibo de paquetes y se eliminarán en la elaboración del *dataset*, dado que contienen información que se encuentra representada de manera concisa dentro de las otras columnas y que puede resultar poco relevante.

El tiempo de duración de las capturas de tráfico de red normal se determinó con base en el tiempo de ruptura de 84 minutos, conocido como *breakout time*, calculado en el reporte global de amenazas de *CrowdStrike,* que se define como el promedio de tiempo transcurrido desde que un adversario obtiene acceso no autorizado a un equipo y el primer movimiento lateral desde dicho equipo hacia otros dispositivos dentro del entorno comprometido (CrowdStrike, 2023).

Para la determinación de los tiempos de duración de las capturas de los diferentes ataques se consideró la regla de *CrowdStrike* de 1-10-60, que hace referencia al tiempo recomendado en minutos dentro del cual es posible minimizar el daño causado por ataques dirigidos al detectar las amenazas durante el primer minuto y entenderlas dentro de 10 minutos, para que sea posible mitigar las consecuencias antes de llegar a los 60 minutos, manteniéndose dentro del tiempo de ruptura (CrowdStrike, 2023).

3.3.1. Tráfico de red normal

Para determinar la línea base de tráfico normal dentro de una red de área local se tomaron en cuenta los resultados del reporte publicado por *Data Reportal*, en colaboración con *We Are Social* y *Meltwater*, titulado *Digital 2023 Global Overview Report.*

El reporte consiste en una recolección de datos y estadísticas sobre el comportamiento de los usuarios a nivel global en relación a las tendencias tecnológicas emergentes, el uso de redes

sociales, comercio electrónico, marketing digital y al uso de diferentes medios digitales para adquirir información (We Are Social y Meltwater, 2023).

En la tabla que se muestra a continuación, se puede observar el tiempo promedio que los usuarios a nivel global, que se encuentran dentro del rango de edades estudiado de 16 a 64 años, dedican a realizar diferentes actividades en Internet y al uso de medios y plataformas digitales cada día, con base en los resultados mostrados en el reporte *Digital 2023 Global Overview Report.*

Tabla 9

Tiempo promedio diario de los usuarios dedicado al uso de medios digitales

Medio o actividad	Tiempo promedio
Cualquier uso de Internet	6 horas 37 minutos
Ver televisión por medio de transmisiones o servicios de *streaming*	3 horas 23 minutos
Uso de redes sociales	2 horas 31 minutos
Lectura de medios de prensa en línea y medios físicos	2 horas 10 minutos
Escuchar música en servicios de *streaming*	1 hora 38 minutos
Uso de consolas de juegos	1 hora 14 minutos
Escuchar *podcasts*	1 hora 2 minutos
Escuchar transmisiones de radio	59 minutos

Nota. En la tabla se muestra el tiempo promedio diario que los usuarios de 16 a 64 años dedican al uso de medios y plataformas digitales. Elaboración con base en el gráfico de la página 26 del reporte *Digital 2023 Global Overview Report*, publicado por *Data Reportal*, en colaboración con *We Are Social* y *Meltwater.*

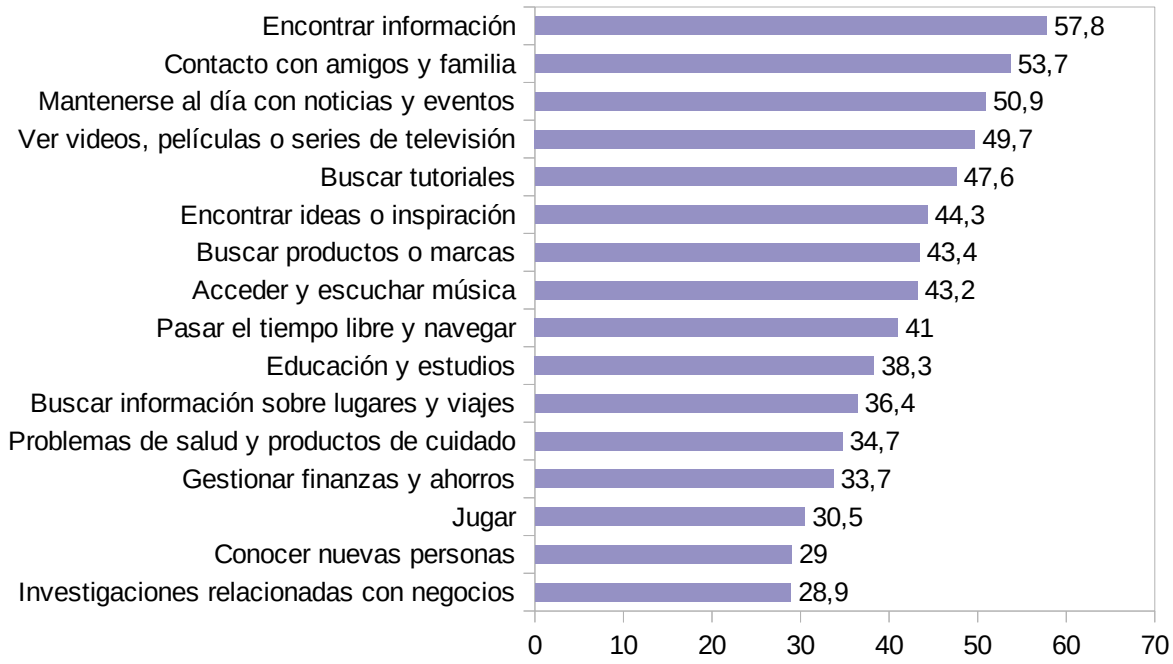
Como se muestra en la Tabla 9, las actividades más frecuentes, sin tomar en cuenta cualquier uso de Internet con un promedio de 6 horas y 37 minutos, relacionadas con el uso de plataformas y medios digitales a las que los usuarios a nivel global de 16 a 64 años le dedican más de una hora al día en promedio son: ver televisión por transmisiones y escuchar música o *podcasts* a través de servicios de *streaming*, interacciones en redes sociales, lectura de plataformas de noticias y el uso de consolas de juegos.

Con el propósito de ampliar la definición de la línea base del tráfico de red normal, a partir de la información recopilada sobre el comportamiento de los usuarios de 16 a 64 años, se

consideraron los principales motivos identificados para utilizar Internet, en donde se destacan las búsquedas generales de información, el contacto con amigos y familia, mantenerse al día con noticias y eventos, ver videos, series o películas mediante plataformas de *streaming,* escuchar música y jugar en línea, por su relación con las actividades mencionadas en la Tabla 9, como se muestra en el siguiente gráfico, elaborado con base en los resultados del reporte *Digital 2023 Global Overview Report.*

Figura 26

Principales motivos de los usuarios para utilizar Internet



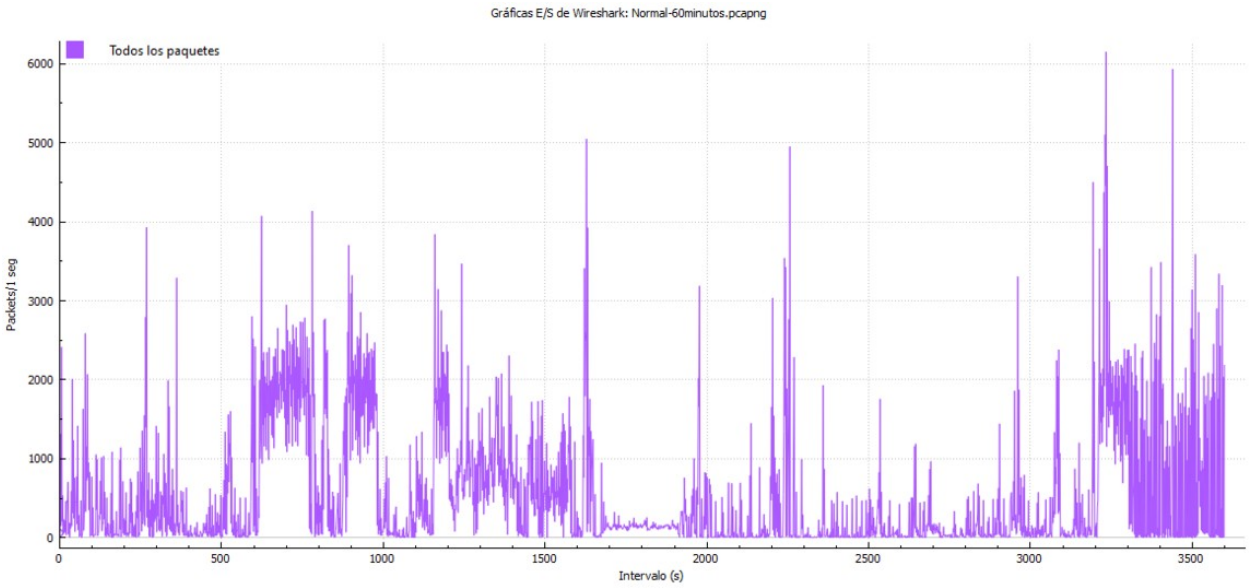
Nota. El gráfico muestra los principales motivos de los usuarios de 16 a 64 años para utilizar Internet. Los resultados del gráfico están basados en la página 63 del reporte *Digital 2023 Global Overview Report*, publicado por *Data Reportal*, en colaboración con *We Are Social* y *Meltwater*.

La captura de tráfico de red normal se realizó en el equipo anfitrión utilizando *Wireshark* de manera constante durante 60 minutos. Las acciones realizadas para la representación de la línea base de tráfico normal fueron las siguientes: búsquedas de información en Internet, interacciones en redes sociales, navegación en sitios de noticias, participar en juegos cooperativos, escuchar música y ver videos en plataformas de *streaming.*

En la siguiente figura, se muestra el gráfico de entrada y salida (E/S) de los 1,897,698 paquetes capturados de tráfico de red normal.

Figura 27

Gráfico E/S de tráfico de red normal



Nota. El gráfico de entrada y salida muestra todos los paquetes capturados por segundo durante la captura de tráfico de red normal de 60 minutos. Elaborado en el programa *Wireshark.*

En el gráfico de la Figura 27, se muestran todos los paquetes capturados durante 60 minutos, dado que no se aplicaron filtros para el análisis del tráfico de red normal, con el propósito de obtener resultados más variados y no limitar los datos incluidos logrando obtener una línea base para la clasificación de tráfico de red normal imparcial.

3.3.2. Tráfico de red de los ataques

3.3.2.1. Tráfico de red de escaneo de puertos

El escaneo de puertos se realizó utilizando la herramienta de código abierto *Nmap* para descubrimiento de redes, desde la máquina virtual de *Kali Linux*, mediante la ejecución de variaciones de los comandos para escaneo de puertos en dos capturas separadas.

El primer comando que se ejecutó durante la captura de tráfico de red de 12 minutos fue para un escaneo de puertos con valores predeterminados, que solo considera a los 1,000 puertos más utilizados, como se muestra a continuación.

Figura 28

Comando para el escaneo de puertos predeterminado

```
sudo nmap 192.168.94.1/24
```

Nota. El gráfico muestra el comando de *Nmap* utilizado para realizar el escaneo de puertos predeterminado a la subred /24. Elaboración propia en Carbon.now.sh.

En la Figura 28, se observa que se utiliza el comando *sudo* al inicio para no limitar las opciones del escaneo de puertos, seguido del comando *nmap* para realizar la búsqueda de puertos abiertos y servicios de los dispositivos de la subred /24.

En la siguiente figura, se muestran los comandos con el parámetro *-F*, que sirve para limitar la búsqueda a los 100 puertos más comunes, logrando que cada escaneo sea más rápido.

Figura 29

Comandos para el escaneo de 100 puertos

```
sudo nmap -F -sT 192.168.94.1/24
sudo nmap -F -sS 192.168.94.1/24
sudo nmap -F -sU 192.168.94.1/24
sudo nmap -F -sV 192.168.94.1/24
```

Nota. El gráfico muestra los comandos de *Nmap* utilizados para el escaneo de los 100 puertos más comunes dentro de la subred /24. Elaboración propia en Carbon.now.sh.

El primer comando de la Figura 29 utiliza el parámetro *-sT*, que especifica el uso de una conexión TCP completa. En el escaneo de la segunda orden, conocido como *stealth scan,* se envían paquetes de tipo TCP SYN mediante el parámetro *-sS.* El tercer comando ejecuta la

instrucción *-sU* para encontrar puertos abiertos que utilizan el protocolo UDP y el último realiza un escaneo para la detección de la versión de los servicios, agregando el parámetro *-sV.*

En el último conjunto de comandos se utiliza la orden *-p-* para llevar a cabo un escaneo completo de los 65,535 puertos existentes, como se observa en la siguiente figura.

Figura 30

Comandos para el escaneo de puertos completo

```
sudo nmap -p- 192.168.94.46
sudo nmap -p- 192.168.94.109
sudo nmap -p- 192.168.94.128
sudo nmap -p- 192.168.94.69
```

Nota. El gráfico muestra los comandos de *Nmap* para el escaneo de puertos completo. Elaboración propia en Carbon.now.sh.

Para la segunda captura de escaneo de puertos, que tuvo una duración de 140 minutos, se ejecutó un solo comando que utiliza el parámetro -p- para realizar el escaneo de puertos completo de las 256 direcciones IP de la subred /24, como se puede observar en la siguiente figura.

Figura 31

Comando para el escaneo de puertos completo de la subred

```
sudo nmap -p- 192.168.94.1/24
```

Nota. El gráfico muestra el comando de *Nmap* para el escaneo de puertos completo de la subred /24. Elaboración propia en Carbon.now.sh.

Para separar los datos generados por el escaneo de puertos de los datos de tráfico de red corriente en la primera captura se aplicaron los siguientes filtros en el programa *Wireshark.*

Figura 32

Filtros para la primera captura de escaneo de puertos

```
ip.src == 192.168.94.1/24
tcp.flags.syn == 1
tcp.flags.ack == 1
udp
ip.dst == 192.168.94.46
ip.dst == 192.168.94.109
ip.dst == 192.168.94.128
ip.dst == 192.168.94.69
```

Nota. En la figura se muestran los filtros utilizados en el programa *Wireshark* para la primera captura de escaneo de puertos. Elaboración propia en Carbon.now.sh.

Utilizando los filtros para el descubrimiento general de *hosts* de la subred /24, solicitudes de conexión TCP SYN, conexiones TCP, conexiones UDP y direcciones IP de destino específicas, se determinó que se utilizarán 4,236 de los 6,398 paquetes de la primera captura.

En la siguiente figura, se muestran los filtros utilizados en *Wireshark* para la segunda captura de escaneo de puertos.

Figura 33

Filtros para la segunda captura de escaneo de puertos

```
ip.src == 192.168.94.1/24
tcp.port
```

Nota. En la figura se muestran los filtros utilizados en el programa *Wireshark* para la segunda captura de escaneo de puertos. Elaboración propia en Carbon.now.sh.

Los filtros de la Figura 33 se aplicaron para separar todos los paquetes TCP que provienen de la subred /24 y determinar que 71,670 de los 87,042 paquetes capturados se utilizarán para el

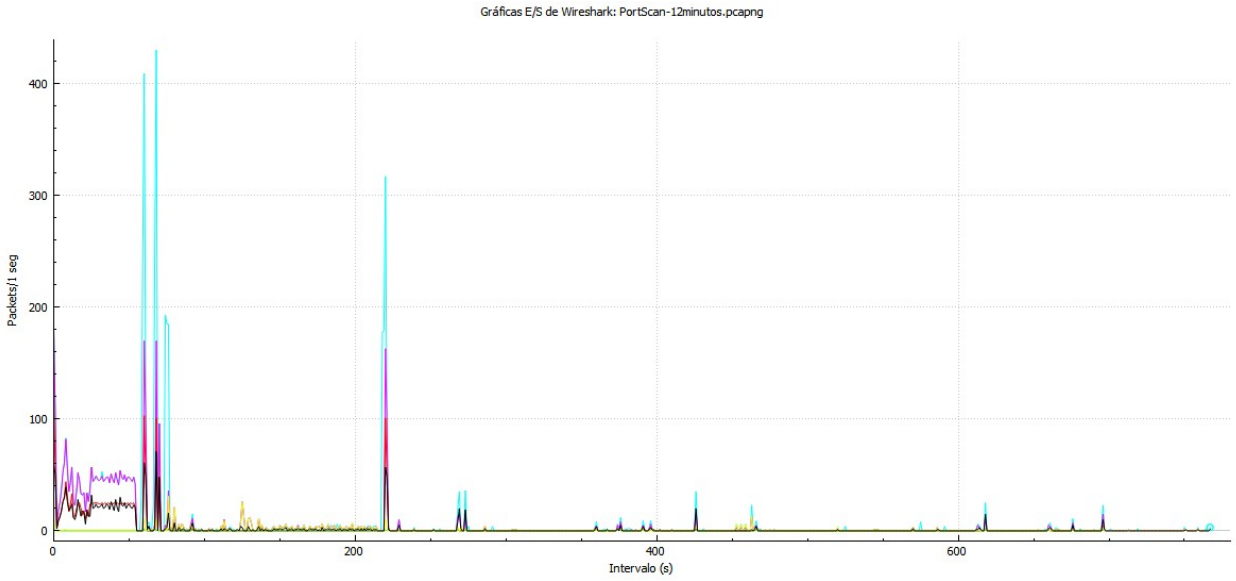
entrenamiento de las redes neuronales, junto con los datos de la primera captura, para la clasificación de tráfico red de escaneo de puertos frente al tráfico de red normal y a otros tipos de ataques.

Para tener una representación visual de la cantidad de paquetes que entran y salen durante las capturas y del flujo de datos que corresponden al escaneo de puertos en una red mediante el uso de filtros, se elaboraron gráficos de E/S para cada captura en *Wireshark.*

En la figura que se muestra a continuación, se observa el gráfico E/S de la primera captura de tráfico de red de escaneo de puertos.

Figura 34

Gráfico E/S de la captura de escaneo de puertos de 12 minutos

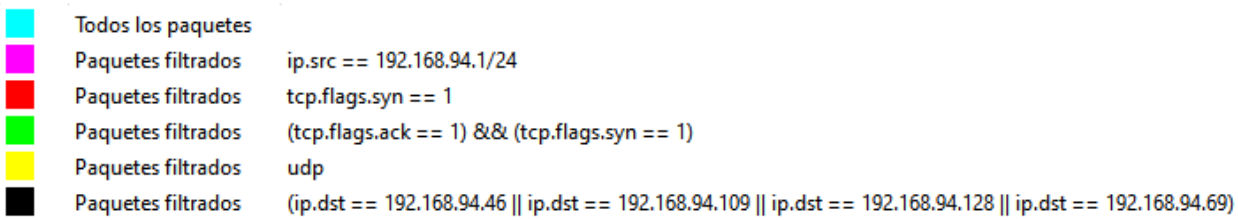


Nota. El gráfico E/S muestra la cantidad de paquetes enviados y recibidos dentro de un intervalo de tiempo de un segundo durante la primera captura de tráfico de escaneo de puertos de 12 minutos. Elaborado en el programa *Wireshark.*

En el gráfico de la Figura 34, se utilizaron los filtros mencionados anteriormente en donde cada color corresponde a un filtro específico, como se muestra en la siguiente leyenda del gráfico E/S.

Figura 35

Leyenda del gráfico E/S de la primera captura de escaneo de puertos

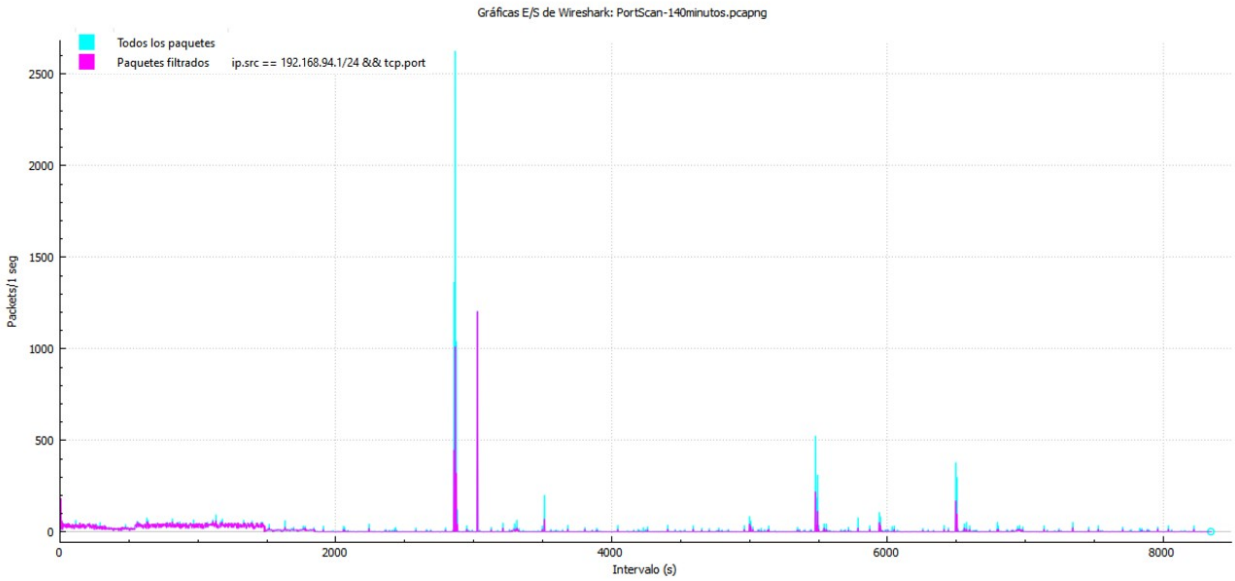


Nota. La figura muestra la leyenda correspondiente al gráfico E/S de la primera captura de tráfico de escaneo de puertos de 12 minutos. Elaborado en el programa *Wireshark.*

En la siguiente figura, se muestra el gráfico E/S y la leyenda de la segunda captura de tráfico de escaneo de puertos.

Figura 36

Gráfico E/S y leyenda de la segunda captura de escaneo de puertos



Nota. El gráfico E/S y la leyenda muestran la cantidad de paquetes enviados y recibidos por segundo durante la segunda captura de tráfico de escaneo de puertos de 140 minutos. Elaborado en el programa *Wireshark.*

3.3.2.2. Tráfico de red de ataques de denegación de servicio

Para la captura de tráfico de red que proviene de un ataque de denegación de servicio se utilizó la herramienta de envío de paquetes personalizados *HPing3* desde la máquina virtual *Kali Linux* hacia la puerta de enlace predeterminada, que corresponde a la dirección IP del dispositivo móvil que actúa como punto de acceso a internet, utilizando el comando que se muestra a continuación.

Figura 37

Comando para el ataque de denegación de servicio

```
sudo hping3 -S --flood 192.168.94.173
```

Nota. En la figura se muestra el comando de *HPing3* para el ataque de denegación de servicio. Elaboración propia en Carbon.now.sh.

En el comando de la Figura 37, se inicia con la orden *sudo* para elevar los privilegios a superusuario, luego se utiliza el comando de *hping3* con los parámetros *-S* y *–flood*, que indican que se realizará un ataque de inundación de paquetes TCP SYN hacia la dirección especificada.

Para separar los paquetes del ataque de denegación de servicio del tráfico normal en la captura, se filtraron las solicitudes de conexión TCP SYN en el programa *Wireshark*, aplicando el siguiente filtro.

Figura 38

Filtro para el ataque de inundación de paquetes TCP SYN

```
tcp.flags.syn == 1
```

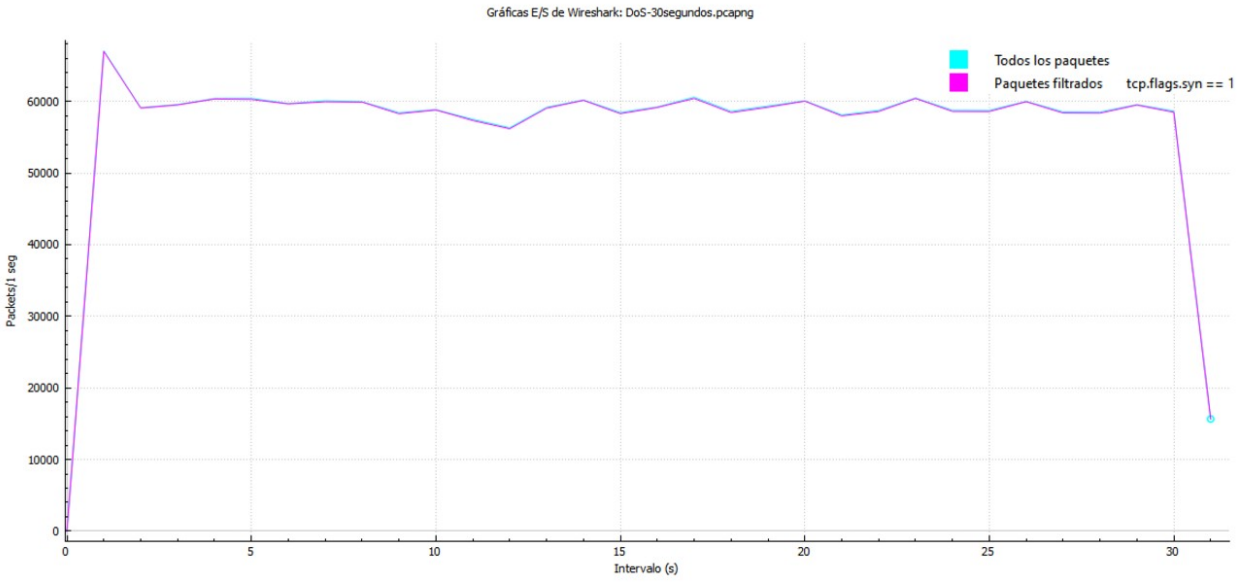
Nota. En la figura se muestra el filtro utilizado en el programa *Wireshark* para el ataque de inundación de paquetes TCP SYN. Elaboración propia en Carbon.now.sh.

La captura se realizó durante 30 segundos, obteniendo 1,798,447 paquetes en total de los cuales se determinó que 1,795,094 fueron generados por el ataque de denegación de servicio, utilizando el filtro de la Figura 38.

A continuación, se muestra el gráfico E/S de la captura de tráfico de red del ataque de denegación de servicio.

Figura 39

Gráfico E/S y leyenda de la captura del ataque de DoS



Nota. El gráfico E/S y la leyenda con el filtro muestran la cantidad de paquetes enviados y recibidos durante la captura de tráfico del ataque de denegación de servicio de 30 segundos. Elaborado en el programa *Wireshark.*

3.3.2.3. Tráfico de red de suplantación de ARP

El ataque de suplantación de ARP se realizó utilizando la herramienta *arpspoof* dentro del paquete *dsniff* instalado en la máquina virtual de *Kali Linux*, con el propósito de simular un ataque de MITM entre el equipo anfitrión y la puerta de enlace predeterminada, provocando que su dirección MAC se reemplace con la dirección de la máquina virtual de *Kali Linux*, mediante el uso de los comandos que se muestran en la siguiente figura, tomando en cuenta que cada comando se ejecutó en terminales separadas.

Figura 40

Comandos para el ataque de suplantación de ARP

```
sudo arpspoof -i eth0 -t 192.168.94.46 192.168.94.173
sudo arpspoof -i eth0 -t 192.168.94.173 192.168.94.46
```

Nota. Comandos utilizados en *Kali Linux* para el ataque de suplantación de ARP. Elaboración propia en Carbon.now.sh.

En el primer comando de la Figura 40, se define la interfaz con el parámetro -*i* y la dirección IP del objetivo del ataque con -*t,* seguida de la dirección de la puerta de enlace predeterminada, mientras que en el segundo comando se realiza el mismo procedimiento invirtiendo las direcciones IP. A continuación, se muestra el comando utilizado para habilitar el reenvío de paquetes de red.

Figura 41

Comando para habilitar el reenvío de paquetes

```
sysctl net.ipv4.ip_forward=1
```

Nota. En la figura se muestra el comando ejecutado en Kali Linux habilitar el reenvío de paquetes de red luego de iniciar el ataque de suplantación de ARP. Elaboración propia en Carbon.now.sh.

Los comandos ejecutados provocan la manipulación del caché ARP mediante la suplantación de la dirección MAC del dispositivo que realiza el ataque, en donde la dirección MAC del dispositivo que actúa como punto de acceso a Internet tendrá la misma dirección que la máquina virtual de *Kali Linux*, la cual podrá interceptar los paquetes que fueron reenviados.

Se realizó la captura de tráfico de red del ataque de suplantación de ARP durante 60 minutos desde la máquina virtual de *Kali Linux* y se obtuvieron 117,342 paquetes en total, de los cuales se utilizarán 2014 que fueron seleccionados por contener la alerta de detección de dirección duplicada, utilizando el filtro que se muestra a continuación.

Figura 42

Filtro para el ataque suplantación de ARP

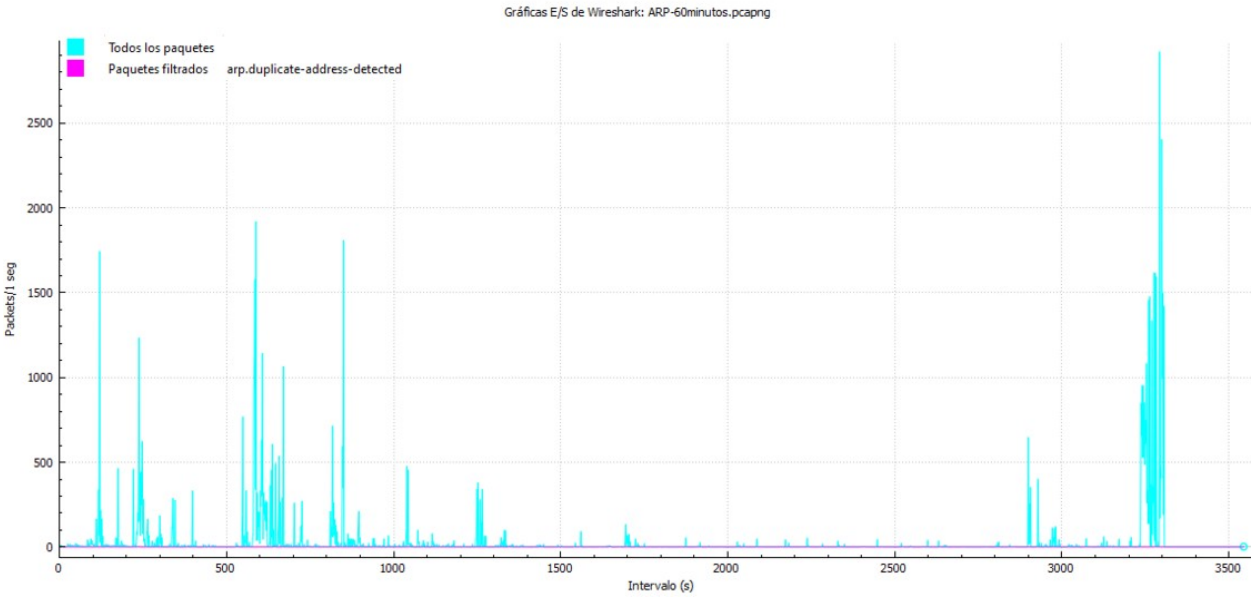
```
arp.duplicate-address-detected
```

Nota. Filtro utilizado en *Wireshark* para el ataque de suplantación de ARP. Elaboración propia en Carbon.now.sh.

En la figura que se muestra a continuación, se observa el gráfico E/S de la captura de suplantación de ARP de 60 minutos.

Figura 43

Gráfico E/S y leyenda de la captura de suplantación de ARP



Nota. El gráfico E/S muestra la cantidad de paquetes enviados y recibidos por segundo durante la captura del ataque de suplantación de ARP, realizado durante 60 minutos y la leyenda con el filtro correspondiente. Elaborado en el programa *Wireshark.*

En el gráfico de la Figura 43, se puede observar que los paquetes con la etiqueta de dirección duplicada se mantienen constantes durante toda la captura.

3.4. PROCESAMIENTO DE LOS DATOS

Para elaborar el *dataset* de las capturas de tráfico de red para el entrenamiento de las redes neuronales, se realizó el procesamiento de los datos en *Jupyter Notebook*, utilizando las librerías de *NumPy* y *Pandas* (ver Anexo D).

Se inició guardando todas las capturas por separado en archivos CSV (valores separados por comas) para facilitar el procesamiento, utilizando los filtros correspondientes detallados anteriormente.

En la tabla que se muestra a continuación, se realiza el conteo de los datos recolectados y filtrados para el entrenamiento.

Tabla 10

Resumen de la recolección de los datos

Tipo de tráfico de red	Duración	Paquetes en total	Paquetes para el entrenamiento
Normal	60 minutos	1,897,698	1,897,698
Escaneo de puertos	12 minutos	6,398	4,236
	140 minutos	87,042	71,670
Denegación de servicio	30 segundos	1,798,447	1,795,094
Suplantación de ARP	60 minutos	117,342	2014
Total	**4 horas, 32 minutos y 30 segundos**	**3,906,927**	**3,770,712**

Nota. En la tabla se muestra el conteo de los datos recolectados y filtrados para el entrenamiento. Elaboración propia.

Como se observa en la Tabla 10, de todos los datos recolectados un total de 3,770,712 serán procesados para el entrenamiento, en donde se comenzó por cargar cada *dataset* por separado, asignar los nombres de las columnas, eliminar las columnas de número de paquete e información adicional, eliminar los valores duplicados para optimizar el entrenamiento y asignar las etiquetas correspondientes para posteriormente unir los cinco *datasets* en uno solo y aplicar las funciones elaboradas de procesamiento de datos.

En la siguiente figura, se muestra el pseudocódigo de la función para agregar 0 si existe un valor nulo en las columnas con valores numéricos.

Figura 44

Función para agregar 0

```
function AddZero(row, column_name):
    if row(column_name) is null:
        return 0
    else
        return row(column_name)
```

Nota. En la figura se muestra el pseudocódigo de la función para agregar 0 si existe un valor nulo en una columna con valores numéricos. Elaboración propia en Carbon.now.sh.

En la figura que se muestra a continuación, se observa el pseudocódigo de la función para agregar -1 en las columnas con valores numéricos para representar los valores nulos que no pueden adoptar el valor de 0.

Figura 45

Función para agregar -1

```
function AddOne(row, column_name):
    if row(column_name) is null:
        return -1
    else
        return row(column_name)
```

Nota. En la figura se muestra el pseudocódigo de la función para agregar -1 si existe un valor nulo que no puede ser 0 en una columna con valores numéricos. Elaboración propia en Carbon.now.sh.

En la siguiente figura, se muestra el pseudocódigo para agregar "*none"* cuando exista un valor nulo en las columnas con valores de texto.

Figura 46

Función para agregar none

```
function AddNull(row, column_name):
    if row(column_name) is null:
        return 'none'
    else
        return row(column_name)
```

Nota. En la figura se muestra el pseudocódigo de la función para agregar *"none"* si existe un valor nulo en una columna con valores de texto. Elaboración propia en Carbon.now.sh.

Para las columnas de dirección de origen, destino y dispositivos con la misma dirección IP se realizaron procedimientos adicionales en donde se generaron nuevas columnas que adquieren valores booleanos según el contenido de la columna original, como se muestra a continuación.

Figura 47

Agregar una columna para los dispositivos con la misma dirección IP

```
function DuplicateRow(df):
    if "Duplicate IP add" not in df.columns:
        AddColumn(df, "Duplicate detected")
        return df

    AddRow(df, "Duplicate detected": "Duplicate detected")

    for each row in df:
        if row("Duplicate IP add") is null or empty:
            df.at["Duplicate detected"] = 0
        else:
            df.at["Duplicate detected"] = 1

    return df
```

Nota. En la figura se muestra el pseudocódigo de la función para agregar una nueva columna para los dispositivos con la misma dirección IP. Elaboración propia en Carbon.now.sh.

En el pseudocódigo de la Figura 47, se observa que la función crea una nueva columna para los dispositivos con la misma dirección IP y asigna el valor de 1 si en la columna original existe un valor, mientras que si es nulo asigna 0 y se elimina la columna original con el objetivo de optimizar el entrenamiento.

En la figura que se muestra a continuación, se observa el pseudocódigo de la función para agregar una nueva columna para la dirección IP de origen y asignar valores booleanos según corresponda.

Figura 48

Agregar una columna para la dirección de origen

```
function SameNetwork_src(df):
    if "Src add" not in df.columns:
        AddColumn(df, "SameNet src add")
        return df

    AddRow(df, "SameNet src add": "SameNet src add")

    for each row in df:
        src_add = string(row("Src add"))
        if src_add starts with ("192.168"):
            df.at["SameNet src add"] = 1
        else:
            df.at["SameNet src add"] = 0

    return df
```

Nota. En la figura se muestra el pseudocódigo de la función para agregar una columna para la dirección de origen y llenarla con valores booleanos. Elaboración propia en Carbon.now.sh.

Como se muestra en la Figura 48, la función crea una nueva columna para las direcciones de origen que provienen de la misma red, convirtiendo los valores de la columna original en texto y verificando si comienzan con “192.168” para asignar 1, cuando pertenecen a la misma red y 0, cuando la primera parte de la dirección IP no coincide, indicando que no proviene de la misma red.

Para la columna de dirección IP de destino se utilizó una nueva función para crear una columna y asignar los valores correspondientes luego de verificar si la dirección de destino pertenece a la misma red, como se muestra a continuación.

Figura 49

Agregar una columna para la dirección de destino

```
function SameNetwork_dst(df):
    if "Dst add" not in df.columns:
        AddColumn(df, "SameNet dst add")
        return df

    AddRow(df, "SameNet dst add": "SameNet dst add")

    for each row in df:
        dst_add = string(row("Dst add"))
        if dst_add starts with ("192.168"):
            df.at["SameNet dst add"] = 1
        else:
            df.at["SameNet dst add"] = 0

    return df
```

Nota. En la figura se muestra el pseudocódigo de la función para agregar una columna para la dirección de destino y llenarla con valores booleanos. Elaboración propia en Carbon.now.sh.

Finalmente, se aplicaron las todas las funciones, lo cual tuvo una duración de 5 minutos y 28 segundos y se verificó que no existan valores faltantes para posteriormente mezclar las filas del *dataset* de manera aleatoria para el entrenamiento de las redes neuronales.

3.5. ENTRENAMIENTO DE LAS REDES NEURONALES

El entrenamiento de los modelos finales con los datos recolectados y procesados se realizó en *Jupyter Notebook*, utilizando *TensorFlow, Keras* y *Scikit-Learn.* Se inició importando las librerías necesarias, cargando el *dataset* y asignando los nombres de las columnas, eliminando los valores nulos y duplicados y definiendo los tipos de datos de las filas para evitar errores durante el entrenamiento, como se muestra en la siguiente tabla.

Tabla 11

Tipos de datos de las columnas del dataset

Nombre	Tipo de dato
Delta time	float32
Protocol	object
Length	int32
Src port	int32
Dst port	int32
Flags	object
Severity	object
Window size	int32
Label	object
Duplicate detected	int32
SameNet src add	int32
SameNet dst add	int32

Nota. En la tabla se muestran los tipos de datos de todas las columnas del *dataset.* Elaboración propia.

Dado que se eliminaron los valores duplicados durante el procesamiento de los datos y se removieron las filas con valores nulos antes de iniciar el entrenamiento, se redujo el tamaño del *dataset*, obteniendo un total de 1,686,859 filas, como se observa en la siguiente tabla del conteo de las clases en la columna *Label.*

Tabla 12

Conteo de las clases de la columna Label

Clase	Número de ejemplares
ARP spoofing	2012
DoS	1,003,235
Normal	606,329
Port scan	75,283
Total	**1,686,859**

Nota. En la tabla se muestra el conteo de las clases de la columna *Label*. Elaboración propia.

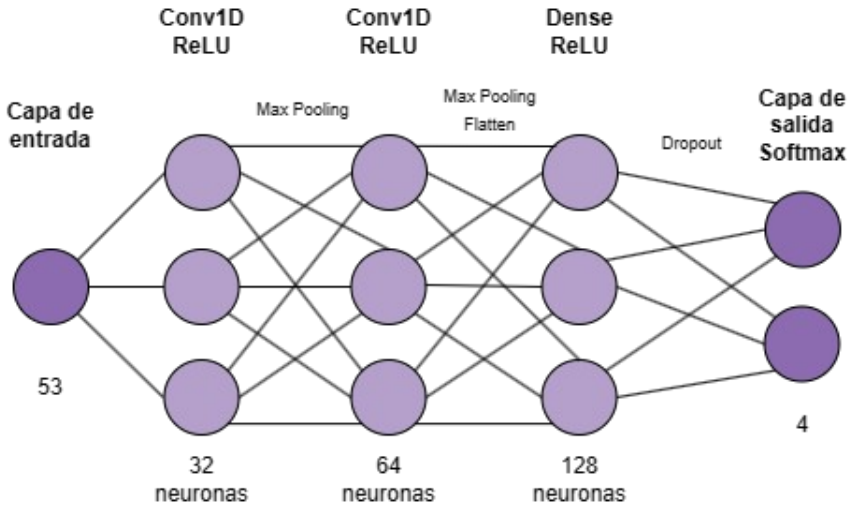
Como se observa en la Tabla 12, el *dataset* elaborado para el entrenamiento presenta una distribución no balanceada de las clases, de manera similar que en el *dataset* utilizado durante la evaluación preliminar.

Durante la etapa de ingeniería de características, solamente se codificaron tres columnas con valores de texto, exceptuando la columna con etiquetas, utilizando la función de la Figura 15, con el propósito de que sea posible realizar predicciones con los modelos entrenados posteriormente. Mediante la codificación por variables ficticias y *One Hot Encoding*, se generaron un total de 53 columnas de características, permitiendo que se realice el modelado de las redes neuronales y la definición de las variables para los tres modelos.

En la figura que se muestra a continuación, se observa el diagrama del modelado de la red neuronal convolucional.

Figura 50

Diagrama de red neuronal convolucional

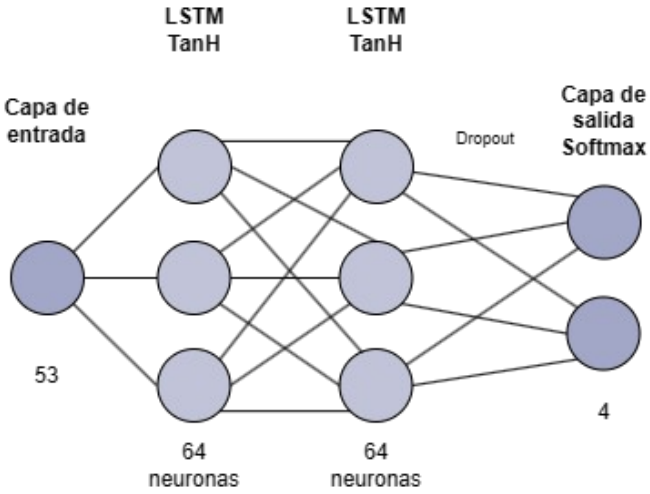


Nota. En la figura se muestra el diagrama de la red neuronal convolucional. Elaboración propia utilizando la herramienta en línea draw.io.

En la siguiente figura, se muestra el diagrama del modelado de la red neuronal recurrente con arquitectura LSTM.

Figura 51

Diagrama de red neuronal recurrente

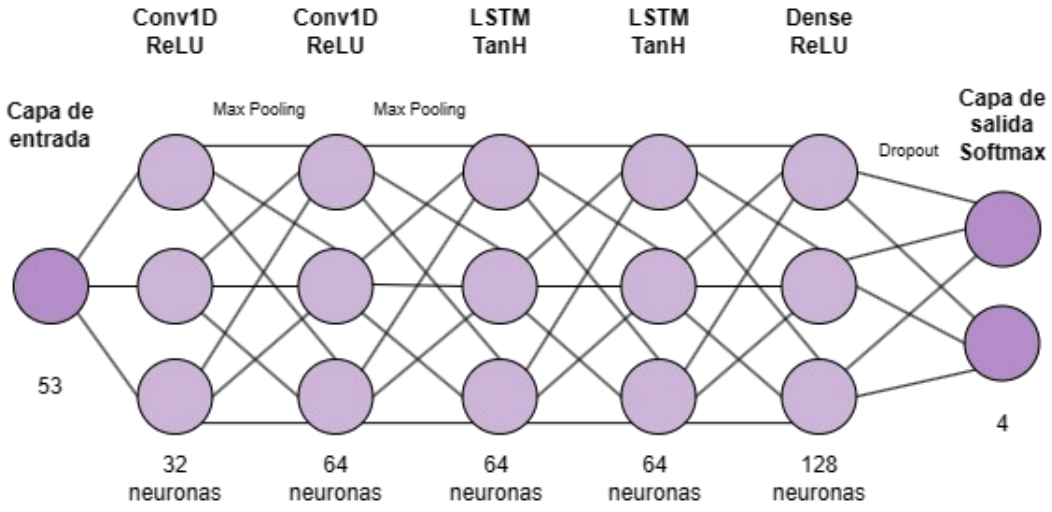


Nota. Diagrama de la red neuronal recurrente. Elaboración propia utilizando la herramienta en línea draw.io.

A continuación, se muestra el modelo de la red neuronal híbrida convolucional recurrente.

Figura 52

Diagrama de red neuronal CNN-RNN



Nota. Diagrama de la red neuronal híbrida convolucional recurrente. Elaboración propia utilizando draw.io.

Para la evaluación final de los modelos de redes neuronales entrenados, se calculó el tiempo de entrenamiento y se utilizaron las funciones detalladas anteriormente para calcular los parámetros de precisión, exactitud y exhaustividad de las pruebas, como se observa en las Figuras 19, 20 y 21, respectivamente.

En la figura que se muestra a continuación, se observa el pseudocódigo de la inicialización, compilación y entrenamiento de la red neuronal convolucional, utilizando validación cruzada estratificada con 30 pliegues (ver Anexo E).

Figura 53

Modelo de red neuronal convolucional

```
sss = StratifiedShuffleSplit(n_splits = 30, test_size = 0.25)

for each fold (train, test) in enumerate(sss.split(x, y)):
    x_train = Reshape(x_train)
    x_test = Reshape(x_test)

    model = Sequential()
    model.add(Conv1D(32, 3, activation='relu', input_shape=(x_train.shape)))
    model.add(MaxPooling1D(2))
    model.add(Conv1D(64, 3, activation='relu'))
    model.add(MaxPooling1D(2))
    model.add(Flatten)
    model.add(Dense(128, activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(y.shape, activation='softmax'))

    model.compile(loss='categorical_crossentropy',
                  optimizer = Adam(learning_rate=0.001),
                  metrics=(calc_precision, calc_accuracy, calc_recall))

    CNN = model.fit(training_data = (x_train, y_train),
                    validation_data=(x_test, y_test),
                    batch_size=1024,
                    epochs=5,
                    callbacks=EarlyStopping)
```

Nota. En la figura se muestra el pseudocódigo de la inicialización, compilación y entrenamiento de la red neuronal convolucional utilizando validación cruzada estratificada. Elaboración propia con la herramienta en línea Carbon.now.sh.

A continuación, se muestra la construcción y entrenamiento del modelo de red neuronal recurrente.

Figura 54

Modelo de red neuronal recurrente

```
sss = StratifiedShuffleSplit(n_splits = 30, test_size = 0.25)

for each fold (train, test) in enumerate(sss.split(x, y)):
    x_train = Reshape(x_train)
    x_test = Reshape(x_test)

    model = Sequential()
    model.add(LSTM(64, activation='tanh', input_shape=(x_train.shape)))
    model.add(LSTM(64, activation='tanh'))
    model.add(Dropout(0.5))
    model.add(Dense(y.shape, activation='softmax'))

    model.compile(loss='categorical_crossentropy',
                  optimizer = Adam(learning_rate=0.001),
                  metrics=(calc_precision, calc_accuracy, calc_recall))

    RNN = model.fit(training_data = (x_train, y_train),
                    validation_data=(x_test, y_test),
                    batch_size=1024,
                    epochs=5,
                    callbacks=EarlyStopping)
```

Nota. En la figura se muestra el pseudocódigo de la inicialización, compilación y entrenamiento de la red neuronal recurrente utilizando validación cruzada estratificada. Elaboración propia en Carbon.now.sh.

Como se muestra en la Figura 54, en la compilación del modelo recurrente, de la misma forma que en el modelo convolucional, se utilizó el algoritmo de optimización *Adam* y una tasa de aprendizaje del 0.001, durante 5 épocas por cada uno de los 30 pliegues y un tamaño de lote de 1024 (ver Anexo F).

En la siguiente figura, se muestra el pseudocódigo de la construcción y entrenamiento de la red neuronal híbrida convolucional recurrente, que presenta características similares en la compilación y entrenamiento que los modelos anteriores (ver Anexo G).

Figura 55

Modelo de red neuronal híbrida CNN-RNN

```
sss = StratifiedShuffleSplit(n_splits = 30, test_size = 0.25)

for each fold (train, test) in enumerate(sss.split(x, y)):
    x_train = Reshape(x_train)
    x_test = Reshape(x_test)

    model = Sequential()
    model.add(Conv1D(32, 3, activation='relu', input_shape=(x_train.shape)))
    model.add(MaxPooling1D(2))
    model.add(Conv1D(64, 3, activation='relu'))
    model.add(MaxPooling1D(2))
    model.add(LSTM(64, return_sequences=True, activation='tanh'))
    model.add(LSTM(64, activation='tanh'))
    model.add(Dense(128, activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(y.shape, activation='softmax'))

    model.compile(loss='categorical_crossentropy',
                  optimizer = Adam(learning_rate=0.001),
                  metrics=(calc_precision, calc_accuracy, calc_recall))

    CNN_RNN = model.fit(training_data = (x_train, y_train),
                    validation_data=(x_test, y_test),
                    batch_size=1024,
                    epochs=5,
                    callbacks=EarlyStopping)
```

Nota. En la figura se muestra el pseudocódigo de la inicialización, compilación y entrenamiento de la red neuronal híbrida CNN-RNN utilizando validación cruzada estratificada. Elaboración propia en Carbon.now.sh.

La validación cruzada estratificada permitió que se calculen 30 valores de los parámetros utilizando el conjunto de prueba durante el entrenamiento de los tres modelos, que se utilizarán para las pruebas de hipótesis y evaluación de los modelos. Se utilizó una llamada de retorno o *callback* de detención temprana que monitorea el parámetro de precisión para detener el entrenamiento si deja de mejorar en dos épocas, previniendo el sobre ajuste de los modelos.

En la tabla que se muestra a continuación, se observa el tiempo de entrenamiento y de las predicciones de los tres modelos entrenados.

Tabla 13

Tiempo de entrenamiento y de las predicciones

Modelo	Tiempo de entrenamiento	Tiempo de las predicciones
CNN	56 minutos y 39 segundos	21 segundos
RNN	37 minutos y 21 segundos	19 segundos
CNN-RNN	2 horas, 39 minutos y 30 segundos	46 segundos
Total	**4 horas, 13 minutos y 30 segundos**	**1 minuto y 26 segundos**

Nota. En la tabla se muestra el tiempo de entrenamiento y de las predicciones de los tres modelos. Elaboración propia.

Como se observa en la Tabla 13, el modelo de red neuronal recurrente tomó menos tiempo en comparación con los otros dos para completar el entrenamiento y es capaz de realizar predicciones en 19 segundos, mientras que el modelo híbrido CNN-RNN tuvo un tiempo mayor de entrenamiento y predicciones, debido a que tiene una estructura más compleja formada por más capas ocultas que los modelos por separado.

A continuación, se muestran los resultados de precisión, exactitud y exhaustividad de las pruebas de los tres modelos.

Tabla 14

Resultados de los parámetros de evaluación de las predicciones

Modelo	Precisión	Exactitud	Exhaustividad
CNN	0.9910	0.9954	0.9905
RNN	0.9914	0.9957	0.9914
CNN-RNN	0.9939	0.9970	0.9939

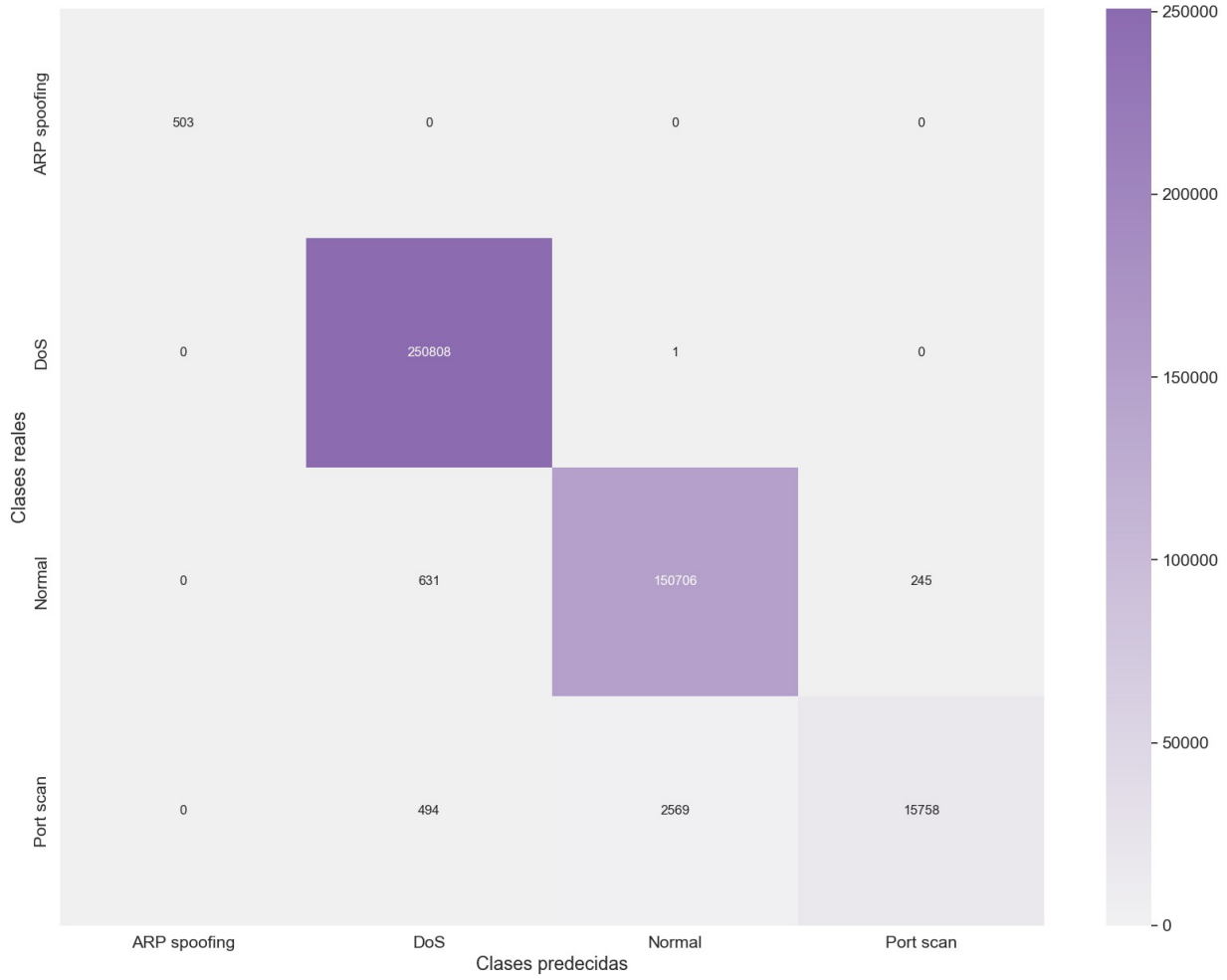
Nota. En la tabla se muestran los resultados de las predicciones. Elaboración propia.

En la Tabla 14, se observa que los tres modelos obtuvieron una precisión, exactitud y exhaustividad del 99% en las predicciones con el conjunto de prueba, lo cual indica un bajo índice de falsos positivos y falsos negativos en contraste con los verdaderos positivos y

verdaderos negativos. Para analizar las predicciones de forma visual, se elaboraron matrices de confusión utilizando la librería de *Scikit-Learn,* como se muestra en la siguiente figura, comenzando con el modelo de red neuronal convolucional.

Figura 56

Matriz de confusión de la red CNN



Nota. En la figura se muestra la matriz de confusión de la red CNN, generada con *Scikit-Learn*. Elaboración propia en *Jupyter Notebook.*

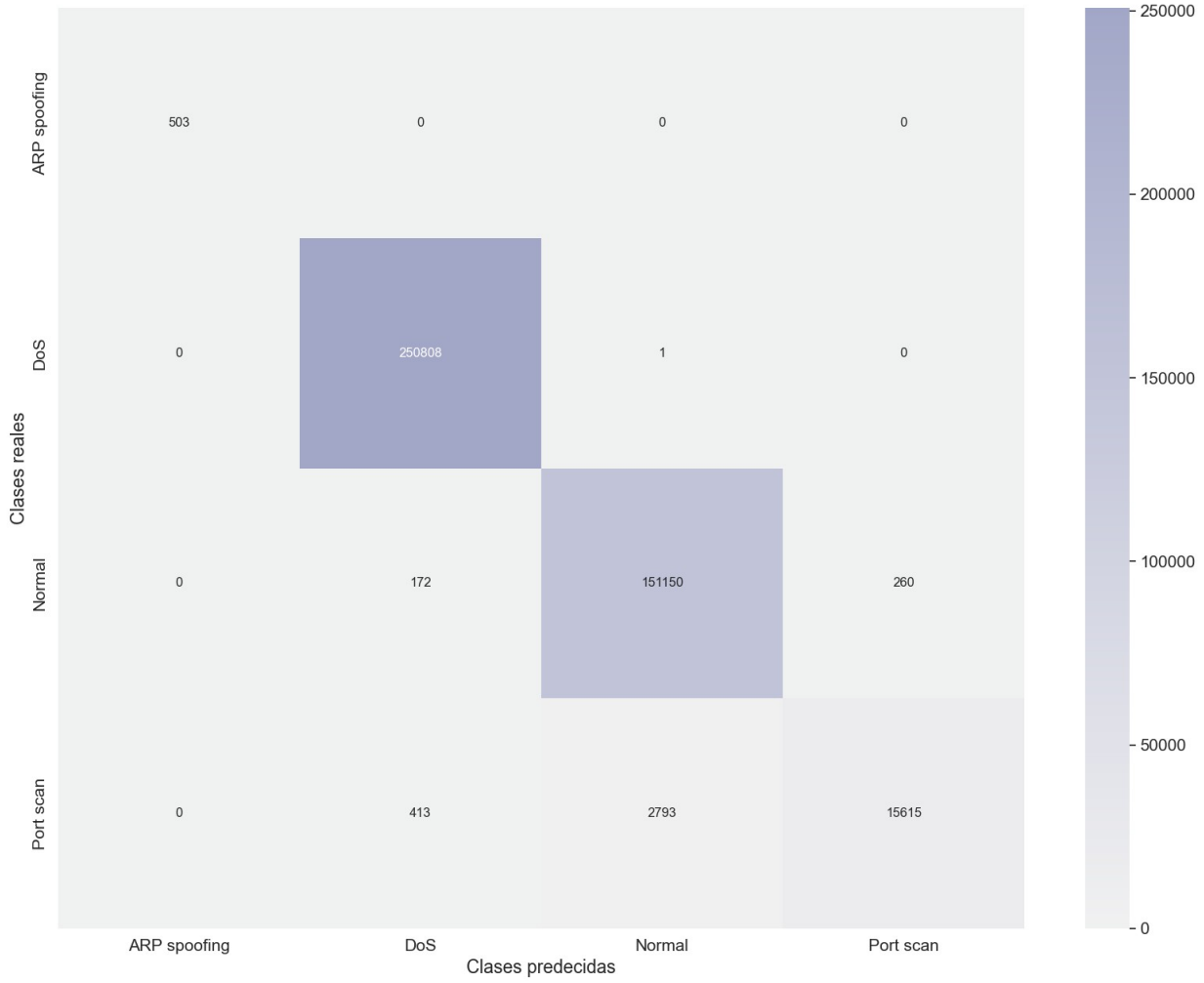
En la matriz de confusión de la Figura 56, se observa un alto índice de verdaderos positivos y verdaderos negativos, representados por los colores más oscuros en las clases con una mayor cantidad de ejemplares dentro del conjunto de prueba, como la clase de tráfico normal y de los

ataques de denegación de servicio, en contraste con la etiqueta de *ARP spoofing*, que no muestra un color oscuro a pesar de no tener predicciones incorrectas, debido a la distribución no balanceada del *dataset*.

En la figura que se muestra a continuación, se observa la matriz de confusión para el modelo de red neuronal recurrente con arquitectura LSTM.

Figura 57

Matriz de confusión de la red RNN



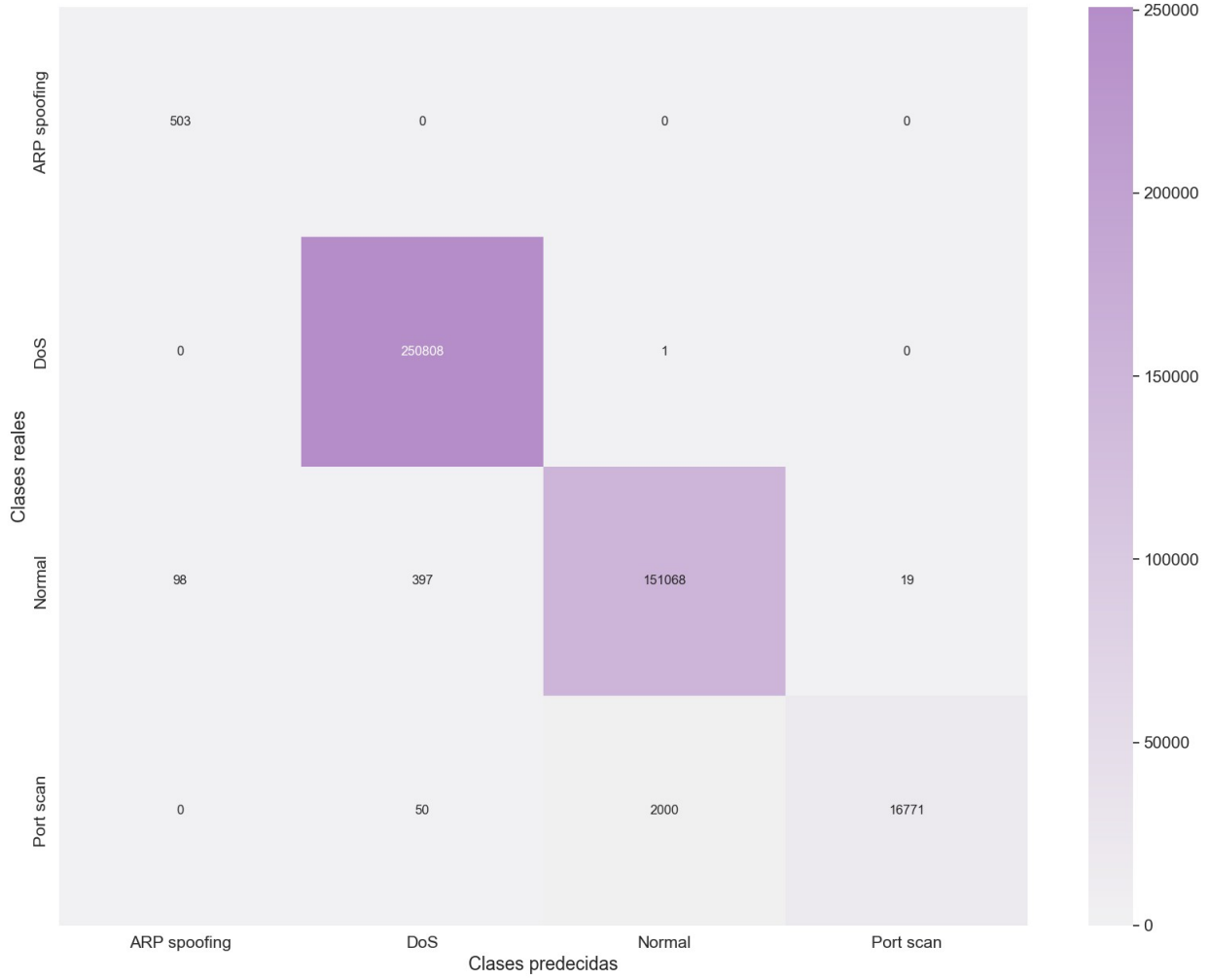
Nota. En la figura se muestra la matriz de confusión de la red RNN, generada con *Scikit-Learn*. Elaboración propia en *Jupyter Notebook*.

Como se muestra en la matriz de confusión de la Figura 57, la red neuronal recurrente tuvo una baja cantidad de falsos negativos y falsos positivos, mostrando un alto rendimiento de manera similar que la red convolucional.

En la siguiente figura, se muestra la matriz de confusión del modelo híbrido convolucional recurrente.

Figura 58

Matriz de confusión de la red CNN-RNN



Nota. En la figura se muestra la matriz de confusión de la red híbrida CNN-RNN, generada con *Scikit-Learn*. Elaboración propia en *Jupyter Notebook.*

En la matriz de confusión del modelo híbrido CNN-RNN de la Figura 58, se observa que a diferencia de los modelos anteriores, no todas las predicciones para la clase *ARP spoofing* fueron aciertos, mientras que la cantidad de verdaderos positivos y verdaderos negativos se mantuvo elevada para el resto de las clases.

3.6. PRUEBAS DE LOS MODELOS

Una vez que concluyó el entrenamiento de los tres modelos de redes neuronales, se realizaron pruebas con nuevas capturas de tráfico de red normal y de los ataques, con una duración de 30 segundos cada una, para obtener un tiempo aproximado de detección de intrusos en una red, utilizando las funciones de procesamiento de datos e ingeniería de características para poder obtener predicciones y mostrar los resultados en *Spyder*, utilizando las librerías de *Tkinter, CustomTkinter* y *Plotly.*

Dado que cada clase representa un porcentaje diferente dentro de una captura de tráfico de red completa, en donde se utilizan todos los paquetes capturados para el entrenamiento sin utilizar filtros, se determinaron condiciones para la detección de los ataques, como se muestra en el siguiente pseudocódigo.

Figura 59

Condiciones para la detección de los ataques

```
if Percentage(ARP Spoofing) > 5 and Percentage(Normal) < 85:
    resultado = "Ataque de ARP Spoofing"

else if Percentage(DoS) > 90:
    resultado = "Ataque de DoS"

else if Percentage(Port scan) > 10 and Percentage(Normal) < 85:
    resultado = "Escaneo de puertos"

else:
    resultado = "Tráfico de red normal"
```

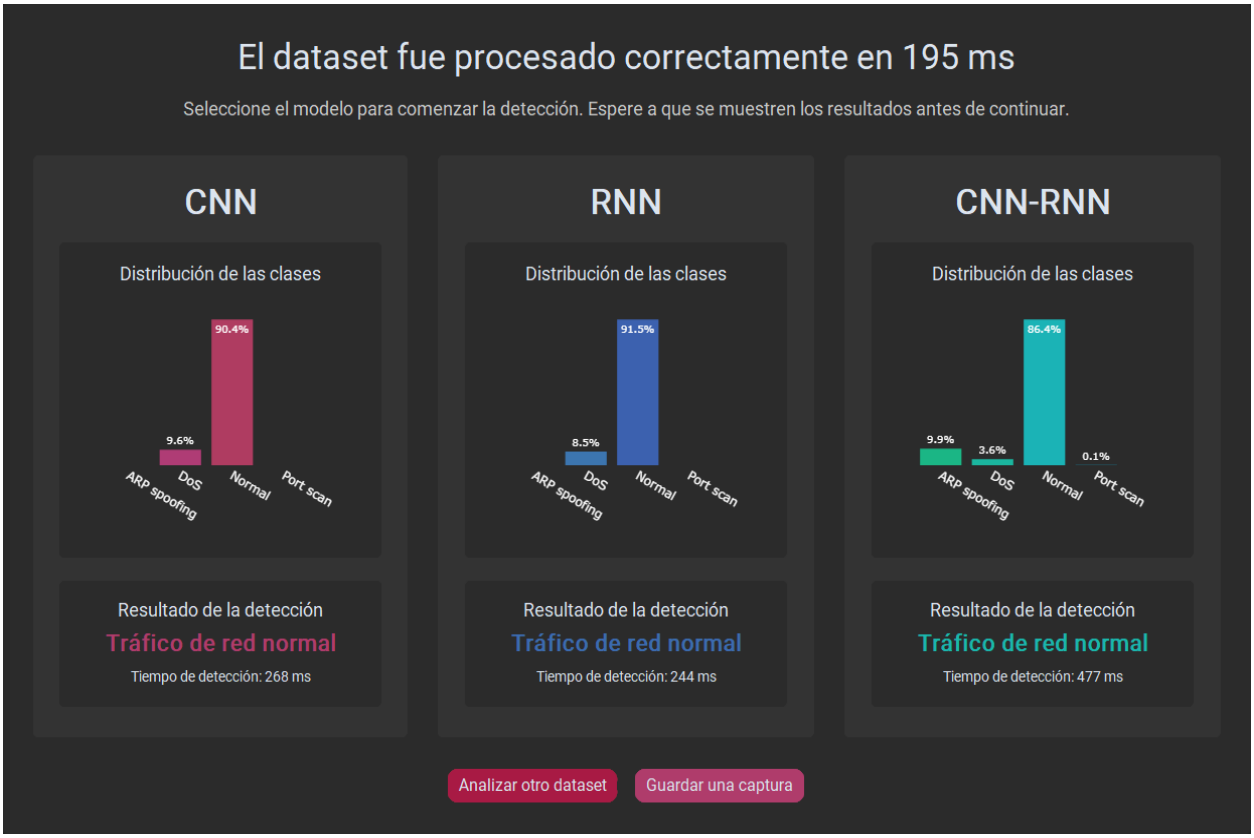
Nota. Pseudocódigo de las condiciones para la detección de los ataques. Elaboración propia en Carbon.now.sh.

Para mostrar los resultados de las predicciones de los modelos de manera visual, se elaboró un programa en el que se inicia por cargar los modelos entrenados y procesar los *datasets* de capturas de tráfico de red, de forma que se puedan analizar diferentes archivos con extensión CSV y guardar los resultados de las predicciones en imágenes, al igual que el tiempo exacto que tuvo el procedimiento.

En la figura que se muestra a continuación, se observan los resultados obtenidos de las predicciones de los tres modelos utilizando la captura de tráfico de red normal de 30 segundos, la distribución de las clases representadas en los gráficos de barras, el tiempo de procesamiento y el tiempo de detección en milisegundos para cada modelo.

Figura 60

Predicciones para el tráfico de red normal



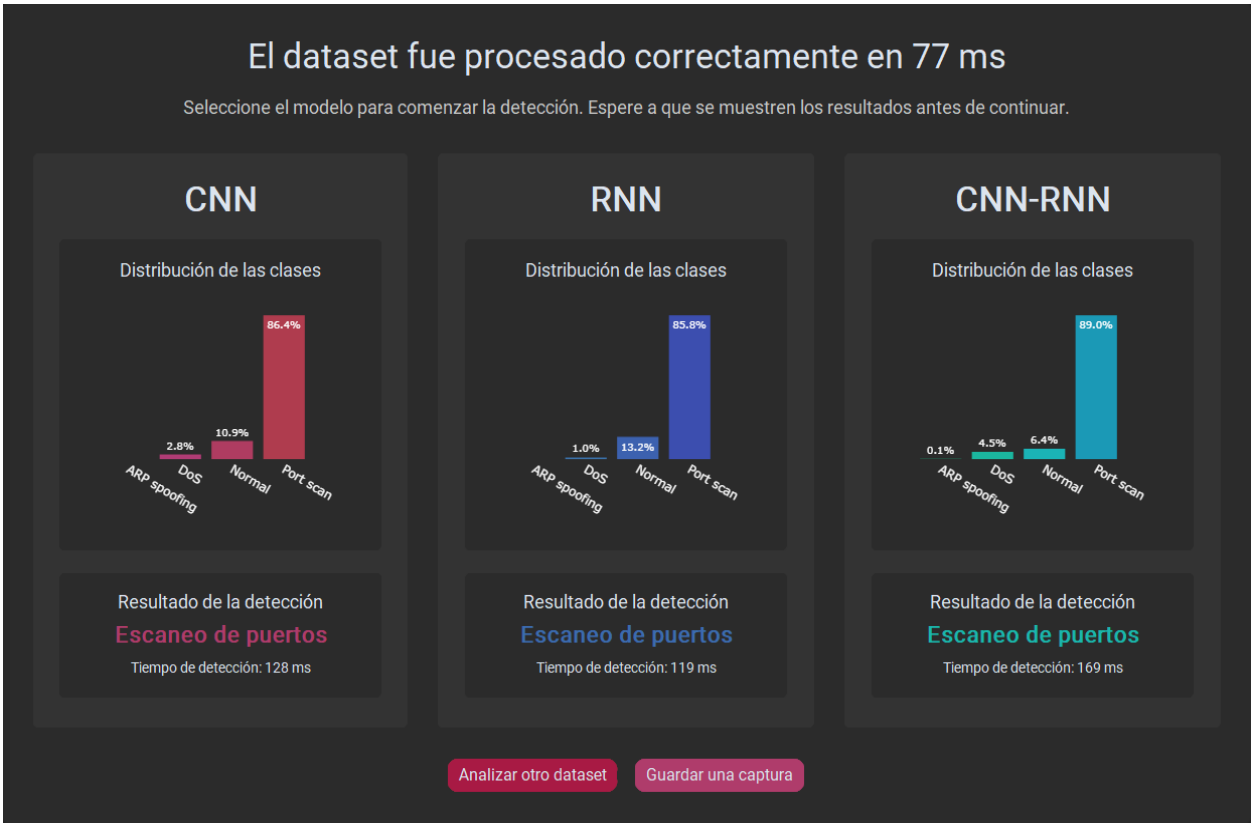
Nota. En la figura se muestra la captura de las predicciones para el tráfico de red normal. Elaboración propia en *Spyder.*

Como se muestra en la Figura 60, a pesar de las diferencias identificadas en la distribución de las clases, los tres modelos acertaron en la detección de tráfico de red normal.

En la siguiente figura, se muestra una captura de los resultados de las predicciones para el tráfico de red producido por escaneo de puertos.

Figura 61

Predicciones para el escaneo de puertos



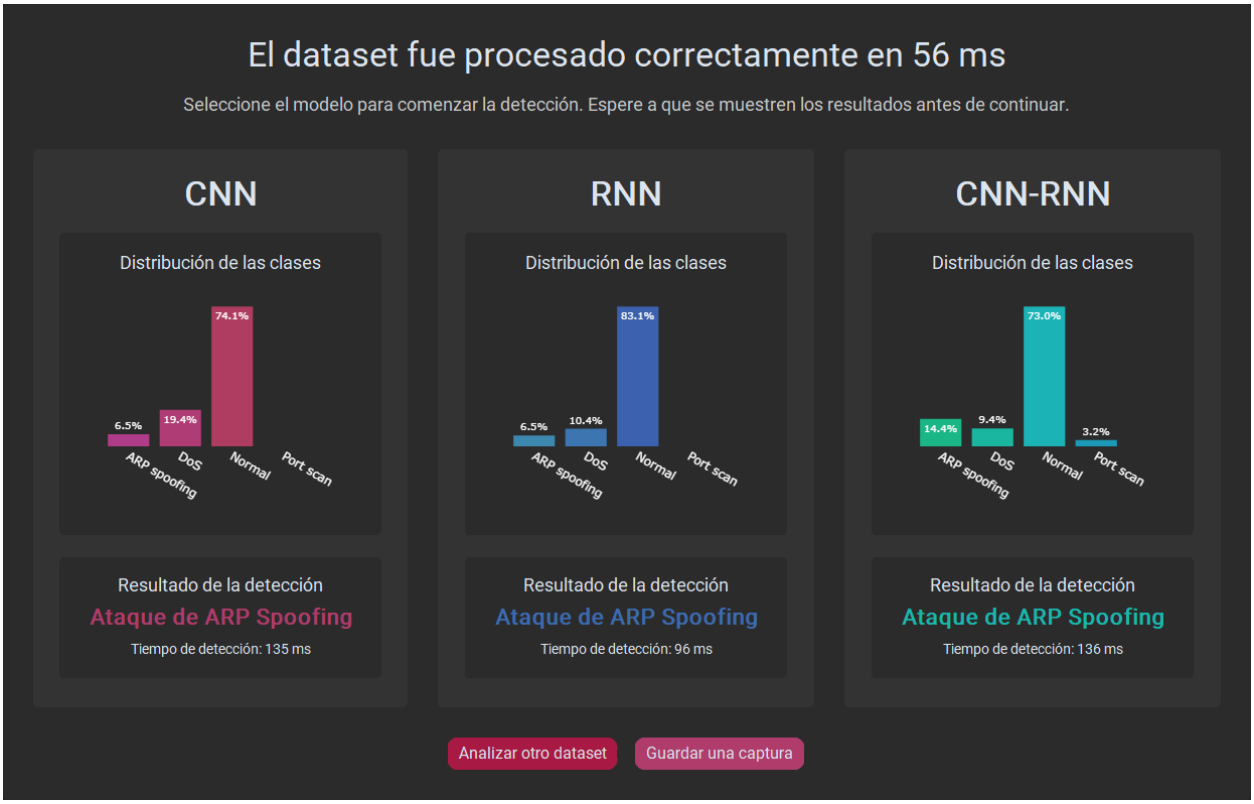
Nota. En la figura se muestra la captura de las predicciones para el escaneo de puertos. Elaboración propia en *Spyder.*

En la captura de la Figura 61, se observa que la distribución de las clases con el *dataset* de escaneo de puertos presenta una gran variedad de clases en los tres modelos, dado que no se utilizaron filtros en las capturas utilizadas para las pruebas, lo cual no afectó el resultado final de la detección, que fue acertada para las tres redes neuronales.

A continuación, se muestran los resultados de las predicciones utilizando el *dataset* del ataque de suplantación de ARP.

Figura 62

Predicciones para el ataque de suplantación de ARP



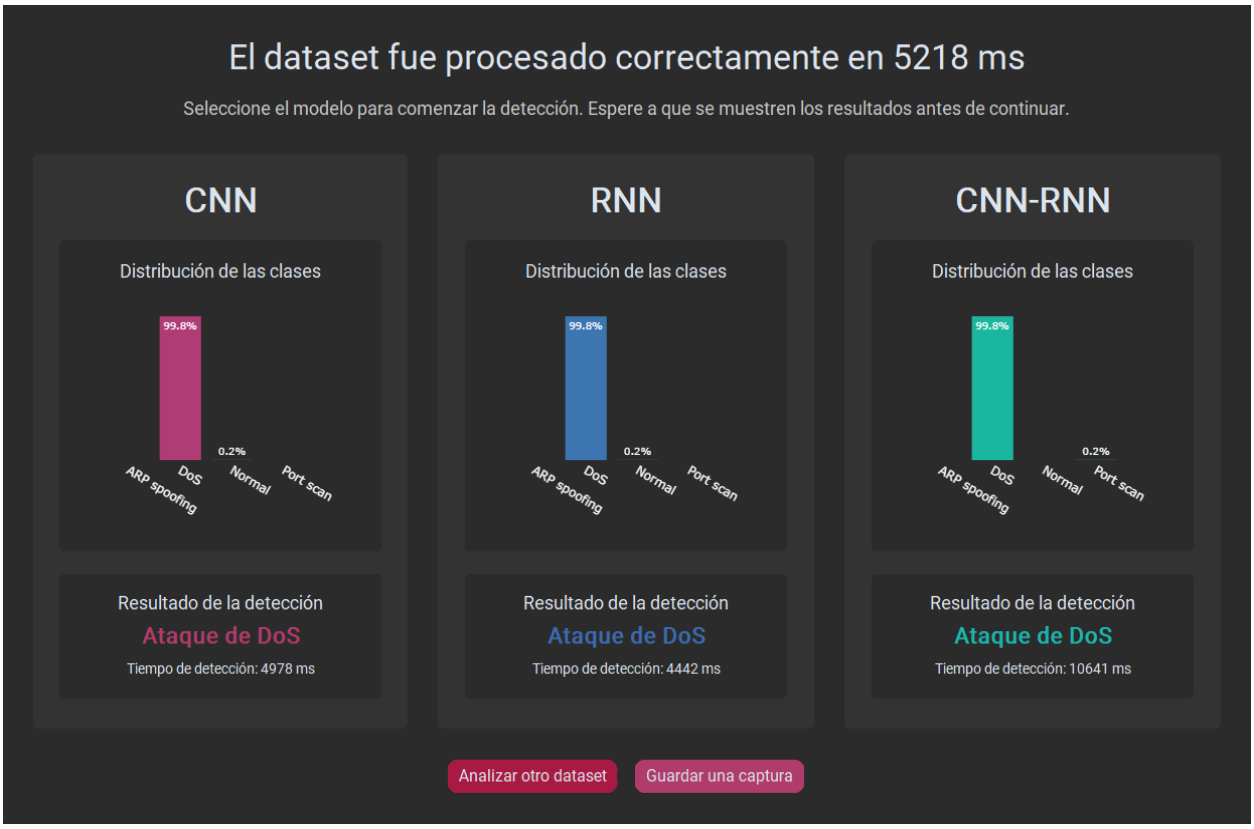
Nota. En la figura se muestra la captura de las predicciones para el ataque de suplantación de ARP. Elaboración propia en *Spyder.*

En la captura de la Figura 62, es importante notar que a pesar de que en los tres modelos existe una mayor cantidad de paquetes identificados como tráfico normal, el porcentaje de paquetes reconocidos un ataque de suplantación de ARP hacen que los resultados obtenidos sean acertados, ya que se aplican las condiciones para la detección como se muestra en la Figura 59.

En la siguiente figura, se muestran los resultados de las predicciones obtenidas con la captura de tráfico de red de un ataque de denegación de servicio de 30 segundos.

Figura 63

Predicciones para el ataque de DoS



Nota. En la figura se muestra la captura de las predicciones para el ataque de denegación de servicio. Elaboración propia en *Spyder.*

En la Figura 63, se observa que los tres modelos identificaron al tráfico de red analizado como un ataque de denegación de servicio, con una distribución de la clase del 99% con las tres redes neuronales.

Con el objetivo de mejorar el rendimiento del programa y evitar posibles errores, luego de la selección del *dataset* y antes de pasar al procesamiento de los datos, se definió que si el archivo tiene más de 100,000 filas, se mantendrán de manera aleatoria sin pasar esa cantidad.

En la tabla que se muestra a continuación, se observa el número de paquetes de las capturas de tráfico de red de 30 segundos y el tiempo de procesamiento de los datos para cada clase.

Tabla 15

Tiempo del procesamiento de datos en el programa

Clase	Número de paquetes	Paquetes procesados	Tiempo de procesamiento
Normal	3692	3692	195 ms
Port Scan	718	718	77 ms
ARP spoofing	278	278	56 ms
DoS	1,433,880	100,000	5218 ms
Total	**1,438,568**	**104,69**	**5546 ms**

Nota. En la tabla se muestra el tiempo del procesamiento de datos para cada clase de tráfico de red en el programa. Elaboración propia.

En la tabla que se muestra a continuación, se observa el tiempo de las predicciones en milisegundos de cada modelo para las diferentes clases de tráfico de red en el programa elaborado y el tiempo total.

Tabla 16

Tiempo de predicción de los modelos en el programa

Clase	CNN	RNN	CNN-RNN
Normal	268 ms	244 ms	477 ms
Port Scan	128 ms	119 ms	169 ms
ARP spoofing	135 ms	96 ms	136 ms
DoS	4978 ms	4442 ms	10641 ms
Total	**5509 ms**	**4901 ms**	**11423 ms**

Nota. En la tabla se muestra el tiempo de predicción de los modelos para cada una de las clases en el programa. Elaboración propia.

Como se observa en la Tabla 16, la cantidad de paquetes y la complejidad de la red neuronal tienen un efecto en el tiempo de predicción de los modelos, al igual que la ejecución simultánea de otros programas, como ocurrió durante las capturas de tráfico de red y la obtención de los resultados de las predicciones.

3.7. DEMOSTRACIÓN DE LA HIPÓTESIS

Para la demostración de la hipótesis, se utilizó el método de análisis de varianza factorial de una vía ANOVA y se realizaron pruebas de *t* de *Student*, con un nivel de significancia del 0.05, que representa un riesgo del 5% de rechazar la hipótesis nula. Las pruebas de hipótesis se realizaron en el lenguaje de programación *Python* en *Jupyter Notebook*, utilizando las librerías de *NumPy, Pandas* y *SciPy* (ver Anexo H).

A continuación, se presentan las hipótesis de investigación y nula, formuladas anteriormente, para realizar un análisis más profundo de lo que implica aceptar o rechazar cada una.

H_1: Las redes neuronales convolucionales, recurrentes e híbridas convolucionales recurrentes generan diferentes valores de precisión, mayores al 95%, permitiendo la detección de intrusos en redes.

H_0: Las redes neuronales convolucionales, recurrentes e híbridas convolucionales recurrentes no generan diferentes valores de precisión y son menores al 95%, dificultando la detección de intrusos en redes.

Para rechazar la primera parte de la hipótesis nula y demostrar que existe una diferencia significativa entre la precisión de los tres modelos mediante el método ANOVA de una vía, se utilizaron las 30 muestras de la precisión de las pruebas de cada uno de los modelos, obtenidas a partir del uso de la validación cruzada estratificada durante el entrenamiento.

A continuación, se realiza el planteamiento para aceptar o rechazar la primera parte de la hipótesis de investigación, con un nivel de significancia del 0.05.

- Se acepta la hipótesis de investigación H_1 si *p-value* es menor que 0.05, indicando que existe una diferencia significativa entre la precisión de los modelos de redes neuronales.
- Se acepta la hipótesis nula H_0 si *p-value* es mayor que 0.05, indicando que no existe una diferencia significativa entre la precisión de los modelos de redes neuronales.

En la siguiente tabla, se observan las 30 muestras de precisión de las pruebas de cada modelo elaborado y el promedio de cada uno.

Tabla 17

Muestras de la precisión de cada modelo

CNN			RNN			CNN-RNN		
0.99104953	0.99176919	0.99307603	0.94147384	0.99333078	0.95628524	0.97370392	0.99436128	0.97041428
0.9929266	0.99002129	0.99079078	0.94798744	0.9927749	0.99355155	0.98795015	0.97518027	0.97267157
0.98955303	0.99140906	0.99260837	0.94349581	0.94129711	0.99176008	0.95216054	0.99405199	0.98668921
0.99344671	0.99359053	0.98399633	0.99230307	0.99189997	0.9924925	0.97408259	0.96941394	0.98547077
0.99246198	0.99137461	0.99018413	0.94275057	0.9916842	0.99246746	0.97178018	0.98226589	0.9665252
0.99013996	0.98984265	0.98976415	0.98961163	0.99154907	0.94369006	0.98898888	0.97154564	0.98325884
0.99062663	0.98959106	0.99103421	0.9922387	0.99178135	0.99155313	0.99490666	0.96390867	0.97969431
0.99289703	0.99095827	0.99241292	0.99103051	0.94804782	0.99169844	0.97148645	0.96498471	0.9891969
0.98935926	0.99201572	0.99220216	0.99187845	0.99118626	0.99128795	0.98998421	0.95923293	0.98646247
0.98856288	0.99247473	0.99104786	0.99024254	0.94069409	0.99137121	0.99455816	0.98973507	0.99394143
Promedio: 0.99103959			**Promedio: 0.97778052**			**Promedio: 0.9792869**		

Nota. En la tabla se muestran los resultados de precisión de las pruebas de cada modelo. Elaboración propia.

En la figura que se muestra a continuación, se observa el pseudocódigo del análisis de varianza factorial de una vía.

Figura 64

Análisis de varianza factorial de una vía

```
alpha = 0.05

f-statistic, p-value = OneWay_ANOVA(CNN, RNN, CNN_RNN)

if p-value < alpha
    resultado = "Se rechaza la hipótesis nula"
else
    resultado = "Se acepta la hipótesis nula"
```

Nota. En la figura se muestra el pseudocódigo del método ANOVA de una vía. Elaboración propia en Carbon.now.sh.

El resultado del estadístico *F* fue de 7.6079 y a partir de ese valor se obtuvo un *p-value* de 0.0009, lo cual indica que existe una diferencia significativa entre la precisión de los tres

modelos, permitiendo que se acepte la primera parte de la hipótesis de investigación. Para rechazar la segunda parte de la hipótesis nula, demostrando que la precisión de los tres modelos es mayor al 95%, se realizaron pruebas individuales de *t* de *Student*, utilizando los valores de precisión de la Tabla 17.

A continuación, se realiza el planteamiento para aceptar la segunda parte de la hipótesis de investigación, con un nivel de significancia del 0.05 y un valor objetivo de precisión del 95%.

- Se acepta la hipótesis de investigación H_1 si *p-value* es menor que 0.05, indicando que la precisión media de los modelos de redes neuronales es significativamente superior al 95%.
- Se acepta la hipótesis nula H_0 si *p-value* es mayor que 0.05, indicando que la precisión media de los modelos de redes neuronales no es significativamente superior al 95%.

En la siguiente figura, se muestra el pseudocódigo de las pruebas de *t* de *Student* para cada uno de los modelos.

Figura 65

Pruebas de t de Student

```
alpha = 0.05
valor_objetivo = 0.95

for each in (CNN, RNN, CNN_RNN)

    t-statistic, p-value = T_test(CNN, RNN, CNN_RNN, valor_objetivo)

    if p-value < alpha
        resultado = "Se rechaza la hipótesis nula"
    else
        resultado = "Se acepta la hipótesis nula"
```

Nota. En la figura se muestra el pseudocódigo de las pruebas de *t* de *Student* para cada uno de los modelos. Elaboración propia en Carbon.now.sh.

En la siguiente tabla, se muestran los resultados del estadístico *t, p-value* y si se rechaza o no la hipótesis nula para cada modelo de red neuronal.

Tabla 18

Resultados de las pruebas de t de Student

Modelo	Estadístico *t*	*p-value*	Se rechaza H_0
CNN	118.8561	1.5263463305159994 e^{-40}	Sí
RNN	6.9342	1.2743148229190142 e^{-7}	Sí
CNN-RNN	13.6387	3.8045040322259805 e^{-14}	Sí

Nota. En la tabla se muestran los resultados de las pruebas de *t* de *Student* para cada modelo. Elaboración propia.

Con base en las pruebas de análisis de varianza factorial de una vía y de *t* de *Student*, se rechaza la hipótesis nula y se acepta la hipótesis de investigación con un 95% de confianza, comprobando que las redes neuronales convolucionales, recurrentes e híbridas convolucionales recurrentes generan diferentes valores de precisión, mayores al 95%, permitiendo la detección de intrusos en redes.

3.8. EVALUACIÓN TÉCNICA

Para la primera parte de la evaluación técnica de los modelos de redes neuronales, se tomaron en cuenta los resultados de los promedios de los parámetros de rendimiento de exactitud, precisión y exhaustividad, obtenidos durante el entrenamiento mediante la validación cruzada estratificada, como se muestra en la siguiente tabla

Tabla 19

Evaluación técnica de los parámetros de rendimiento

Modelo	Precisión	Exactitud	Exhaustividad
CNN	0.9910	0.9954	0.9904
RNN	0.9778	0.9889	0.9777
CNN-RNN	0.9793	0.9889	0.9764

Nota. En la tabla se muestra la evaluación técnica de los parámetros de rendimiento de las pruebas los tres modelos entrenados. Elaboración propia.

Como se muestra en la Tabla 19, los modelos de redes neuronales entrenados presentan un alto rendimiento, con una precisión, exactitud y exhaustividad del 99% para la red neuronal convolucional y resultados mayores al 97% para el modelo recurrente e híbrido convolucional recurrente.

Para la segunda parte de la evaluación técnica, se consideró el tiempo de entrenamiento de cada modelo y el tiempo de detección total de las cuatro capturas de tráfico de red utilizadas para las pruebas, como se muestra en la siguiente tabla.

Tabla 20

Evaluación técnica de los tiempos

Modelo	Tiempo de entrenamiento	Tiempo de detección
CNN	56 minutos y 39 segundos	5509 ms
RNN	37 minutos y 21 segundos	4901 ms
CNN-RNN	2 horas, 39 minutos y 30 segundos	11423 ms
Total	**4 horas, 13 minutos y 30 segundos**	**21833 ms**

Nota. En la tabla se muestra la evaluación técnica de los tiempos de entrenamiento y detección de los tres modelos. Elaboración propia.

Como se observa en la Tabla 20, la red neuronal híbrida CNN-RNN tuvo un mayor tiempo de entrenamiento y detección en contraste con los otros dos modelos y la red neuronal recurrente fue la más rápida.

3.9. EVALUACIÓN ECONÓMICA

Para la evaluación económica se tomó en cuenta el costo de estudio, el costo de la recolección y procesamiento de los datos, el costo del entrenamiento y el costo del esfuerzo basado en el tiempo empleado.

Para determinar el costo de estudio se consideró el precio mensual del plan de Internet móvil, utilizado para la búsqueda de información durante la investigación y para la red de área local

utilizada para la recolección de los datos para el entrenamiento y las pruebas de las redes neuronales, como se muestra en la siguiente tabla.

Tabla 21

Costo de estudio

Descripción	Cantidad	Costo individual (Bs.)	Costo total (Bs.)
Plan de Internet móvil ilimitado	11 meses	198	2178
		Total	**2178**

Nota. En la tabla se muestra el costo de estudio. Elaboración propia.

En la tabla que se muestra a continuación, se resume el tiempo empleado en la recolección, procesamiento y entrenamiento de los tres modelos de redes neuronales.

Tabla 22

Resumen del tiempo de elaboración de los modelos

Descripción	Tiempo total
Recolección de los datos	4 horas, 32 minutos y 30 segundos
Procesamiento de los datos	5 minutos y 28 segundos
Entrenamiento de los modelos	4 horas, 13 minutos y 30 segundos
Total	**8 horas, 51 minutos y 28 segundos**

Nota. En la tabla se muestra el tiempo empleado en la recolección, procesamiento y entrenamiento de los modelos de redes neuronales. Elaboración propia.

Para la evaluación económica basada en el esfuerzo y tiempo dedicado se consideró como base un sueldo de tiempo completo de Bs.- 5000 al mes para el desarrollo de modelos de redes neuronales, considerando el tiempo de desarrollo de la investigación de 4 horas al día durante 11 meses, se obtiene un total de Bs.- 27,500, como se muestra a continuación.

$$2500 * 11 = 27500$$

En la siguiente tabla, se muestra el costo de las herramientas utilizadas para la evaluación preliminar, recolección de los datos, procesamiento de los datos, entrenamiento y pruebas de los modelos de redes neuronales.

Tabla 23

Costo de las herramientas

Herramienta	Descripción	Costo total (Bs.)
Jupyter Notebook	Evaluación preliminar, procesamiento de los datos y entrenamiento de los modelos	0
Kali Linux	Recolección de los datos	0
Wireshark	Recolección de los datos	0
Spyder	Pruebas de los modelos	0
	Total	**0**

Nota. En la tabla se muestra el costo de las herramientas. Elaboración propia.

En la tabla que se muestra a continuación, se observa el costo total para la evaluación económica de la investigación.

Tabla 24

Evaluación económica

Descripción	Costo total (Bs.)
Costo de estudio	2178
Esfuerzo y tiempo	27500
Costo de las herramientas	0
Total	**29678**

Nota. En la tabla se muestra el resumen de la evaluación económica. Elaboración propia.

Como se muestra en el resumen de la Tabla 24, el costo total estimado de la investigación fue de Bs.- 29,678.

CAPÍTULO 4

CONCLUSIONES Y RECOMENDACIONES

CAPÍTULO 4
CONCLUSIONES Y RECOMENDACIONES

4.1. CONCLUSIONES

La presente investigación sobre el entrenamiento de redes neuronales convolucionales, recurrentes e híbridas convolucionales recurrentes para realizar la comparación según la precisión de detección de amenazas en redes, específicamente ataques de denegación de servicio, suplantación de ARP y escaneo de puertos, llegó a la conclusión con base en los resultados obtenidos de las pruebas de hipótesis que, con un 95% de confianza, se acepta la hipótesis de investigación y se rechaza la hipótesis nula, estableciendo que existe una diferencia significativa entre la precisión de los tres modelos elaborados para la detección de intrusos en redes y que es consistentemente superior al 95%, cumpliendo con el objetivo general planteado.

Con respecto al primer objetivo específico, la recolección de información sobre los modelos de redes neuronales seleccionados y su aplicación en la detección de intrusos en redes ayudó a validar la aplicación de las redes convolucionales y recurrentes en la clasificación múltiple de datos numéricos y de texto, mientras que las investigaciones de los últimos años contribuyeron a la validación de la factibilidad de obtener una alta precisión de detección utilizando modelos híbridos.

El segundo objetivo específico permitió obtener una evaluación preliminar del desempeño de las tres redes neuronales mediante su entrenamiento utilizando el *dataset KDD Cup 1999*, lo cual contribuyó a la selección de las columnas durante la creación de un nuevo *dataset* con las amenazas seleccionadas para la clasificación y ayudó a mejorar las técnicas utilizadas para la elaboración de los modelos finales.

En lo que respecta a los siguientes dos objetivos específicos, se estableció una línea base de tráfico de red normal y se replicaron satisfactoriamente los ataques en un ambiente controlado, concluyendo con el tercer objetivo y utilizando la recolección y procesamiento de los datos para la elaboración de un *dataset* que posteriormente se utilizó para el entrenamiento de los tres modelos de redes neuronales, cumpliendo exitosamente con el cuarto objetivo específico de la investigación.

Una vez finalizado el entrenamiento de las redes neuronales, se elaboró un programa que permite introducir capturas de tráfico de red en un archivo CSV, procesar los datos y obtener la distribución de clases, resultados de la detección y el tiempo de detección de cada red neuronal, validando la capacidad de detección de intrusos en redes de los tres modelos.

Concluyendo con el último objetivo específico, se realizaron dos pruebas para aceptar o rechazar la hipótesis de investigación, comenzando por el análisis de varianza factorial de una vía ANOVA, utilizando 30 muestras de la precisión de cada modelo con 90 muestras en total, para determinar si existe una diferencia significativa entre la precisión de los tres modelos entrenados con el *dataset* elaborado y pruebas de *t* de *Student*, para verificar que la precisión de cada modelo es mayor al 95%, concluyendo en que la precisión de las redes neuronales convolucionales, recurrentes e híbridas convolucionales recurrentes alcanza el 95% y presenta diferencias entre los tres modelos.

4.2. RECOMENDACIONES

4.2.1. Recomendaciones metodológicas

Como recomendación metodológica en relación al desarrollo de modelos de redes neuronales, se sugiere la aplicación de técnicas de validación cruzada con respecto a la cantidad de columnas que los modelos necesitan para realizar la detección, monitoreando los parámetros seleccionados para garantizar una alta precisión y lograr optimizar el procesamiento de los datos, el entrenamiento y el tiempo de predicción.

En cuanto a la selección de modelos de redes neuronales para su implementación en sistemas de detección o prevención de intrusos en redes, se recomienda optar por estructuras menos complejas, es decir, con menos capas, dado que requieren menos recursos computacionales y son capaces de generar resultados óptimos en menos tiempo.

4.2.2. Recomendaciones académicas

Como recomendación académica, se sugiere ampliar las investigaciones sobre las aplicaciones de los diferentes modelos de redes neuronales en el área de la seguridad en redes, con un énfasis en los tipos de amenazas en redes que la implementación de estos modelos puede

ayudar a detectar. Además, se recomienda fomentar la comparación en la investigación de las redes neuronales, dado que ayuda a determinar el tipo de modelo que se adapta mejor a cada situación con base en resultados cuantificables.

4.2.3. Recomendaciones prácticas

En el ámbito de la aplicación práctica de los modelos, se sugiere la implementación de las redes neuronales en sistemas de detección o prevención de intrusos, ya que permiten una mejor visualización de los datos y el envío de alertas cuando existen valores fuera del rango de lo normal.

Se recomienda su implementación con un enfoque en la reducción del tiempo de procesamiento de los datos y de la detección, optimizando los modelos para que requieran menos recursos computacionales y tengan un bajo índice de falsos positivos y negativos.

BIBLIOGRAFÍA

BIBLIOGRAFÍA

Ahmad, Z., Khan, A., Shiang, C. W., Abdullah, J. y Ahmad, F. J. (2021). Network intrusion detection system: A systematic study of machine learning and deep learning approaches. Transactions On Emerging Telecommunications Technologies. https://doi.org/10.1002/ett.4150

Amarudin, Ferdiana, R. y Widyawan. (2020). A Systematic Literature Review of Intrusion Detection System for Network Security: Research Trends, Datasets and Methods. https://doi.org/10.1109/icicos51170.2020.9299068

Amidi, A. y Amidi, S. (s.f.-a). CS 230 - Convolutional Neural Networks Cheatsheet. Recuperado 10 de octubre de 2023, de https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-convolutional-neural-networks

Amidi, A. y Amidi, S. (s.f.-b). CS 230 - Recurrent Neural Networks Cheatsheet. Recuperado 10 de octubre de 2023, de https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks

Anaconda Navigator. (s.f.). Anaconda Documentation. Recuperado 19 de abril de 2023, de https://docs.anaconda.com/free/navigator/index.html

Burkov, A. (2019). The Hundred-page Machine Learning Book.

Burkov, A. (2020). Machine Learning Engineering. True Positive Incorporated.

Carbon. (s.f.). Carbon. https://carbon.now.sh/

Chollet, F. (2017). Deep Learning with Python. Manning Publications.

Cisco. (2022a, marzo 25). ¿Qué Es la Seguridad de Red? Recuperado 27 de marzo de 2023, de https://www.cisco.com/c/es_mx/products/security/what-is-network-security.html

Cisco. (2022b, abril 9). What Is A Wireless LAN (WLAN)? Recuperado 19 de abril de 2023, de https://www.cisco.com/c/en/us/products/wireless/wireless-lan.html

Collins English Dictionary. (s.f.). En Non-linear Definition And Meaning. Recuperado 27 de marzo de 2023, de https://www.collinsdictionary.com/dictionary/english/non-linear#:~:text=If%20you%20describe%20something%20as,a%20non%2Dlinear%20narrative%20structure

Cornejo Tarqui, F. R. (2020). Detección de malware en una red con machine learning [Tesis]. Universidad Mayor de San Andrés.

Creswell, J. W. y Creswell, J. D. (2017). Research Design: Qualitative, Quantitative, and Mixed Methods Approaches (5ta ed.). SAGE Publications.

CrowdStrike. (2023, 13 julio). 2023 Global Threat Report | CrowdStrike. crowdstrike.com. https://www.crowdstrike.com/global-threat-report/

Draw.io. (s.f.). https://www.drawio.com/

Dua, S. y Du, X. (2016). Data Mining and Machine Learning in Cybersecurity. CRC Press.

Erickson, J. (2008). Hacking: The Art of Exploitation (2da ed.). No Starch Press.

Forcepoint. (2022, 24 octubre). What Is Network Security? Recuperado 18 de abril de 2023, de https://www.forcepoint.com/cyber-edu/network-security

Géron, A. (2017). Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems. O'Reilly Media, Inc.

Goldberg, Y. (2017). Neural Network Methods in Natural Language Processing. Morgan y Claypool Publishers.

Goodfellow, I., Bengio, Y. y Courville, A. (2016). Deep Learning. MIT Press.

Gulli, A., Kapoor, A. y Pal, S. (2019). Deep Learning with TensorFlow 2 and Keras - Second Edition. Packt Publishing.

Haberfellner, R., De Weck, O., Fricke, E. y Vössner, S. (2019). Systems Engineering: Fundamentals and Applications. Springer.

Jupyter Notebook. (s.f.). Jupyter Notebook Documentation. Recuperado 19 de abril de 2023, de https://jupyter-notebook.readthedocs.io/en/latest/

Kali Linux Documentation. (s.f.). What Is Kali Linux? Recuperado 19 de abril de 2023, de https://www.kali.org/docs/introduction/what-is-kali-linux/

KDD Cup 1999 Data. (2018, 11 diciembre). Kaggle. Recuperado 18 de junio de 2023, de https://www.kaggle.com/datasets/galaxyh/kdd-cup-1999-data

Keras. (s.f.). Deep Learning For Humans. Recuperado 19 de abril de 2023, de https://keras.io/

Kim, D. y Solomon, M. G. (2016). Fundamentals of Information Systems Security.

Meliboev, A., Alikhanov, J. y Kim, W. (2020). 1D CNN based network intrusion detection with normalization on imbalanced data. 2020 International Conference On Artificial Intelligence In Information And Communication (ICAIIC). https://doi.org/10.1109/icaiic48513.2020.9064976

Newbold, P., Carlson, W. L. y Thorne, B. (2022). Statistics for Business and Economics, Global Edition (10ma ed.). Pearson Higher Ed.

Oracle VM VirtualBox. (s.f.). Recuperado 19 de abril de 2023, de https://www.virtualbox.org/

Patterson, J. y Gibson, A. (2017). Deep Learning: A Practitioner’s Approach. O’Reilly Media, Inc.

Python.org. (s.f.). Applications For Python. Recuperado 19 de abril de 2023, de https://www.python.org/about/apps/

Raschka, S. (2015). Python Machine Learning.

Ravichandiran, S. (2019). Hands-On Deep Learning Algorithms with Python: Master deep learning algorithms with extensive math by implementing them using TensorFlow. Packt Publishing Ltd.

Rios Aliaga, L. J. (2021). Modelo para la detección de escaneo de puertos de la computadora en una red WLAN [Tesis]. Universidad Mayor de San Andrés.

Russell, S. y Norvig, P. (2021). Artificial Intelligence: A Modern Approach (4ta ed.). Pearson Higher Ed.

Shimeall, T. y Spring, J. (2014). Introduction to Information Security: A Strategic-Based Approach. Elsevier. https://doi.org/10.1016/B978-1-59749-969-9.00009-2

Spyder. (s.f.). Spyder Website Contributors. Recuperado 19 de abril de 2023, de https://www.spyder-ide.org/

Sun, P., Liu, P., Li, Q., Liu, C., Lu, X., Hao, R. y Chen, J. (2020). DL-IDS: Extracting features using CNN-LSTM hybrid network for Intrusion Detection System. Security And Communication Networks, 2020, 1-11. https://doi.org/10.1155/2020/8890306

TensorFlow. (s.f.). Why TensorFlow. Recuperado 19 de abril de 2023, de https://www.tensorflow.org/about

Ubuntu. (s.f.). Ubuntu PC Operating System. Recuperado 19 de abril de 2023, de https://ubuntu.com/desktop

We Are Social y Meltwater. (2023). Digital 2023 Global Overview Report. En Datareportal. Datareportal. Recuperado 22 de septiembre de 2023, de https://datareportal.com/reports/digital-2023-global-overview-report?utm_source=Global_Digital_Reports&utm_medium=PDF&utm_campaign=Digital_2023&utm_content=Global_Overview_Foreword

Wireshark Documentation. (s.f.). Chapter 1. Introduction. Recuperado 19 de abril de 2023, de https://www.wireshark.org/docs/wsug_html_chunked/ChapterIntroduction.html

Witte, R. S. y Witte, J. S. (2017). Statistics (11va ed.). John Wiley y Sons.

Yao, R., Wang, N., Liu, Z., Chen, P. y Sheng, X. (2021). Intrusion detection system in the advanced metering Infrastructure: a Cross-Layer Feature-Fusion CNN-LSTM-Based approach. Sensors, 21(2), 626. https://doi.org/10.3390/s21020626

La Salle Universidad Bolivia

GLOSARIO

GLOSARIO DE TÉRMINOS TÉCNICOS

Anomalía: Cuando los resultados obtenidos se desvían de un rango establecido de valores normales.

Antivirus: Los programas o aplicaciones que se encargan de analizar, escanear, detectar, prevenir, aislar o remover en tiempo real programas y archivos con contenido malicioso de manera automática.

Aprendizaje automático: El desarrollo de algoritmos con la capacidad de adaptarse a diversas situaciones de manera automática, usualmente aplicados para tareas de regresión y clasificación.

Aprendizaje profundo: Modelos de redes neuronales que tienen múltiples capas para adaptarse y resolver problemas complejos mediante el entrenamiento basado en una gran cantidad de datos.

Clasificación: Método basado en el entrenamiento de modelos de aprendizaje automático o aprendizaje profundo para predecir la clase a la que pertenecen determinados datos, dicha clasificación puede ser binaria, cuando se utilizan solamente dos clases o múltiple cuando se emplean más de dos clases.

Dataset: Son los conjuntos de datos relacionados entre sí con respecto a una temática específica, utilizados para el entrenamiento de modelos de aprendizaje automático o aprendizaje profundo.

Detección de intrusos: Es el proceso de análisis y monitoreo de los eventos que ocurren en una red o en un solo sistema, con el propósito de identificar posibles amenazas y anomalías en tiempo real.

Firewall: Los dispositivos de *software* o *hardware* utilizados para controlar y monitorear en tiempo real el tráfico de red que entra o sale, con base en reglas establecidas predeterminadas que pueden ser modificadas.

Host: Los dispositivos, sistemas computacionales o servidores que forman parte de una misma red, de manera que pueden enviar y recibir datos.

Prevención de intrusos: Es la acción posterior a la detección de intrusos y consiste en responder a la existencia de anomalías o amenazas detectadas.

GLOSARIO DE ACRÓNIMOS Y ABREVIATURAS

Adam: Estimación adaptativa de momentos.

AIDS: Sistema de detección de intrusos basada en anomalías.

ANN: Redes neuronales artificiales.

ANOVA: Análisis de varianza factorial.

ARP: Protocolo de resolución de direcciones.

CNN: Red neuronal convolucional.

CSV: Valores separados por comas.

DDoS: Ataque de denegación de servicio distribuido.

DoS: Ataque de denegación de servicio.

FN: Falso negativo.

FP: Falso positivo.

FPR: Tasa de falsos positivos.

HIDS: Sistema de detección de intrusos basado en el *host*.

IDS: Sistema de detección de intrusos.

IP: Protocolo de Internet.

IPS: Sistema de prevención de intrusos.

KDD: Descubrimiento de conocimientos y minería de datos.

LAN: Red de área local.

LSTM: Memoria de largo corto plazo.

MAC: Control de acceso al medio.

MITM: *Man in the middle.*

Ms: Milisegundos.

NAT: Traducción de direcciones de red.

NIDS: Sistema de detección de intrusos basado en redes.

ReLU: Unidad lineal rectificada.

RF: Bosque aleatorio.

RNN: Red neuronal recurrente.

SIDS: Sistema de detección de intrusos basado en firmas.

SVM: Máquinas de vectores de soporte.

SYN: Sincronizar.

TanH: Tangente hiperbólica.

TCP: Protocolo de control de transmisiones.

TN: Verdadero negativo.

TP: Verdadero positivo.

TPR: Tasa de verdaderos positivos.

UDP: Protocolo de datagramas de usuario.

WLAN: Red de área local inalámbrica.

ANEXOS

ANEXO A: MODELO DE RED NEURONAL CONVOLUCIONAL PRELIMINAR

```
import warnings
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
from tensorflow.keras import backend as K
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv1D, MaxPooling1D, Flatten,
Dense, Dropout
# Cargar dataset
df = pd.read_csv('kddcup.data_10_percent.gz', header=None)

# Asignar los nombres de las columnas
df.columns = [
    'duration', 'protocol_type', 'service', 'flag', 'src_bytes', 'dst_bytes', 'land',
'wrong_fragment', 'urgent', 'hot',
    'num_failed_logins', 'logged_in', 'num_compromised', 'root_shell',
'su_attempted', 'num_root', 'num_file_creations',
    'num_shells', 'num_access_files', 'num_outbound_cmds', 'is_host_lo-
gin', 'is_guest_login', 'count', 'srv_count',
    'serror_rate', 'srv_serror_rate', 'rerror_rate', 'srv_rerror_rate',
'same_srv_rate', 'diff_srv_rate', 'srv_diff_host_rate',
    'dst_host_count', 'dst_host_srv_count', 'dst_host_same_srv_rate',
'dst_host_diff_srv_rate', 'dst_host_same_src_port_rate',
    'dst_host_srv_diff_host_rate', 'dst_host_serror_rate', 'dst_host_srv_se-
rror_rate', 'dst_host_rerror_rate',
    'dst_host_srv_rerror_rate', 'outcome'
]

# Eliminar valores duplicados y filas con valores faltantes
df.drop_duplicates(keep='first', inplace=True)
df.dropna(inplace=True, axis=1)

# Ingeniería de características

# Ignorar advertencias
warnings.filterwarnings("ignore")

# Codificar los valores numéricos utilizando puntuación Z
def encode_numeric(df, name, mean=None, sd=None):
    if mean is None:
        mean = df[name].mean()
    if sd is None:
        sd = df[name].std()

    df[name] = (df[name] - mean) / sd

# Codificar los valores de texto con One Hot Encoding y variables ficti-
cias
def encode_text(df, name):
    dummies = pd.get_dummies(df[name])
    for i in dummies.columns:
        dummy_name = f"{name}-{i}"
        df[dummy_name] = dummies[i]
    df.drop(name, axis=1, inplace=True)

# Codificar todos los valores según corresponda
encode_numeric(df, 'duration')
encode_text(df, 'protocol_type')
encode_text(df, 'service')
encode_text(df, 'flag')
encode_numeric(df, 'src_bytes')
encode_numeric(df, 'dst_bytes')
encode_text(df, 'land')
encode_numeric(df, 'wrong_fragment')
encode_numeric(df, 'urgent')
encode_numeric(df, 'hot')
encode_numeric(df, 'num_failed_logins')
encode_text(df, 'logged_in')
encode_numeric(df, 'num_compromised')
encode_numeric(df, 'root_shell')
encode_numeric(df, 'su_attempted')
encode_numeric(df, 'num_root')
encode_numeric(df, 'num_file_creations')
encode_numeric(df, 'num_shells')
encode_numeric(df, 'num_access_files')
encode_numeric(df, 'num_outbound_cmds')
encode_text(df, 'is_host_login')
encode_text(df, 'is_guest_login')
encode_numeric(df, 'count')
encode_numeric(df, 'srv_count')
encode_numeric(df, 'serror_rate')
encode_numeric(df, 'srv_serror_rate')
encode_numeric(df, 'rerror_rate')
encode_numeric(df, 'srv_rerror_rate')
encode_numeric(df, 'same_srv_rate')
encode_numeric(df, 'diff_srv_rate')
encode_numeric(df, 'srv_diff_host_rate')
encode_numeric(df, 'dst_host_count')
encode_numeric(df, 'dst_host_srv_count')
encode_numeric(df, 'dst_host_same_srv_rate')
encode_numeric(df, 'dst_host_diff_srv_rate')
encode_numeric(df, 'dst_host_same_src_port_rate')
encode_numeric(df, 'dst_host_srv_diff_host_rate')
encode_numeric(df, 'dst_host_serror_rate')
encode_numeric(df, 'dst_host_srv_serror_rate')
encode_numeric(df, 'dst_host_rerror_rate')
encode_numeric(df, 'dst_host_srv_rerror_rate')

# Eliminar filas con valores faltantes
df.dropna(inplace=True,axis=1)

# Extraer la columna de características y definir variables
x_columns = df.columns.drop('outcome')
x = df[x_columns].values

# Codificar los valores de texto con variables ficticias
outcomes = pd.get_dummies(df['outcome'])
y = outcomes.values

# Mostras los nombres de las columnas
print(df.columns.tolist())

# Dividir en datos de entrenamiento y datos de prueba
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.25, ran-
dom_state=42)

# Preparar los datos de entrada
x_train = x_train.reshape(x_train.shape[0], x_train.shape[1], 1)
x_test = x_test.reshape(x_test.shape[0], x_test.shape[1], 1)

# Iniciar el modelo CNN
model = Sequential()
model.add(Conv1D(32, 3, activation='relu',
input_shape=(x_train.shape[1], 1)))
model.add(MaxPooling1D(2))
model.add(Conv1D(64, 3, activation='relu'))
model.add(MaxPooling1D(2))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(y.shape[1], activation='softmax'))

# Definir funciones para calcular la exactitud, precisión y exhaustividad

def calc_accuracy(y_true, y_pred):
    TP = K.sum(K.round(K.clip(y_true * y_pred, 0, 1)))
    TN = K.sum(K.round(K.clip((1 - y_true) * (1 - y_pred), 0, 1)))
    FP = K.sum(K.round(K.clip((1 - y_true) * y_pred, 0, 1)))
    FN = K.sum(K.round(K.clip(y_true * (1 - y_pred), 0, 1)))

    accuracy = (TP + TN) / (TP + TN + FP + FN + K.epsilon())
    return accuracy

def calc_precision(y_true, y_pred):
    TP = K.sum(K.round(K.clip(y_true * y_pred, 0, 1)))
    FP = K.sum(K.round(K.clip((1 - y_true) * y_pred, 0, 1)))

    precision = TP / (TP + FP + K.epsilon())
    return precision

def calc_recall(y_true, y_pred):
```

```
    TP = K.sum(K.round(K.clip(y_true * y_pred, 0, 1)))
    FN = K.sum(K.round(K.clip(y_true * (1 - y_pred), 0, 1)))

    recall = TP / (TP + FN + K.epsilon())
    return recall

# Compilar el modelo
model.compile(loss='categorical_crossentropy', optimizer='adam', metri-
cs=[calc_accuracy, calc_precision, calc_recall])

# Entrenamiento del modelo
CNN = model.fit(x_train, y_train, validation_data=(x_test, y_test),
batch_size=256, epochs=20)

# Realizar predicciones con el modelo entrenado
y_pred = model.predict(x_test)
y_pred_classes = np.argmax(y_pred, axis=1)
y_test_classes = np.argmax(y_test, axis=1)
# Evaluación de las predicciones
accuracy_pred = calc_accuracy(y_test, y_pred)
precision_pred = calc_precision(y_test, y_pred)
recall_pred = calc_recall(y_test, y_pred)

# Resultados
print(f'Exactitud de las predicciones: {accuracy_pred:.4f}')
print(f'Precisión de las predicciones: {precision_pred:.4f}')
print(f'Exhaustividad de las predicciones: {recall_pred:.4f}')
```

```
# Generar una matriz de confusión con etiquetas
def confusion_matrix_labels(y_test, y_pred):
    cm = confusion_matrix(y_test, y_pred)
    cm_df = pd.DataFrame(cm)

    labels = ['back', 'buffer_overflow', 'ftp_write', 'guess_passwd', 'imap',
'ipsweep', 'land', 'loadmodule',
          'multihop', 'neptune', 'nmap', 'normal', 'perl', 'phf', 'pod', 'portsweep',
'rootkit', 'satan',
          'smurf', 'teardrop', 'warezclient', 'warezmaster']

    plt.figure(figsize=(20, 15))
    sns.set(font_scale=1.4)

    custom_cmap = "Purples"

    sns.heatmap(cm_df, annot=True, annot_kws={"size": 12}, fmt='g', xti-
cklabels=labels, yticklabels=labels, cmap=custom_cmap)
    plt.ylabel('Clases reales')
    plt.xlabel('Clases predecidas')

    # Guardar la matriz de confusión
    plt.savefig('Matriz-CNN.png')

    plt.show()

confusion_matrix_labels(y_test_classes, y_pred_classes)
```

ANEXO B: MODELO DE RED NEURONAL RECURRENTE PRELIMINAR

```
import warnings
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
from tensorflow.keras import backend as K
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Dropout

# Cargar dataset
df = pd.read_csv('kddcup.data_10_percent.gz', header=None)

# Asignar los nombres de las columnas
df.columns = [
    'duration', 'protocol_type', 'service', 'flag', 'src_bytes', 'dst_bytes', 'land',
'wrong_fragment', 'urgent', 'hot',
    'num_failed_logins', 'logged_in', 'num_compromised', 'root_shell',
'su_attempted', 'num_root', 'num_file_creations',
    'num_shells', 'num_access_files', 'num_outbound_cmds', 'is_host_lo-
gin', 'is_guest_login', 'count', 'srv_count',
    'serror_rate', 'srv_serror_rate', 'rerror_rate', 'srv_rerror_rate',
'same_srv_rate', 'diff_srv_rate', 'srv_diff_host_rate',
    'dst_host_count', 'dst_host_srv_count', 'dst_host_same_srv_rate',
'dst_host_diff_srv_rate', 'dst_host_same_src_port_rate',
    'dst_host_srv_diff_host_rate', 'dst_host_serror_rate', 'dst_host_srv_se-
rror_rate', 'dst_host_rerror_rate',
    'dst_host_srv_rerror_rate', 'outcome'
]

# Eliminar valores duplicados y filas con valores faltantes
df.drop_duplicates(keep='first', inplace=True)
df.dropna(inplace=True, axis=1)

# Ingeniería de características

# Ignorar advertencias
warnings.filterwarnings("ignore")

# Codificar los valores numéricos utilizando puntuación Z
def encode_numeric(df, name, mean=None, sd=None):
    if mean is None:
        mean = df[name].mean()
    if sd is None:
        sd = df[name].std()

    df[name] = (df[name] - mean) / sd

# Codificar los valores de texto con One Hot Encoding y variables ficti-
cias
def encode_text(df, name):
    dummies = pd.get_dummies(df[name])
    for i in dummies.columns:
        dummy_name = f"{name}-{i}"
        df[dummy_name] = dummies[i]
    df.drop(name, axis=1, inplace=True)

# Codificar todos los valores según corresponda
encode_numeric(df, 'duration')
encode_text(df, 'protocol_type')
encode_text(df, 'service')
encode_text(df, 'flag')
encode_numeric(df, 'src_bytes')
encode_numeric(df, 'dst_bytes')
encode_text(df, 'land')
encode_numeric(df, 'wrong_fragment')
encode_numeric(df, 'urgent')
encode_numeric(df, 'hot')
encode_numeric(df, 'num_failed_logins')
encode_text(df, 'logged_in')
encode_numeric(df, 'num_compromised')
encode_numeric(df, 'root_shell')
encode_numeric(df, 'su_attempted')
encode_numeric(df, 'num_root')
encode_numeric(df, 'num_file_creations')
encode_numeric(df, 'num_shells')
encode_numeric(df, 'num_access_files')
encode_numeric(df, 'num_outbound_cmds')
encode_text(df, 'is_host_login')
encode_text(df, 'is_guest_login')
encode_numeric(df, 'count')
encode_numeric(df, 'srv_count')
encode_numeric(df, 'serror_rate')
encode_numeric(df, 'srv_serror_rate')
encode_numeric(df, 'rerror_rate')
encode_numeric(df, 'srv_rerror_rate')
encode_numeric(df, 'same_srv_rate')
encode_numeric(df, 'diff_srv_rate')
encode_numeric(df, 'srv_diff_host_rate')
encode_numeric(df, 'dst_host_count')
encode_numeric(df, 'dst_host_srv_count')
encode_numeric(df, 'dst_host_same_srv_rate')
encode_numeric(df, 'dst_host_diff_srv_rate')
encode_numeric(df, 'dst_host_same_src_port_rate')
encode_numeric(df, 'dst_host_srv_diff_host_rate')
encode_numeric(df, 'dst_host_serror_rate')
encode_numeric(df, 'dst_host_srv_serror_rate')
encode_numeric(df, 'dst_host_rerror_rate')
encode_numeric(df, 'dst_host_srv_rerror_rate')

# Eliminar filas con valores faltantes
df.dropna(inplace=True,axis=1)

# Extraer la columna de características y definir variables
x_columns = df.columns.drop('outcome')
x = df[x_columns].values

# Codificar los valores de texto con variables ficticias
outcomes = pd.get_dummies(df['outcome'])
y = outcomes.values

# Mostras los nombres de las columnas
print(df.columns.tolist())

# Dividir en datos de entrenamiento y datos de prueba
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.25, ran-
dom_state=42)

# Preparar los datos de entrada
x_train = np.reshape(x_train, (x_train.shape[0], 1, x_train.shape[1]))
x_test = np.reshape(x_test, (x_test.shape[0], 1, x_test.shape[1]))

# Iniciar el modelo RNN
model = Sequential()
model.add(LSTM(64, activation='tanh', input_shape=(1,
x_train.shape[2]), return_sequences=True))
model.add(LSTM(64, activation='tanh'))
model.add(Dropout(0.5))
model.add(Dense(y.shape[1], activation='softmax'))

# Definir funciones para calcular la exactitud, precisión y exhaustividad
def calc_accuracy(y_true, y_pred):
    TP = K.sum(K.round(K.clip(y_true * y_pred, 0, 1)))
    TN = K.sum(K.round(K.clip((1 - y_true) * (1 - y_pred), 0, 1)))
    FP = K.sum(K.round(K.clip((1 - y_true) * y_pred, 0, 1)))
    FN = K.sum(K.round(K.clip(y_true * (1 - y_pred), 0, 1)))

    accuracy = (TP + TN) / (TP + TN + FP + FN + K.epsilon())
    return accuracy

def calc_precision(y_true, y_pred):
    TP = K.sum(K.round(K.clip(y_true * y_pred, 0, 1)))
    FP = K.sum(K.round(K.clip((1 - y_true) * y_pred, 0, 1)))

    precision = TP / (TP + FP + K.epsilon())
    return precision

def calc_recall(y_true, y_pred):
    TP = K.sum(K.round(K.clip(y_true * y_pred, 0, 1)))
    FN = K.sum(K.round(K.clip(y_true * (1 - y_pred), 0, 1)))

    recall = TP / (TP + FN + K.epsilon())
    return recall
```

```
# Compilar el modelo
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=[calc_accuracy, calc_precision, calc_recall])

# Entrenamiento del modelo
RNN = model.fit(x_train, y_train, validation_data=(x_test, y_test), batch_size=256, epochs=20)

# Realizar predicciones con el modelo entrenado
y_pred = model.predict(x_test)
y_pred_classes = np.argmax(y_pred, axis=1)
y_test_classes = np.argmax(y_test, axis=1)

# Evaluación de las predicciones
accuracy_pred = calc_accuracy(y_test, y_pred)
precision_pred = calc_precision(y_test, y_pred)
recall_pred = calc_recall(y_test, y_pred)

# Resultados
print(f'Exactitud de las predicciones: {accuracy_pred:.4f}')
print(f'Precisión de las predicciones: {precision_pred:.4f}')
print(f'Exhaustividad de las predicciones: {recall_pred:.4f}')

# Generar una matriz de confusión con etiquetas
def confusion_matrix_labels(y_test, y_pred):
    cm = confusion_matrix(y_test, y_pred)
    cm_df = pd.DataFrame(cm)
    labels = ['back', 'buffer_overflow', 'ftp_write', 'guess_passwd', 'imap', 'ipsweep', 'land', 'loadmodule',
          'multihop', 'neptune', 'nmap', 'normal', 'perl', 'phf', 'pod', 'portsweep', 'rootkit', 'satan',
          'smurf', 'teardrop', 'warezclient', 'warezmaster']

    plt.figure(figsize=(20, 15))
    sns.set(font_scale=1.4)

    custom_cmap = "Purples"

    sns.heatmap(cm_df, annot=True, annot_kws={"size": 12}, fmt='g', xticklabels=labels, yticklabels=labels, cmap=custom_cmap)
    plt.ylabel('Clases reales')
    plt.xlabel('Clases predecidas')

    # Guardar la matriz de confusión
    plt.savefig('Matriz-RNN.png')

    plt.show()

confusion_matrix_labels(y_test_classes, y_pred_classes)
```

ANEXO C: MODELO DE RED NEURONAL HÍBRIDA CONVOLUCIONAL RECURRENTE PRELIMINAR

```
import warnings
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
from tensorflow.keras import backend as K
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv1D, MaxPooling1D, LSTM, Dense, Dropout

# Cargar dataset
df = pd.read_csv('kddcup.data_10_percent.gz', header=None)

# Asignar los nombres de las columnas
df.columns = [
    'duration', 'protocol_type', 'service', 'flag', 'src_bytes', 'dst_bytes', 'land', 'wrong_fragment', 'urgent', 'hot',
    'num_failed_logins', 'logged_in', 'num_compromised', 'root_shell', 'su_attempted', 'num_root', 'num_file_creations',
    'num_shells', 'num_access_files', 'num_outbound_cmds', 'is_host_login', 'is_guest_login', 'count', 'srv_count',
    'serror_rate', 'srv_serror_rate', 'rerror_rate', 'srv_rerror_rate', 'same_srv_rate', 'diff_srv_rate', 'srv_diff_host_rate',
    'dst_host_count', 'dst_host_srv_count', 'dst_host_same_srv_rate', 'dst_host_diff_srv_rate', 'dst_host_same_src_port_rate',
    'dst_host_srv_diff_host_rate', 'dst_host_serror_rate', 'dst_host_srv_serror_rate', 'dst_host_rerror_rate',
    'dst_host_srv_rerror_rate', 'outcome'
]

# Eliminar valores duplicados y filas con valores faltantes
df.drop_duplicates(keep='first', inplace=True)
df.dropna(inplace=True, axis=1)

# Ingeniería de características

# Ignorar advertencias
warnings.filterwarnings("ignore")

# Codificar los valores numéricos utilizando puntuación Z
def encode_numeric(df, name, mean=None, sd=None):
    if mean is None:
        mean = df[name].mean()
    if sd is None:
        sd = df[name].std()

    df[name] = (df[name] - mean) / sd

# Codificar los valores de texto con One Hot Encoding y variables ficticias
def encode_text(df, name):
    dummies = pd.get_dummies(df[name])
    for i in dummies.columns:
        dummy_name = f"{name}-{i}"
        df[dummy_name] = dummies[i]
    df.drop(name, axis=1, inplace=True)

# Codificar todos los valores según corresponda
encode_numeric(df, 'duration')
encode_text(df, 'protocol_type')
encode_text(df, 'service')
encode_text(df, 'flag')
encode_numeric(df, 'src_bytes')
encode_numeric(df, 'dst_bytes')
encode_text(df, 'land')
encode_numeric(df, 'wrong_fragment')
encode_numeric(df, 'urgent')
encode_numeric(df, 'hot')
encode_numeric(df, 'num_failed_logins')
encode_text(df, 'logged_in')
encode_numeric(df, 'num_compromised')
encode_numeric(df, 'root_shell')
encode_numeric(df, 'su_attempted')
encode_numeric(df, 'num_root')
encode_numeric(df, 'num_file_creations')
encode_numeric(df, 'num_shells')
encode_numeric(df, 'num_access_files')
encode_numeric(df, 'num_outbound_cmds')
encode_text(df, 'is_host_login')
encode_text(df, 'is_guest_login')
encode_numeric(df, 'count')
encode_numeric(df, 'srv_count')
encode_numeric(df, 'serror_rate')
encode_numeric(df, 'srv_serror_rate')
encode_numeric(df, 'rerror_rate')
encode_numeric(df, 'srv_rerror_rate')
encode_numeric(df, 'same_srv_rate')
encode_numeric(df, 'diff_srv_rate')
encode_numeric(df, 'srv_diff_host_rate')
encode_numeric(df, 'dst_host_count')
encode_numeric(df, 'dst_host_srv_count')
encode_numeric(df, 'dst_host_same_srv_rate')
encode_numeric(df, 'dst_host_diff_srv_rate')
encode_numeric(df, 'dst_host_same_src_port_rate')
encode_numeric(df, 'dst_host_srv_diff_host_rate')
encode_numeric(df, 'dst_host_serror_rate')
encode_numeric(df, 'dst_host_srv_serror_rate')
encode_numeric(df, 'dst_host_rerror_rate')
encode_numeric(df, 'dst_host_srv_rerror_rate')

# Eliminar filas con valores faltantes
df.dropna(inplace=True,axis=1)

# Extraer la columna de características y definir variables
x_columns = df.columns.drop('outcome')
x = df[x_columns].values

# Codificar los valores de texto con variables ficticias
outcomes = pd.get_dummies(df['outcome'])
y = outcomes.values

# Mostras los nombres de las columnas
print(df.columns.tolist())

# Dividir en datos de entrenamiento y datos de prueba
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.25, random_state=42)

# Preparar los datos de entrada
x_train = x_train.reshape(x_train.shape[0], x_train.shape[1], 1)
x_test = x_test.reshape(x_test.shape[0], x_test.shape[1], 1)

# Iniciar el modelo híbrido CNN-RNN
model = Sequential()
model.add(Conv1D(32, 3, activation='relu',
input_shape=(x_train.shape[1], 1)))
model.add(MaxPooling1D(2))
model.add(Conv1D(64, 3, activation='relu'))
model.add(MaxPooling1D(2))
model.add(LSTM(64, return_sequences=True, activation='tanh'))
model.add(LSTM(64, activation='tanh'))
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(y.shape[1], activation='softmax'))

# Definir funciones para calcular la exactitud, precisión y exhaustividad
def calc_accuracy(y_true, y_pred):
    TP = K.sum(K.round(K.clip(y_true * y_pred, 0, 1)))
    TN = K.sum(K.round(K.clip((1 - y_true) * (1 - y_pred), 0, 1)))
    FP = K.sum(K.round(K.clip((1 - y_true) * y_pred, 0, 1)))
    FN = K.sum(K.round(K.clip(y_true * (1 - y_pred), 0, 1)))

    accuracy = (TP + TN) / (TP + TN + FP + FN + K.epsilon())
    return accuracy

def calc_precision(y_true, y_pred):
    TP = K.sum(K.round(K.clip(y_true * y_pred, 0, 1)))
    FP = K.sum(K.round(K.clip((1 - y_true) * y_pred, 0, 1)))
```

```
    precision = TP / (TP + FP + K.epsilon())
    return precision

def calc_recall(y_true, y_pred):
    TP = K.sum(K.round(K.clip(y_true * y_pred, 0, 1)))
    FN = K.sum(K.round(K.clip(y_true * (1 - y_pred), 0, 1)))

    recall = TP / (TP + FN + K.epsilon())
    return recall

# Compilar el modelo
model.compile(loss='categorical_crossentropy', optimizer='adam', metri-
cs=[calc_accuracy, calc_precision, calc_recall])

# Entrenamiento del modelo
CNN_RNN = model.fit(x_train, y_train, validation_data=(x_test, y_test),
batch_size=256, epochs=20)

# Realizar predicciones con el modelo entrenado
y_pred = model.predict(x_test)
y_pred_classes = np.argmax(y_pred, axis=1)
y_test_classes = np.argmax(y_test, axis=1)

# Evaluación de las predicciones
accuracy_pred = calc_accuracy(y_test, y_pred)
precision_pred = calc_precision(y_test, y_pred)
recall_pred = calc_recall(y_test, y_pred)

# Resultados
print(f'Exactitud de las predicciones: {accuracy_pred:.4f}')
print(f'Precisión de las predicciones: {precision_pred:.4f}')
print(f'Exhaustividad de las predicciones: {recall_pred:.4f}')

# Generar una matriz de confusión con etiquetas
def confusion_matrix_labels(y_test, y_pred):
    cm = confusion_matrix(y_test, y_pred)
    cm_df = pd.DataFrame(cm)

    labels = ['back', 'buffer_overflow', 'ftp_write', 'guess_passwd', 'imap',
'ipsweep', 'land', 'loadmodule',
    'neptune', 'nmap', 'normal', 'perl', 'phf', 'pod', 'portsweep', 'rootkit',
'satan',
          'smurf', 'teardrop', 'warezclient', 'warezmaster']

    plt.figure(figsize=(20, 15))
    sns.set(font_scale=1.4)

    custom_cmap = "Purples"

    sns.heatmap(cm_df, annot=True, annot_kws={"size": 12}, fmt='g', xti-
cklabels=labels, yticklabels=labels, cmap=custom_cmap)
    plt.ylabel('Clases reales')
    plt.xlabel('Clases predecidas')

    # Guardar la matriz de confusión
    plt.savefig('Matriz-CNN-RNN.png')

    plt.show()

confusion_matrix_labels(y_test_classes, y_pred_classes)
```

ANEXO D: PROCESAMIENTO DE LOS DATOS

```
import time
import warnings
import pandas as pd
import numpy as np
from tqdm import tqdm

# Ignorar advertencias
warnings.filterwarnings("ignore")

# Preparar el DataFrame
def format_df(df):
    # Asignar los nombres de las columnas
    df.columns = ['No.', 'Delta time', 'Src add', 'Dst add', 'Protocol', 'Length',
'Src port',
                  'Dst port', 'Flags', 'Duplicate IP add', 'Severity', 'Info', 'Window
size']

    # Eliminar la primera fila
    df = df.drop(0)

    # Eliminar columnas que no se utilizarán
    df = df.drop('No.', axis=1)
    df = df.drop('Info', axis=1)

    # Eliminar columnas duplicadas
    df = df.drop_duplicates()

    return df

# Agregar la columna con las etiquetas y asignar valores
def label_column(df, label):
    df['Label'] = label
    return df

# Agregar 0 si existe un valor nulo en la columna con valores numéricos
def add_zero(row, column_name):
    if pd.isna(row[column_name]):
        return 0
    else:
        return row[column_name]

# Agregar -1 si existe un valor nulo que no puede ser 0 en la columna
con valores numéricos
def add_one(row, column_name):
    if pd.isna(row[column_name]):
        return -1
    else:
        return row[column_name]

# Agregar none si existe un valor nulo en la columna con valores de tex-
to
def add_null(row, column_name):
    if pd.isna(row[column_name]):
        return 'none'
    else:
        return row[column_name]

# Agregar una columna nueva para Duplicate IP add
def duplicate_row(df):
    if "Duplicate IP add" not in df.columns:
        df["Duplicate detected"] = 0
        return df
    df = df.append({"Duplicate detected": "Duplicate detected"}, ignore_in-
dex=True)
    for index, row in df.iterrows():
        if pd.isna(row["Duplicate IP add"]) or row["Duplicate IP add"] == "" or
row["Duplicate IP add"] == "Duplicate detected":
            df.at[index, "Duplicate detected"] = 0
        else:
            df.at[index, "Duplicate detected"] = 1

    return df

# Agregar una columna nueva para Src add
def same_network_src(df):
    if "Src add" not in df.columns:
        df["SameNet src add"] = 0
        return df
    df = df.append({"SameNet src add": "SameNet src add"}, ignore_in-
dex=True)
    for index, row in df.iterrows():
        src_add = str(row["Src add"]).strip()
        if src_add.startswith("192.168"):
            df.at[index, "SameNet src add"] = 1
        else:
            df.at[index, "SameNet src add"] = 0

    return df

# Agregar una columna nueva para Dst add
def same_network_dst(df):
    if "Dst add" not in df.columns:
        df["SameNet dst add"] = 0
        return df
    df = df.append({"SameNet dst add": "SameNet dst add"}, ignore_in-
dex=True)
    for index, row in df.iterrows():
        dst_add = str(row["Dst add"]).strip()
        if dst_add.startswith("192.168"):
            df.at[index, "SameNet dst add"] = 1
        else:
            df.at[index, "SameNet dst add"] = 0

    return df

# Función para mostrar una barra de progreso
def loading_bar(n, desc, color):
    total = len(n)
    prefix = f"{desc}: "
    for i, item in enumerate(n, 1):
        bar_length = 30
        percent = int(i / total * bar_length)
        bar = f"\033[1;{color}m\033[0m" * percent + "-" * (bar_length - per-
cent)
        progress_text = f"{i/total*100:.2f}%"
        print(f"\r{prefix}|{bar}| {progress_text}", end='', flush=True)
        yield item

# Función para mostrar el tiempo en horas, minutos y segundos
def time_format(seconds):
    minutes, seconds = divmod(seconds, 60)
    hours, minutes = divmod(minutes, 60)
    return hours, minutes, seconds

# Cargar dataset de ARP spoofing
df_ARP = pd.read_csv('csv/ARP-60-dataset.csv', header=None)

# Aplicar las funciones
df_ARP = format_df(df_ARP)
df_ARP = label_column(df_ARP, 'ARP spoofing')

# Cargar datasets de port scanning
df_portscan_12 = pd.read_csv('csv/PortScan-12-dataset.csv',
header=None)
df_portscan_140 = pd.read_csv('csv/PortScan-140-dataset.csv',
header=None)

# Aplicar las funciones
df_portscan_12 = format_df(df_portscan_12)
df_portscan_140 = format_df(df_portscan_140)

# Unir los objetos DataFrame y agregar etiquetas
df_portscan = pd.concat([df_portscan_12, df_portscan_140], ignore_in-
dex=True)
df_portscan = label_column(df_portscan, 'Port scan')

# Cargar dataset de DoS
df_DoS = pd.read_csv('csv/DoS-30-dataset.csv', header=None)

# Aplicar las funciones
df_DoS = format_df(df_DoS)
df_DoS = label_column(df_DoS, 'DoS')

# Cargar dataset de tráfico normal
```

```
df_normal = pd.read_csv('csv/Normal-60-dataset.csv', header=None)

# Aplicar las funciones
df_normal = format_df(df_normal)
df_normal = label_column(df_normal, 'Normal')

# Unir los 4 objetos DataFrame
df_dataset = pd.concat([df_ARP, df_portscan, df_DoS, df_normal], ignore_index=True)
print("Número de filas: ",df_dataset.shape[0])

# Iniciar el cronómetro
tiempo_inicial = time.time()

progress_bar = loading_bar(range(12), "Procesando", "38;5;147")

# Aplicar las funciones
df_dataset['Delta time'] = df_dataset.apply(lambda row: add_zero(row, 'Delta time'), axis=1)
next(progress_bar)

df_dataset['Protocol'] = df_dataset.apply(lambda row: add_null(row, 'Protocol'), axis=1)
next(progress_bar)

df_dataset['Length'] = df_dataset.apply(lambda row: add_zero(row, 'Length'), axis=1)
next(progress_bar)

df_dataset['Src port'] = df_dataset.apply(lambda row: add_zero(row, 'Src port'), axis=1)
next(progress_bar)

df_dataset['Dst port'] = df_dataset.apply(lambda row: add_zero(row, 'Dst port'), axis=1)
next(progress_bar)

df_dataset['Flags'] = df_dataset.apply(lambda row: add_null(row, 'Flags'), axis=1)
next(progress_bar)

df_dataset['Severity'] = df_dataset.apply(lambda row: add_null(row, 'Severity'), axis=1)
next(progress_bar)

df_dataset['Window size'] = df_dataset.apply(lambda row: add_one(row, 'Window size'), axis=1)
next(progress_bar)

df_dataset = df_dataset.drop_duplicates()
next(progress_bar)

try:
    if 'Duplicate IP add' in df_dataset.columns:
        df_dataset = duplicate_row(df_dataset)
        df_dataset = df_dataset.drop('Duplicate IP add', axis=1)
        df_dataset.drop(df_dataset.index[-1], inplace=True)
    next(progress_bar)
except KeyError:
    pass

try:
    if 'Src add' in df_dataset.columns:
        df_dataset = same_network_src(df_dataset)
        df_dataset = df_dataset.drop('Src add', axis=1)
        df_dataset.drop(df_dataset.index[-1], inplace=True)
    next(progress_bar)
except KeyError:
    pass

try:
    if 'Dst add' in df_dataset.columns:
        df_dataset = same_network_dst(df_dataset)
        df_dataset = df_dataset.drop('Dst add', axis=1)
        df_dataset.drop(df_dataset.index[-1], inplace=True)
    next(progress_bar)
except KeyError:
    pass

# Detener el cronómetro
tiempo_final = time.time()

# Calcular el tiempo total
tiempo_total = tiempo_final - tiempo_inicial
hours, minutes, seconds = time_format(tiempo_total)
print(f"\nTiempo total del procesamiento de los datos: {int(hours)} horas, {int(minutes)} minutos y {int(seconds)} segundos")

# Convertir las columnas a los tipos de datos especificados
df_dataset['Delta time'] = df_dataset['Delta time'].astype('float32')
df_dataset['Length'] = df_dataset['Length'].astype('int32')
df_dataset['Src port'] = df_dataset['Src port'].astype('int32')
df_dataset['Dst port'] = df_dataset['Dst port'].astype('int32')
df_dataset['Window size'] = df_dataset['Window size'].astype('int32')
df_dataset['Duplicate detected'] = df_dataset['Duplicate detected'].astype('int32')
df_dataset['SameNet src add'] = df_dataset['SameNet src add'].astype('int32')
df_dataset['SameNet dst add'] = df_dataset['SameNet dst add'].astype('int32')

# Guardar como CSV
df_dataset.to_csv('dataset/network-traffic-full-dataset.csv', index=False)

# Mezclar dataset
df_shuffled = df_dataset.sample(frac=1).reset_index(drop=True)

# Guardar como CSV
df_shuffled.to_csv('dataset/network-traffic-dataset-shuffled.csv', index=False)
```

ANEXO E: ENTRENAMIENTO DE LA RED NEURONAL CONVOLUCIONAL

```
import time
import warnings
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import tensorflow as tf
from sklearn.metrics import confusion_matrix
from tensorflow.keras import backend as K
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv1D, MaxPooling1D, Flatten,
Dense, BatchNormalization, Dropout
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.optimizers import Adam
from sklearn.model_selection import StratifiedShuffleSplit

# Ignorar advertencias
warnings.filterwarnings("ignore")

# Cargar dataset
df = pd.read_csv('dataset/network-traffic-dataset-shuffled.csv',
header=None, low_memory=False)

# Asignar los nombres de las columnas
df.columns = ['Delta time', 'Protocol', 'Length', 'Src port', 'Dst port', 'Flags',
'Severity',
              'Window size', 'Label', 'Duplicate detected', 'SameNet src add',
'SameNet dst add']

df = df.drop(0)
df = df.drop_duplicates()
df = df.dropna(axis=1, how='all')

# Convertir las columnas a los tipos de datos especificados
df['Delta time'] = df['Delta time'].astype('float32')
df['Length'] = df['Length'].astype('int32')
df['Src port'] = df['Src port'].astype('int32')
df['Dst port'] = df['Dst port'].astype('int32')
df['Window size'] = df['Window size'].astype('int32')
df['Duplicate detected'] = df['Duplicate detected'].astype('int32')
df['SameNet src add'] = df['SameNet src add'].astype('int32')
df['SameNet dst add'] = df['SameNet dst add'].astype('int32')

# Ingeniería de características

# Codificar los valores de texto con One Hot Encoding y variables ficti-
cias
def encode_text(df, name):
    dummies = pd.get_dummies(df[name])
    for i in dummies.columns:
        dummy_name = f"{name}-{i}"
        df[dummy_name] = dummies[i]
    df.drop(name, axis = 1, inplace = True)

# Codificar todos los valores según corresponda
encode_text(df, 'Protocol')
encode_text(df, 'Flags')
encode_text(df, 'Severity')

# Extraer la columna de características y definir variables
x_columns = df.columns.drop('Label')
x = df[x_columns].values

# Codificar los valores de texto con variables ficticias
outcomes = pd.get_dummies(df['Label'])
y = outcomes.values

# Mostras los nombres de las columnas
print(df.columns.tolist())

# Definir funciones para calcular la exactitud, precisión y exhaustividad
def calc_accuracy(y_true, y_pred):
    TP = K.sum(K.round(K.clip(y_true * y_pred, 0, 1)))
    TN = K.sum(K.round(K.clip((1 - y_true) * (1 - y_pred), 0, 1)))
    FP = K.sum(K.round(K.clip((1 - y_true) * y_pred, 0, 1)))
    FN = K.sum(K.round(K.clip(y_true * (1 - y_pred), 0, 1)))
    try:
        accuracy = (TP + TN) / (TP + TN + FP + FN + K.epsilon())
    except ZeroDivisionError:
        accuracy = 0
    return accuracy

def calc_precision(y_true, y_pred):
    TP = K.sum(K.round(K.clip(y_true * y_pred, 0, 1)))
    FP = K.sum(K.round(K.clip((1 - y_true) * y_pred, 0, 1)))

    try:
        precision = TP / (TP + FP + K.epsilon())
    except ZeroDivisionError:
        precision = 0
    return precision

def calc_recall(y_true, y_pred):
    TP = K.sum(K.round(K.clip(y_true * y_pred, 0, 1)))
    FN = K.sum(K.round(K.clip(y_true * (1 - y_pred), 0, 1)))

    try:
        recall = TP / (TP + FN + K.epsilon())
    except ZeroDivisionError:
        recall = 0
    return recall

# Definir el número de pliegues
k = 30

puntajes_precision = []
puntajes_accuracy = []
puntajes_recall = []
epochs_completadas = []

# División de los datos en los pliegues
sss = StratifiedShuffleSplit(n_splits = k, test_size = 0.25, random_state =
42)

# Iniciar el cronómetro
tiempo_inicial = time.time()

# Iterar a través de cada pliegue
for fold, (train_index, test_index) in enumerate(sss.split(x, y)):
    print(f"Pliegue {fold + 1}/{k}")

    # Datos para cada pliegue
    x_train, x_test = x[train_index], x[test_index]
    y_train, y_test = y[train_index], y[test_index]

    # Preparar los datos de entrada
    x_train = x_train.reshape(x_train.shape[0], x_train.shape[1], 1)
    x_test = x_test.reshape(x_test.shape[0], x_test.shape[1], 1)

    # Iniciar el modelo CNN
    model = Sequential()
    model.add(Conv1D(32, 3, activation='relu', input_shape=(x_train.sha-
pe[1], 1)))
    model.add(MaxPooling1D(2))
    model.add(Conv1D(64, 3, activation='relu'))
    model.add(MaxPooling1D(2))
    model.add(Flatten())
    model.add(Dense(128, activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(y.shape[1], activation='softmax'))

    # Compilar el modelo
    model.compile(loss='categorical_crossentropy', optimizer = tf.kera-
s.optimizers.Adam(learning_rate=0.001), metrics=[calc_precision,
calc_accuracy, calc_recall])

    # Definir callback de EarlyStopping
    early_stopping = EarlyStopping(monitor='calc_precision', patience=2,
restore_best_weights=True)

    # Entrenamiento del modelo
    CNN = model.fit(x_train, y_train, validation_data=(x_test, y_test), ba-
tch_size=1024, epochs=5, callbacks=[early_stopping])
```

```
    # Número de épocas completadas en cada pliegue
    num_epochs = len(CNN.history['calc_precision'])
    epochs_completadas.append(num_epochs)
    print(f"Épocas completadas en el pliegue {fold + 1}: {num_epochs}")

    # Calcular el promedio de la precisión y guardar el puntaje de cada
pliegue
    precision_pliegue = model.evaluate(x_test, y_test, verbose=0)[1]
    puntajes_precision.append(precision_pliegue)

    np.save('datos/puntajes_precision_cnn.npy', puntajes_precision)

    # Calcular el promedio de la exactitud y guardar el puntaje de cada
pliegue
    accuracy_pliegue = model.evaluate(x_test, y_test, verbose=0)[2]
    puntajes_accuracy.append(accuracy_pliegue)

    np.save('datos/puntajes_accuracy_cnn.npy', puntajes_accuracy)

    # Calcular el promedio de la exhaustividad y guardar el puntaje de ca-
da pliegue
    recall_pliegue = model.evaluate(x_test, y_test, verbose=0)[3]
    puntajes_recall.append(recall_pliegue)

    np.save('datos/puntajes_recall_cnn.npy', puntajes_recall)

# Detener el cronómetro
tiempo_final = time.time()

# Función para mostrar el tiempo en horas, minutos y segundos
def time_format(seconds):
    minutes, seconds = divmod(seconds, 60)
    hours, minutes = divmod(minutes, 60)
    return hours, minutes, seconds

# Guardar el modelo
model.save('modelos/CNN.h5')

# Función para mostrar el tiempo en horas, minutos y segundos
def time_format(seconds):
    minutes, seconds = divmod(seconds, 60)
    hours, minutes = divmod(minutes, 60)
    return hours, minutes, seconds

# Calcular el tiempo total
tiempo_total = tiempo_final - tiempo_inicial
hours, minutes, seconds = time_format(tiempo_total)
print(f"Tiempo total del entrenamiento: {int(hours)} horas, {int(minutes)}
minutos y {int(seconds)} segundos")

# Iniciar el cronómetro
tiempo_inicial = time.time()

# Realizar predicciones con el modelo
y_pred = model.predict(x_test)
y_pred_classes = np.argmax(y_pred, axis=1)
y_test_classes = np.argmax(y_test, axis=1)

# Evaluación de las predicciones
accuracy_pred = calc_accuracy(y_test, y_pred)
precision_pred = calc_precision(y_test, y_pred)
recall_pred = calc_recall(y_test, y_pred)

# Resultados
print(f"Exactitud de las predicciones: {accuracy_pred:.4f}")
print(f"Precisión de las predicciones: {precision_pred:.4f}")
print(f"Exhaustividad de las predicciones: {recall_pred:.4f}")

# Detener el cronómetro
tiempo_final = time.time()

# Calcular el tiempo total
tiempo_total = tiempo_final - tiempo_inicial
hours, minutes, seconds = time_format(tiempo_total)
print(f"Tiempo total de las predicciones: {int(hours)} horas, {int(minutes)}
minutos y {int(seconds)} segundos")

# Generar una matriz de confusión con etiquetas
def confusion_matrix_labels(y_test, y_pred):
    cm = confusion_matrix(y_test, y_pred)
    cm_df = pd.DataFrame(cm)

    labels = ['ARP spoofing', 'DoS', 'Normal', 'Port scan']

    color = "#8c6bb1"

    custom_palette = sns.light_palette(color, as_cmap=True, reverse=Fal-
se)

    plt.figure(figsize=(20, 15))
    sns.set(font_scale=1.4)

    sns.heatmap(cm_df, annot=True, annot_kws={"size": 12}, fmt="g", xti-
cklabels=labels, yticklabels=labels, cmap=custom_palette)
    plt.ylabel("Clases reales")
    plt.xlabel("Clases predecidas")

    # Guardar la matriz de confusión
    plt.savefig('img/Matriz-CNN.png')

    plt.show()

confusion_matrix_labels(y_test_classes, y_pred_classes)
```

ANEXO F: ENTRENAMIENTO DE LA RED NEURONAL RECURRENTE

```
import time
import warnings
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import tensorflow as tf
from sklearn.metrics import confusion_matrix
from tensorflow.keras import backend as K
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Dropout
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.optimizers import Adam
from sklearn.model_selection import StratifiedShuffleSplit

# Ignorar advertencias
warnings.filterwarnings("ignore")

# Cargar dataset
df = pd.read_csv('dataset/network-traffic-dataset-shuffled.csv',
header=None, low_memory=False)

# Asignar los nombres de las columnas
df.columns = ['Delta time', 'Protocol', 'Length', 'Src port', 'Dst port', 'Flags',
'Severity',
              'Window size', 'Label', 'Duplicate detected', 'SameNet src add',
'SameNet dst add']

df = df.drop(0)
df = df.drop_duplicates()
df = df.dropna(axis=1, how='all')

# Convertir las columnas a los tipos de datos especificados
df['Delta time'] = df['Delta time'].astype('float32')
df['Length'] = df['Length'].astype('int32')
df['Src port'] = df['Src port'].astype('int32')
df['Dst port'] = df['Dst port'].astype('int32')
df['Window size'] = df['Window size'].astype('int32')
df['Duplicate detected'] = df['Duplicate detected'].astype('int32')
df['SameNet src add'] = df['SameNet src add'].astype('int32')
df['SameNet dst add'] = df['SameNet dst add'].astype('int32')

# Ingeniería de características

# Codificar los valores de texto con One Hot Encoding y variables ficti-
cias
def encode_text(df, name):
    dummies = pd.get_dummies(df[name])
    for i in dummies.columns:
        dummy_name = f"{name}-{i}"
        df[dummy_name] = dummies[i]
    df.drop(name, axis = 1, inplace = True)

# Codificar todos los valores según corresponda
encode_text(df, 'Protocol')
encode_text(df, 'Flags')
encode_text(df, 'Severity')

# Extraer la columna de características y definir variables
x_columns = df.columns.drop('Label')
x = df[x_columns].values

# Codificar los valores de texto con variables ficticias
outcomes = pd.get_dummies(df['Label'])
y = outcomes.values

# Mostras los nombres de las columnas
print(df.columns.tolist())

# Definir funciones para calcular la exactitud, precisión y exhaustividad
def calc_accuracy(y_true, y_pred):
    TP = K.sum(K.round(K.clip(y_true * y_pred, 0, 1)))
    TN = K.sum(K.round(K.clip((1 - y_true) * (1 - y_pred), 0, 1)))
    FP = K.sum(K.round(K.clip((1 - y_true) * y_pred, 0, 1)))
    FN = K.sum(K.round(K.clip(y_true * (1 - y_pred), 0, 1)))

    try:
        accuracy = (TP + TN) / (TP + TN + FP + FN + K.epsilon())
    except ZeroDivisionError:
        accuracy = 0
    return accuracy

def calc_precision(y_true, y_pred):
    TP = K.sum(K.round(K.clip(y_true * y_pred, 0, 1)))
    FP = K.sum(K.round(K.clip((1 - y_true) * y_pred, 0, 1)))

    try:
        precision = TP / (TP + FP + K.epsilon())
    except ZeroDivisionError:
        precision = 0
    return precision

def calc_recall(y_true, y_pred):
    TP = K.sum(K.round(K.clip(y_true * y_pred, 0, 1)))
    FN = K.sum(K.round(K.clip(y_true * (1 - y_pred), 0, 1)))

    try:
        recall = TP / (TP + FN + K.epsilon())
    except ZeroDivisionError:
        recall = 0
    return recall

# Definir el número de pliegues
k = 30

puntajes_precision = []
puntajes_accuracy = []
puntajes_recall = []
epochs_completadas = []

# División de los datos en los pliegues
sss = StratifiedShuffleSplit(n_splits = k, test_size = 0.25, random_state =
42)

# Iniciar el cronómetro
tiempo_inicial = time.time()

# Iterar a través de cada pliegue
for fold, (train_index, test_index) in enumerate(sss.split(x, y)):
    print(f"Pliegue {fold + 1}/{k}")

    # Datos para cada pliegue
    x_train, x_test = x[train_index], x[test_index]
    y_train, y_test = y[train_index], y[test_index]

    # Preparar los datos de entrada
    x_train = np.reshape(x_train, (x_train.shape[0], 1, x_train.shape[1]))
    x_test = np.reshape(x_test, (x_test.shape[0], 1, x_test.shape[1]))

   # Iniciar el modelo RNN
    model = Sequential()
    model.add(LSTM(64, activation='tanh', input_shape=(1,
x_train.shape[2]), return_sequences=True))
    model.add(LSTM(64, activation='tanh'))
    model.add(Dropout(0.5))
    model.add(Dense(y.shape[1], activation='softmax'))

    # Compilar el modelo
    model.compile(loss='categorical_crossentropy', optimizer = tf.kera-
s.optimizers.Adam(learning_rate=0.001), metrics=[calc_precision,
calc_accuracy, calc_recall])

    # Definir callback de EarlyStopping
    early_stopping = EarlyStopping(monitor='calc_precision', patience=2,
restore_best_weights=True)

    # Entrenamiento del modelo
    RNN = model.fit(x_train, y_train, validation_data=(x_test, y_test), ba-
tch_size=1024, epochs=5, callbacks=[early_stopping])

    # Número de épocas completadas en cada pliegue
    num_epochs = len(RNN.history['calc_precision'])
    epochs_completadas.append(num_epochs)
    print(f"Épocas completadas en el pliegue {fold + 1}: {num_epochs}")
```

```
    # Calcular el promedio de la precisión y guardar el puntaje de cada
pliegue
    precision_pliegue = model.evaluate(x_test, y_test, verbose=0)[1]
    puntajes_precision.append(precision_pliegue)

    np.save('datos/puntajes_precision_rnn.npy', puntajes_precision)

    # Calcular el promedio de la exactitud y guardar el puntaje de cada
pliegue
    accuracy_pliegue = model.evaluate(x_test, y_test, verbose=0)[2]
    puntajes_accuracy.append(accuracy_pliegue)

    np.save('datos/puntajes_accuracy_rnn.npy', puntajes_accuracy)

    # Calcular el promedio de la exhaustividad y guardar el puntaje de ca-
da pliegue
    recall_pliegue = model.evaluate(x_test, y_test, verbose=0)[3]
    puntajes_recall.append(recall_pliegue)

    np.save('datos/puntajes_recall_rnn.npy', puntajes_recall)

# Detener el cronómetro
tiempo_final = time.time()

# Función para mostrar el tiempo en horas, minutos y segundos
def time_format(seconds):
    minutes, seconds = divmod(seconds, 60)
    hours, minutes = divmod(minutes, 60)
    return hours, minutes, seconds

# Calcular el tiempo total
tiempo_total = tiempo_final - tiempo_inicial
hours, minutes, seconds = time_format(tiempo_total)
print(f"Tiempo total del entrenamiento: {int(hours)} horas, {int(minutes)}
minutos y {int(seconds)} segundos")

# Guardar el modelo
model.save('modelos/RNN.h5')

# Iniciar el cronómetro
tiempo_inicial = time.time()

# Realizar predicciones con el modelo
y_pred = model.predict(x_test)
y_pred_classes = np.argmax(y_pred, axis=1)
y_test_classes = np.argmax(y_test, axis=1)

# Evaluación de las predicciones
accuracy_pred = calc_accuracy(y_test, y_pred)
precision_pred = calc_precision(y_test, y_pred)
recall_pred = calc_recall(y_test, y_pred)

# Resultados
print(f"Exactitud de las predicciones: {accuracy_pred:.4f}")
print(f"Precisión de las predicciones: {precision_pred:.4f}")
print(f"Exhaustividad de las predicciones: {recall_pred:.4f}")

# Detener el cronómetro
tiempo_final = time.time()

# Calcular el tiempo total
tiempo_total = tiempo_final - tiempo_inicial
hours, minutes, seconds = time_format(tiempo_total)
print(f"Tiempo total de las predicciones: {int(hours)} horas, {int(minutes)}
minutos y {int(seconds)} segundos")

# Generar una matriz de confusión con etiquetas
def confusion_matrix_labels(y_test, y_pred):
    cm = confusion_matrix(y_test, y_pred)
    cm_df = pd.DataFrame(cm)

    labels = ['ARP spoofing', 'DoS', 'Normal', 'Port scan']

    color = "#a2a7c8"

    custom_palette = sns.light_palette(color, as_cmap=True, reverse=Fal-
se)

    plt.figure(figsize=(20, 15))
    sns.set(font_scale=1.4)

    sns.heatmap(cm_df, annot=True, annot_kws={"size": 12}, fmt="g", xti-
cklabels=labels, yticklabels=labels, cmap=custom_palette)
    plt.ylabel("Clases reales")
    plt.xlabel("Clases predecidas")

    # Guardar la matriz de confusión
    plt.savefig('img/Matriz-RNN.png')

    plt.show()

confusion_matrix_labels(y_test_classes, y_pred_classes)
```

ANEXO G: ENTRENAMIENTO DE LA RED NEURONAL HÍBRIDA CONVOLUCIONAL RECURRENTE

```
import time
import warnings
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import tensorflow as tf
from sklearn.metrics import confusion_matrix
from tensorflow.keras import backend as K
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv1D, MaxPooling1D, LSTM, Dense, Dropout
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.optimizers import Adam
from sklearn.model_selection import StratifiedShuffleSplit

# Ignorar advertencias
warnings.filterwarnings("ignore")
# Cargar dataset
df = pd.read_csv('dataset/network-traffic-dataset-shuffled.csv', header=None, low_memory=False)

# Asignar los nombres de las columnas
df.columns = ['Delta time', 'Protocol', 'Length', 'Src port', 'Dst port', 'Flags', 'Severity',
              'Window size', 'Label', 'Duplicate detected', 'SameNet src add', 'SameNet dst add']

df = df.drop(0)
df = df.drop_duplicates()
df = df.dropna(axis=1, how='all')

# Convertir las columnas a los tipos de datos especificados
df['Delta time'] = df['Delta time'].astype('float32')
df['Length'] = df['Length'].astype('int32')
df['Src port'] = df['Src port'].astype('int32')
df['Dst port'] = df['Dst port'].astype('int32')
df['Window size'] = df['Window size'].astype('int32')
df['Duplicate detected'] = df['Duplicate detected'].astype('int32')
df['SameNet src add'] = df['SameNet src add'].astype('int32')
df['SameNet dst add'] = df['SameNet dst add'].astype('int32')

# Ingeniería de características

# Codificar los valores de texto con One Hot Encoding y variables ficticias
def encode_text(df, name):
    dummies = pd.get_dummies(df[name])
    for i in dummies.columns:
        dummy_name = f"{name}-{i}"
        df[dummy_name] = dummies[i]
    df.drop(name, axis = 1, inplace = True)


# Codificar todos los valores según corresponda
encode_text(df, 'Protocol')
encode_text(df, 'Flags')
encode_text(df, 'Severity')

# Extraer la columna de características y definir variables
x_columns = df.columns.drop('Label')
x = df[x_columns].values

# Codificar los valores de texto con variables ficticias
outcomes = pd.get_dummies(df['Label'])
y = outcomes.values

# Mostras los nombres de las columnas
print(df.columns.tolist())

# Definir funciones para calcular la exactitud, precisión y exhaustividad
def calc_accuracy(y_true, y_pred):
    TP = K.sum(K.round(K.clip(y_true * y_pred, 0, 1)))
    TN = K.sum(K.round(K.clip((1 - y_true) * (1 - y_pred), 0, 1)))
    FP = K.sum(K.round(K.clip((1 - y_true) * y_pred, 0, 1)))
    FN = K.sum(K.round(K.clip(y_true * (1 - y_pred), 0, 1)))

    try:
        accuracy = (TP + TN) / (TP + TN + FP + FN + K.epsilon())
    except ZeroDivisionError:
        accuracy = 0
    return accuracy

def calc_precision(y_true, y_pred):
    TP = K.sum(K.round(K.clip(y_true * y_pred, 0, 1)))
    FP = K.sum(K.round(K.clip((1 - y_true) * y_pred, 0, 1)))

    try:
        precision = TP / (TP + FP + K.epsilon())
    except ZeroDivisionError:
        precision = 0
    return precision

def calc_recall(y_true, y_pred):
    TP = K.sum(K.round(K.clip(y_true * y_pred, 0, 1)))
    FN = K.sum(K.round(K.clip(y_true * (1 - y_pred), 0, 1)))

    try:
        recall = TP / (TP + FN + K.epsilon())
    except ZeroDivisionError:
        recall = 0
    return recall

# Definir el número de pliegues
k = 30

puntajes_precision = []
puntajes_accuracy = []
puntajes_recall = []
epochs_completadas = []

# División de los datos en los pliegues
sss = StratifiedShuffleSplit(n_splits = k, test_size = 0.25, random_state = 42)

# Iniciar el cronómetro
tiempo_inicial = time.time()

# Iterar a través de cada pliegue
for fold, (train_index, test_index) in enumerate(sss.split(x, y)):
    print(f"Pliegue {fold + 1}/{k}")

    # Datos para cada pliegue
    x_train, x_test = x[train_index], x[test_index]
    y_train, y_test = y[train_index], y[test_index]

    # Preparar los datos de entrada
    x_train = x_train.reshape(x_train.shape[0], x_train.shape[1], 1)
    x_test = x_test.reshape(x_test.shape[0], x_test.shape[1], 1)

    # Iniciar el modelo híbrido CNN-RNN
    model = Sequential()
    model.add(Conv1D(32, 3, activation='relu', input_shape=(x_train.shape[1], 1)))
    model.add(MaxPooling1D(2))
    model.add(Conv1D(64, 3, activation='relu'))
    model.add(MaxPooling1D(2))
    model.add(LSTM(64, return_sequences=True, activation='tanh'))
    model.add(LSTM(64, activation='tanh'))
    model.add(Dense(128, activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(y.shape[1], activation='softmax'))

    # Compilar el modelo
    model.compile(loss='categorical_crossentropy', optimizer = tf.keras.optimizers.Adam(learning_rate=0.001), metrics=[calc_precision, calc_accuracy, calc_recall])

    # Definir callback de EarlyStopping
```

```
    early_stopping = EarlyStopping(monitor='calc_precision', patience=2, restore_best_weights=True)

    # Entrenamiento del modelo
    CNN_RNN = model.fit(x_train, y_train, validation_data=(x_test, y_test), batch_size=1024, epochs=5, callbacks=[early_stopping])

    # Número de épocas completadas en cada pliegue
    num_epochs = len(CNN_RNN.history['calc_precision'])
    epochs_completadas.append(num_epochs)
    print(f"Épocas completadas en el pliegue {fold + 1}: {num_epochs}")

    # Calcular el promedio de la precisión y guardar el puntaje de cada pliegue
    precision_pliegue = model.evaluate(x_test, y_test, verbose=0)[1]
    puntajes_precision.append(precision_pliegue)

    np.save('datos/puntajes_precision_cnn_rnn.npy', puntajes_precision)

    # Calcular el promedio de la exactitud y guardar el puntaje de cada pliegue
    accuracy_pliegue = model.evaluate(x_test, y_test, verbose=0)[2]
    puntajes_accuracy.append(accuracy_pliegue)

    np.save('datos/puntajes_accuracy_cnn_rnn.npy', puntajes_accuracy)

    # Calcular el promedio de la exhaustividad y guardar el puntaje de cada pliegue
    recall_pliegue = model.evaluate(x_test, y_test, verbose=0)[3]
    puntajes_recall.append(recall_pliegue)

    np.save('datos/puntajes_recall_cnn_rnn.npy', puntajes_recall)

# Detener el cronómetro
tiempo_final = time.time()

# Función para mostrar el tiempo en horas, minutos y segundos
def time_format(seconds):
    minutes, seconds = divmod(seconds, 60)
    hours, minutes = divmod(minutes, 60)
    return hours, minutes, seconds

# Calcular el tiempo total
tiempo_total = tiempo_final - tiempo_inicial
hours, minutes, seconds = time_format(tiempo_total)
print(f"Tiempo total del entrenamiento: {int(hours)} horas, {int(minutes)} minutos y {int(seconds)} segundos")

# Guardar el modelo
model.save('modelos/CNN-RNN.h5')

# Iniciar el cronómetro
tiempo_inicial = time.time()

# Realizar predicciones con el modelo
y_pred = model.predict(x_test)
y_pred_classes = np.argmax(y_pred, axis=1)
y_test_classes = np.argmax(y_test, axis=1)

# Evaluación de las predicciones
accuracy_pred = calc_accuracy(y_test, y_pred)
precision_pred = calc_precision(y_test, y_pred)
recall_pred = calc_recall(y_test, y_pred)

# Resultados
print(f"Exactitud de las predicciones: {accuracy_pred:.4f}")
print(f"Precisión de las predicciones: {precision_pred:.4f}")
print(f"Exhaustividad de las predicciones: {recall_pred:.4f}")

# Detener el cronómetro
tiempo_final = time.time()

# Calcular el tiempo total
tiempo_total = tiempo_final - tiempo_inicial
hours, minutes, seconds = time_format(tiempo_total)
print(f"Tiempo total de las predicciones: {int(hours)} horas, {int(minutes)} minutos y {int(seconds)} segundos")

# Generar una matriz de confusión con etiquetas
def confusion_matrix_labels(y_test, y_pred):
    cm = confusion_matrix(y_test, y_pred)
    cm_df = pd.DataFrame(cm)

    labels = ['ARP spoofing', 'DoS', 'Normal', 'Port scan']

    color = "#b58ec9"

    custom_palette = sns.light_palette(color, as_cmap=True, reverse=False)

    plt.figure(figsize=(20, 15))
    sns.set(font_scale=1.4)

    sns.heatmap(cm_df, annot=True, annot_kws={"size": 12}, fmt="g", xticklabels=labels, yticklabels=labels, cmap=custom_palette)
    plt.ylabel("Clases reales")
    plt.xlabel("Clases predecidas")

    # Guardar la matriz de confusión
    plt.savefig('img/Matriz-CNN-RNN.png')

    plt.show()

confusion_matrix_labels(y_test_classes, y_pred_classes)
```

ANEXO H: DEMOSTRACIÓN DE LA HIPÓTESIS

```
Importar las librerías
import pandas as pd
import numpy as np
from scipy import stats

Cargar los datos de los modelos
CNN = np.load('datos/puntajes_precision_cnn.npy')
print(CNN)

RNN = np.load('datos/puntajes_precision_rnn.npy')
print(RNN)

CNN_RNN = np.load('datos/puntajes_precision_cnn_rnn.npy')
print(CNN_RNN)

[0.99104953 0.99176919 0.99307603 0.9929266  0.99002129
0.99079078
 0.98955303 0.99140906 0.99260837 0.99344671 0.99359053
0.98399633
 0.99246198 0.99137461 0.99018413 0.99013996 0.98984265
0.98976415
 0.99062663 0.98959106 0.99103421 0.99289703 0.99095827
0.99241292
 0.98935926 0.99201572 0.99220216 0.98856288 0.99247473
0.99104786]
[0.94147384 0.99333078 0.95628524 0.94798744 0.9927749
0.99355155
 0.94349581 0.94129711 0.99176008 0.99230307 0.99189997
0.9924925
 0.94275057 0.9916842  0.99246746 0.98961163 0.99154907
0.94369006
 0.9922387  0.99178135 0.99155313 0.99103051 0.94804782
0.99169844
 0.99187845 0.99118626 0.99128795 0.99024254 0.94069409
0.99137121]
[0.97370392 0.99436128 0.97041428 0.98795015 0.97518027
0.97267157
 0.95216054 0.99405199 0.98668921 0.97408259 0.96941394
0.98547077
 0.97178018 0.98226589 0.9665252  0.98898888 0.97154564
0.98325884
 0.99490666 0.96390867 0.97969431 0.97148645 0.96498471
0.9891969
 0.98998421 0.95923293 0.98646247 0.99455816 0.98973507
0.99394143]

Definir el valor objetivo y el nivel de significancia
valor_objetivo = 0.95
alpha = 0.05

Análisis de varianza factorial ANOVA de una vía
f_statistic, p_value = stats.f_oneway(CNN, RNN, CNN_RNN)
print("Estadístico F:", f_statistic)
print("p-value:", p_value)

if p_value < alpha:
    print("Se rechaza la hipótesis nula. Existe una diferencia significativa
entre los modelos.")
else:
    print("No se rechaza la hipótesis nula. No existe una diferencia signifi-
cativa entre los modelos.")

Estadístico F: 7.607890988243418
p-value: 0.0009016882843702106
Se rechaza la hipótesis nula. Existe una diferencia significativa entre los
modelos.

Prueba de t de Student para cada modelo
for modelo, datos in zip(['CNN', 'RNN', 'CNN_RNN'], [CNN, RNN,
CNN_RNN]):
    if len(datos) < 2:
        print(f"\nModelo de red neuronal: {modelo}")
        print("Error: No se puede realizar la prueba t. La longitud de los da
                tos es menor que 2.")
        continue

    t_statistic, p_value = stats.ttest_1samp(datos, valor_objetivo)

    print(f"Modelo de red neuronal: {modelo}")
    print("Estadístico t:", t_statistic)
    print("p-value:", p_value)

    if p_value < alpha:
        print(f"Se rechaza la hipótesis nula. La precisión media es significa-
tivamente superior al {int(valor_objetivo*100)}%.")
    else:
        print(f"No se rechaza la hipótesis nula. La precisión media no es
significativamente superior al {int(valor_objetivo*100)}%.")

Modelo de red neuronal: CNN
Estadístico t: 118.85605738114067
p-value: 1.5263463305159994e-40
Se rechaza la hipótesis nula. La precisión media es significativamente
superior al 95%.

Modelo de red neuronal: RNN
Estadístico t: 6.934172211249392
p-value: 1.2743148229190142e-07
Se rechaza la hipótesis nula. La precisión media es significativamente
superior al 95%.

Modelo de red neuronal: CNN_RNN
Estadístico t: 13.638700512303219
p-value: 3.8045040322259805e-14
Se rechaza la hipótesis nula. La precisión media es significativamente
superior al 95%.
```