UNIVERSIDAD DE BURGOS
ESCUELA POLITÉCNICA SUPERIOR
Grado en Ingeniería Informática

TFG del Grado en Ingeniería Informática

Transformers para la Clasificación de Tráfico IoT en Ventanas Temporales: Un enfoque basado en NLP

Presentado por Sergio Martín Reizábal
en Universidad de Burgos — 5 de julio de 2025
Tutor: Rubén Ruiz González
Co-tutor: Nuño Basurto Hornillos

UNIVERSIDAD DE BURGOS
ESCUELA POLITÉCNICA SUPERIOR
Grado en Ingeniería Informática

D. Rubén Ruiz González, Profesor Ayudante Doctor del Departamento de Digitalización, área de Ingeniería de Sistemas y Automática.

D. Nuño Basurto Hornillos, Profesor Ayudante Doctor del Departamento de Digitalización, área de Ciencia de la Computación e Inteligencia Artificial.

Exponen:

Que el alumno D. Sergio Martín Reizábal, con DNI 71306412H, ha realizado el Trabajo Fin de Grado en Ingeniería Informática titulado "Transformers para la Clasificación de Tráfico IoT en Ventanas Temporales: Un enfoque basado en NLP".

Y que dicho trabajo ha sido realizado por el alumno bajo la dirección de quienes suscriben, en virtud de lo cual se autoriza su presentación y defensa.

En Burgos, 5 de julio de 2025

Vº. Bº. del Tutor:	Vº. Bº. del co-tutor:
D. Rubén Ruiz González	D. Nuño Basurto Hornillos

Resumen

El crecimiento masivo de dispositivos IoT ha convertido la monitorización del tráfico de red en un reto urgente, especialmente para sistemas de detección de intrusiones (IDS) que suelen situarse en gateways, routers o nodos de borde con recursos limitados. Estos dispositivos deben ser capaces de identificar patrones anómalos en tiempo real sin depender de reglas estáticas ni consumir grandes cantidades de cómputo.

Este Trabajo Fin de Grado presenta un IDS ligero basado en Transformers, diseñado específicamente para ser embebido en un router dentro de una red local IoT. A partir de capturas PCAP del dataset CICIoT2023, se extraen flujos con CICFlowMeter, que se agrupan en ventanas temporales de 5 segundos y se transforman en tensores uniformes que alimentan un modelo Transformer implementado en PyTorch. Los resultados experimentales muestran una precisión general del 98,83 % y un F1 ponderado del 98,85 %, con métricas destacadas como una precisión del 99,88 % en tráfico benigno y *recall* del 98,68 % para ataques de tipo PortScan. La eficacia del sistema y su baja latencia lo hacen adecuado para proteger redes locales en tiempo real sin necesidad de hardware adicional ni conexión a la nube.

Descriptores

IoT, clasificación de tráfico, CICIoT2023, Transformers, CICFlowMeter, ventanas temporales, tiempo real.

Abstract

The massive growth of IoT devices has turned network traffic monitoring into an urgent challenge, especially for intrusion detection systems (IDS) typically deployed at gateways, routers, or edge nodes with limited computational resources. These systems must detect anomalous patterns in real time without relying on static rules or consuming excessive processing power.

This Bachelor's Thesis presents a lightweight IDS based on Transformers, specifically designed to be embedded in a router within a local IoT network. Starting from PCAP captures from the CICIoT2023 dataset, individual flows are extracted using CICFlowMeter, grouped into fixed 5-second time windows, and transformed into uniform tensors fed into a Transformer model implemented in PyTorch. Experimental results show an overall accuracy of 98.83 % and a weighted F1-score of 98.85 %, with notable metrics such as 99.88 % precision on benign traffic and 98.68 % recall for PortScan attacks. The system's efficiency and low latency make it suitable for real-time protection of local networks without requiring additional hardware or cloud connectivity.

Keywords

IoT, traffic classification, CICIoT2023, Transformers, CICFlowMeter, temporal windows, real-time.

Índice general

Índice de figuras

Índice de tablas

1. Introducción

La heterogeneidad de los dispositivos IoT (*Internet of Things*) ha propiciado en los últimos años un crecimiento explosivo del tráfico de red. Este tráfico presenta características únicas derivadas de la diversidad de dispositivos, protocolos y patrones de comunicación, lo que dificulta la aplicación de métodos tradicionales de detección y clasificación de amenazas [19]. Se estima que el número de dispositivos IoT en todo el mundo alcanzará los 40 600 millones en 2034, lo que supone más del doble respecto a los 19 800 millones previstos para 2025 [6], muchos de ellos funcionando con firmware básico y muy limitado, recursos de cómputo escasos y seguridad casi inexistente [19]. Esta realidad supone un reto para los IDS (*Intrusion Detection Systems*): las técnicas clásicas de monitorización, basadas en firmas o reglas estáticas, carecen de la flexibilidad necesaria para detectar comportamientos novedosos y, al mismo tiempo, suelen requerir una capacidad de procesamiento que excede la disponible en entornos IoT. Por este motivo, los IDS se implementan habitualmente en gateways, routers o nodos de borde, donde el tráfico puede ser monitorizado sin comprometer los recursos limitados de los dispositivos finales [21].

En el ámbito específico de la **detección de intrusiones en entornos IoT**, la clasificación de tráfico en tiempo real constituye la primera línea de defensa. Tradicionalmente se ha abordado este problema desde dos frentes complementarios. Por un lado, la literatura recoge aproximaciones estadísticas y de *machine learning* "ligeras" (SVM, Random Forest, k–NN) que, pese a su bajo consumo de recursos, procesan los flujos de forma aislada y adolecen de esa capacidad para capturar dependencias temporales [27]. Por otro lado, las soluciones basadas en aprendizaje profundo (CNN, LSTM, GRU) elevan el nivel de acierto, pero conllevan costes de entrenamiento e

inferencia poco asumibles para ejecutarse *in situ* en pasarelas domésticas o *edge routers* [31].

Los **Transformers** han irrumpido recientemente como una alternativa capaz de combinar precisión y paralelismo [22]. Sin embargo, su aplicación a la seguridad de redes IoT permanece todavía poco explorada. Las propuestas existentes se centran, en su mayoría, en paquetes o flujos individuales [28]. Además, apenas se ha estudiado la granularidad óptima de análisis ni la manera de representar tráfico IoT como una secuencia homogénea manejable por el mecanismo de auto-atención [22].

Contribución de este trabajo

Este Trabajo Fin de Grado introduce una arquitectura completa de detección y clasificación de tráfico IoT basada en un *encoder* Transformer ligero. Las aportaciones son las siguientes:

1. **Modelado temporal del tráfico**. Se propone la analogía *tráfico-como-lenguaje*, donde cada ventana fija de 5 s se interpreta como una "frase" y cada flujo bidireccional como una "palabra", lo que permite explotar directamente el mecanismo de auto-atención. Ahora, cada flujo dispone de un contexto **holístico**, en el que puede tener en cuenta el resto de flujos pertenecientes a la misma ventana temporal.

2. **Pipeline de preprocesado reproducible**. A partir del dataset CICIoT2023[25] se construye un procedimiento que extrae flujos con `CICFlowMeter` [23], los agrupa en ventanas consecutivas y genera tensores de tamaño uniforme (128×79) listos para inferencia.

3. **Diseño de un Transformer compacto**. El modelo resultante, con $\approx 0{,}5$ M parámetros, alcanza un **98,85 %** de F_1 ponderado y procesa cada ventana en 0,23 ms, lo que hace viable su despliegue en routers domésticos y otros sistemas embebidos.

4. **Demostrador web de código abierto**. Se desarrolla una API basada en `FastAPI` y un *frontend* interactivo que permite cargar trazas de red, visualizar predicciones en tiempo real y analizar métricas de rendimiento.

Estructura de la memoria

El resto de esta memoria se organiza del siguiente modo:

- **Capítulo 2. Objetivos del proyecto**: define el objetivo general y los objetivos específicos.
- **Capítulo 3. Conceptos teóricos**: resume los fundamentos de tráfico IoT, flujos de red, ventanas temporales, aprendizaje profundo y Transformers.
- **Capítulo 4. Técnicas y herramientas**: repasa el software, librerías y conjuntos de datos empleados.
- **Capítulo 5. Desarrollo del proyecto**: describe el pipeline de datos, la implementación y evaluación del modelo y el despliegue de la aplicación.
- **Capítulo 7. Conclusiones y líneas futuras**: resume las aportaciones y propone mejoras para trabajos posteriores.

Todo el código fuente, así como los recursos desarrollados durante el proyecto, están disponibles públicamente en el siguiente repositorio de GitHub:

https://github.com/SergioMartinReizabal/TFG-GII-Transformer-Series-Temporales-Trafico-IoT

2. Objetivos del proyecto

En este trabajo se propone desarrollar y desplegar un sistema para detectar y clasificar ataques en tiempo real sobre tráfico IoT mediante técnicas de procesamiento del lenguaje natural (NLP). El enfoque principal consiste en analizar el tráfico generado por dispositivos IoT utilizando una analogía con el procesamiento del lenguaje, describiendo ventanas temporales como secuencias y flujos como elementos dentro de ellas.

Objetivo general

Diseñar e implementar un sistema integral que, partiendo de capturas de tráfico en formato PCAP, clasifique automáticamente el tráfico IoT en tiempo real mediante un modelo Transformer propio.

Objetivos específicos

Investigación y desarrollo científico

- Extraer flujos individuales a partir de capturas PCAP utilizando la herramienta CICFlowMeter [23].
- Proponer un método eficiente para agrupar estos flujos en ventanas temporales fijas de 5 segundos.
- Diseñar una metodología para transformar cada ventana en tensores de tamaño fijo $[N, 128, 79]$, incluyendo técnicas como padding y truncamiento.

- Diseñar, entrenar y evaluar un modelo Transformer ligero capaz de clasificar las ventanas en tipos específicos de tráfico, incluyendo la detección de ataques.

Objetivos funcionales

- Facilitar la carga interactiva de archivos CSV generados por CICFlowMeter a través de una interfaz web.
- Mostrar resultados claros y dinámicos mediante tablas y gráficos interactivos cuando existan etiquetas.
- Presentar métricas clave de rendimiento (accuracy, F1 score ponderado) en los casos donde las etiquetas reales estén disponibles.

Objetivos técnicos

- Desarrollar un backend robusto y eficiente en FastAPI para procesar y clasificar datos en tiempo real.
- Crear un frontend responsivo con tecnologías modernas como Bootstrap [5], DataTables [8] y Plotly [11].
- Integrar componentes del pipeline en una arquitectura modular que permita mantenimiento y futuras extensiones.
- Documentar claramente cada componente para facilitar posteriores adaptaciones y mejoras.

3. Conceptos teóricos

Este capítulo presenta los fundamentos teóricos necesarios para comprender el enfoque propuesto en este proyecto. Se abordan conceptos de redes, tráfico IoT, aprendizaje profundo y modelos de tipo transformer, así como la analogía con el procesamiento del lenguaje natural (NLP).

3.1. Tráfico de red en entornos IoT y sus particularidades

El *Internet de las Cosas* (*IoT*) representa un paradigma en el que multitud de dispositivos heterogéneos (sensores, actuadores, electrodomésticos inteligentes, etc.) se conectan a la red para intercambiar datos sin necesidad de intervención humana directa. Esto produce un tráfico de red con características particulares respecto al tráfico tradicional. En entornos IoT predominan las comunicaciones *machine-to-machine* (M2M) de baja potencia (LPWAN, *Low-Power Wide Area Network*): muchos dispositivos de bajo consumo energético y frecuentemente alimentados por batería envían mensajes pequeños a través de redes con recursos limitados (por ejemplo, enlaces inalámbricos de baja potencia) [24]. Como resultado, el tráfico IoT suele consistir en paquetes de tamaño reducido, transmisiones periódicas o disparadas por eventos, y protocolos ligeros optimizados para entornos con altas tasas de pérdida y baja capacidad (p.ej., MQTT, CoAP), a diferencia del tráfico centrado en usuarios humanos (como streaming de video, navegación web, etc.). Esta diferencia implica que las tasas de datos y patrones de comunicación en IoT suelen ser muy distintos: abundan ráfagas breves o informes periódicos de sensores, en vez de flujos largos y sostenidos. Ade-

más, el número de dispositivos conectados crece exponencialmente, lo que incrementa proporcionalmente el volumen de tráfico generado [24].

3.2. Archivos PCAP

Los archivos **PCAP** (*Packet Capture*) son archivos estándar utilizados para almacenar capturas de paquetes de tráfico de red. Estos archivos contienen información detallada sobre el tráfico que circula por una red, incluyendo cabeceras de paquetes, datos de protocolos y contenido transmitido. PCAP es un formato ampliamente utilizado en análisis forense de redes, auditoría de seguridad, diagnóstico de problemas de red y desarrollo de sistemas de detección de intrusiones (IDS).

Estos archivos suelen generarse utilizando herramientas como *Wireshark* [16] o *tcpdump* [15], que permiten capturar y visualizar en detalle las comunicaciones de red. Cada paquete almacenado incluye información como las direcciones IP de origen y destino, puertos, protocolos involucrados y la carga útil del paquete (*payload*). La utilidad de estos archivos radica en su capacidad para realizar análisis detallados y reproducibles del tráfico de red capturado [7].

3.3. ¿Qué es un flujo de red y cómo se extrae?

En el ámbito del análisis de tráfico, un *flujo de red* (*network flow*) se define comúnmente como un conjunto de paquetes que comparten ciertas propiedades y ocurren dentro de un intervalo temporal determinado. Según la definición de la IETF (Internet Engineering Task Force) [1], un flujo es "un conjunto de paquetes que pasa por un punto de observación de la red durante un intervalo de tiempo, y que comparten un conjunto de propiedades comunes", tales como las direcciones IP de origen y destino, puertos de origen y destino, y el protocolo de nivel de transporte [17]. En otras palabras, todos los paquetes de un flujo pertenecen a una misma comunicación o sesión lógica entre dos extremos.

Para extraer flujos de red a partir de un conjunto de paquetes (por ejemplo, de una traza PCAP), usualmente se utiliza la 5–tupla $\{IP_{origen},$

[1] La IETF es la organización responsable del desarrollo de estándares técnicos como los RFC, que definen muchos de los protocolos y prácticas fundamentales de Internet.

$\text{IP}_{\text{destino}}$, $\text{puerto}_{\text{origen}}$, $\text{puerto}_{\text{destino}}$, protocolo} para identificar el flujo, y posteriormente se definen los siguientes parámetros:

- **Flow timeout** o **Idle timeout**: tiempo tras el cual un flujo se da por terminado si no se observan más paquetes, típico para finalizar flujos UDP.
- **Activity timeout**: tiempo máximo que puede durar un flujo activo, independientemente de si siguen llegando paquetes.

Una herramienta ampliamente utilizada para este propósito es *CIC-FlowMeter*, desarrollada por el *Canadian Institute for Cybersecurity*, la cual genera flujos bidireccionales: considera que el primer paquete visto determina la dirección *forward* (origen→destino), y el sentido inverso constituye el *backward* (destino→origen) [23]. Esta herramienta calcula para cada flujo más de 80 características estadísticas (*features*) relacionadas con el tráfico, tales como la duración del flujo, número de paquetes y bytes transmitidos en cada dirección, tamaños de paquetes mínimo/medio/máximo, intervalos de tiempo entre paquetes, número de paquetes con flags específicas (SYN, ACK, FIN, etc.), entre muchas otras [23].

3.4. Ventanas temporales en el análisis de red

El análisis de tráfico de red a menudo emplea ventanas temporales para segmentar los datos en bloques más manejables a lo largo del tiempo. Una *ventana temporal* es un intervalo de tiempo (de duración fija o variable) durante el cual se recopilan o agregan eventos de red con un futuro propósito analítico. Por ejemplo, en clasificación de tráfico se pueden construir vectores de características que resumen el comportamiendo de un flujo en sub–intervalos temporales.

El uso de ventanas temporales introduce un compromiso importante entre *granularidad temporal* y *estabilidad de las medidas* [30].

- **Ventanas muy cortas (p. ej., unos pocos segundos)** capturan con detalle cambios rápidos en el tráfico y permiten una detección más rápida de eventos anómalos, pero también tienden a ser más sesnsibles al ruido y a variaciones breves, pudiendo producir métricas inestables de un intervalo a otro.

- **Ventanas más grandes (p. ej., varios minutos)** proporcionan estimaciones más estables del comportamiento medio, suavizando fluctuaciones, pero al costo de introducir mayor retraso en la detección de cambios y de mezclar potencialmente comportamientos diferentes dentro de la misma ventana.

En la literatura, ciertos trabajos encontraron que para caracterizar flujos de red, una ventana de 5 minutos (con ventanas deslizantes cada 1 minuto) estabiliza muchas métricas del flujo, mientras que ventanas más cortas muestran alta variabilidad [30].

En resumen, las ventanas temporales actúan como unidades de análisis que transforman un flujo continuo de paquetes en una secuencia de observaciones discretas, lo que facilita la aplicación de algoritmos de aprendizaje o técnicas estadísticas.

Existen dos tipos de ventanas temporales:

- **Ventanas deslizantes (overlapping):** se solapan en el tiempo y ofrecen una visión continua, útil para promedios móviles o detección temprana de cambios.
- **Ventanas consecutivas (no overlapping):** se colocan una tras otra sin solaparse; dan particiones disjuntas del timeline y simplifican el análisis por bloques.

3.5. Analogía entre el procesamiento del lenguaje natural (NLP) y el análisis del tráfico de red

La intuición en la que se basa este proyecto es pensar en el tráfico de red como si fuese un lenguaje. Cada ventana de 5 s se comporta como una "frase" y cada flujo que circula dentro de ella se interpreta como una "palabra". En la Tabla 3.1 se resumen las equivalencias clave:

¿Cómo se translada esta analogía al modelo?

- **Secuencia de entrada.** Para cada ventana se construye una lista con todos los flujos *activos* en ese intervalo, incluso si empezaron en la ventana anterior.

Procesamiento del Lenguaje Natural	**Análisis de Tráfico IoT**
Frase u oración	Ventana temporal de 5 s
Palabra	Flujo individual
Embedding de palabras	Vector de 79 características por flujo
Orden de palabras	Orden cronológico por *timestamp*
Token `[CLS]` de BERT	Flujo virtual que resume la ventana (clasificación)
Padding con `[PAD]`	Flujos nulos añadidos hasta alcanzar 128 posiciones

Tabla 3.1: Analogía conceptual entre NLP y tráfico IoT

- **Token `[CLS]`.** Se añade un flujo virtual al principio; su embedding se alimenta del contexto mediante el mecanismo de auto-atención y, al final de las capas, actúa como vector resumen para clasificar la ventana.

- **Codificación de posición.** Igual que en un texto no es lo mismo decir «*El perro mordió al vigilante*» que «*El vigilante mordió al perro*», en una ventana de tráfico tampoco es igual que veamos primero un *flujo corto de resolución DNS* y después un *flujo HTTPS* que usa esa resolución, que al revés (HTTPS antes del DNS). Para que el modelo distinga *qué ocurrió antes*, se suma a cada embedding un vector posicional derivado del instante de inicio del flujo.

- **Tamaño fijo de secuencia.** El modelo necesita tensores de longitud constante, por lo que fijamos `max_seq_len` = 128 posiciones: 1 reservada al token `[CLS]` y 127 para flujos reales. *Si la ventana trae menos de 127 flujos*, completamos con flujos nulos (*padding*). *Si la ventana supera los 127 flujos*, conservamos los 127 primeros ordenados por *timestamp* y descartamos el resto.

En definitiva, pensar en el «tráfico como lenguaje» simplifica el diseño del modelo y aprovecha el poder de los *transformers* para entender los patrones colectivos de los flujos dentro de cada ventana temporal de la red IoT.

3.6. Fundamentos de aprendizaje profundo y modelos secuenciales

El **aprendizaje profundo** (o *deep learning*) emplea redes neuronales con varias capas que transforman progresivamente los datos brutos en representaciones cada vez más abstractas. El libro de Géron [20] muestra en un ejemplo de tratamiento de imágenes cómo, capa tras capa, las primeras neuronas aprenden patrones simples (p.ej., bordes, formas, texturas) y las últimas combinan esos patrones para reconocer conceptos completos.

Modelos secuenciales clásicos

Cuando los datos presentan un orden temporal (texto, audio, series de tiempo), necesitamos redes que *recuerden* lo visto en paso anteriores. Las redes neuronales recurrentes (**RNN**, *Recurrent Neural Network*) añaden un estado interno que se actualiza paso a paso, capturando dependencias cortas. Sin embargo, las RNN "puras" sufren el problema del *gradiente que se desvanece*, lo que dificulta aprender patrones largos [26].

Para solventarlo aparecieron las celdas con puertas, en especial la **Long Short-Term Memory (LSTM)**, que introduce mecanismos de entrada, olvido y salida. Estas puertas controla que información se guarda, cual se descarta y qué parte del estado interno se expone en cada caso [20]. Existe también la variante GRU (*Gated Recurrent Unit*), algo más ligera pero basada en la misma idea.

Por qué los Transformers han tomado el relevo

Las LSTM y las GRU solucionaron gran parte de las limitaciones de las RNN, especialmente en la captura de dependencias a largo plazo. Sin embargo, su naturaleza secuencial impide paralelizar completamente el entrenamiento. La arquitectura **Transformer** supera este problema gracias a un mecanismo de *auto-atención* que procesa todos los elementos de la secuencia en paralelo y captura dependencias largas con mayor eficiencia. Sus detalles se describen en la Sección 3.7.

3.7. Transformers

El *Transformer*, introducido por Vaswani et al. [29], reemplaza la recurrencia por un mecanismo de *auto-atención* que procesa toda la secuencia en

paralelo y captura dependencias largas con gran eficiencia. En este proyecto utilizamos únicamente la parte *encoder* para clasificar ventanas de 5 s de tráfico IoT.

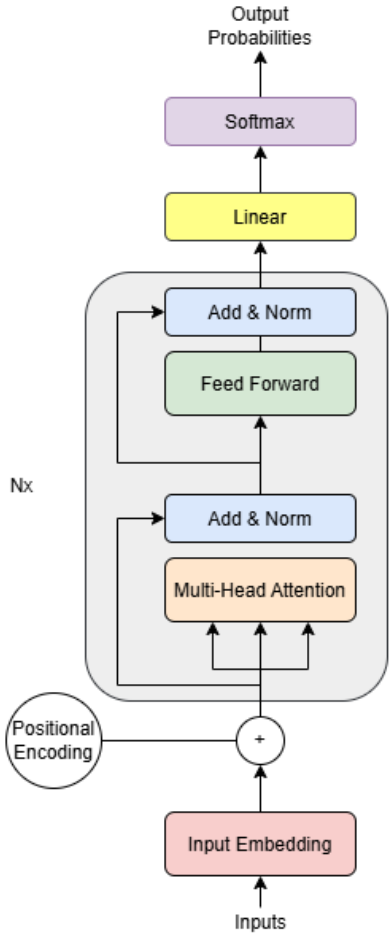

Figura 3.1: Arquitectura del encoder Transformer utilizada: cada flujo se proyecta a $d_{\text{model}} = 64$ y atraviesa $n_{\text{layers}} = 2$ bloques de atención multi-cabeza más MLP.

Parámetros clave y notación

- $\mathbf{d}_{\text{model}} = \mathbf{64}$: tamaño del **embedding**. Cada flujo, descrito inicialmente por $n_{\text{features}} = 79$ características, se proyecta a $\mathbb{R}^{64}$ mediante una capa lineal $W_e \in \mathbb{R}^{79\times 64}$.

- **$\mathbf{L = 128}$: longitud de la ventana de contexto**. Un token `[CLS]` + hasta 127 flujos reales; los que faltan se rellenan con *padding*.
- **$\mathbf{d_{ff} = 128}$**: dimensión intermedia del MLP (*feed-forward*) que sigue a la atención $64 \to 128 \to 64$.
- **$\mathbf{d_k = d_v = 16}$**: dimensión de claves y valores dentro de cada cabeza de atención (d_{model}/h al usar $h = 4$ cabezas).

Atención multi-cabeza

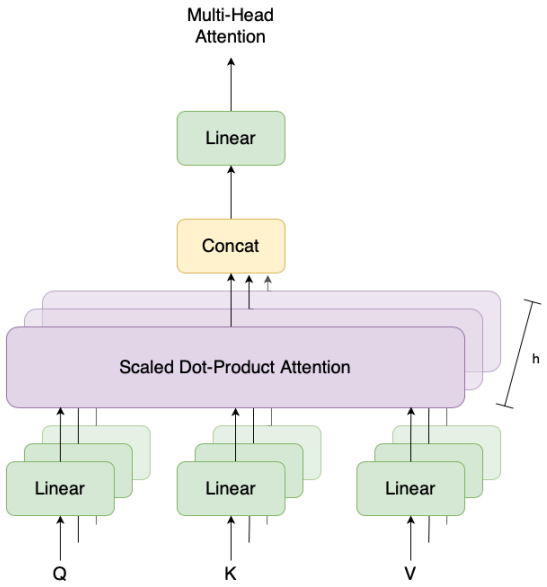

Figura 3.2: Atención multi-cabeza con $h = 4$ cabezas, todas recibiendo la misma entrada $H \in \mathbb{R}^{L \times 64}$.

Cada cabeza aprende a fijarse en aspectos distintos de los flujos:

$$Q_i = HW_i^Q, \quad K_i = HW_i^K, \quad V_i = HW_i^V,$$

con $W_i^Q, W_i^K, W_i^V \in \mathbb{R}^{64 \times 16}$. Para un flujo concreto $x_1 \in \mathbb{R}^{1 \times 64}$ (es decir, una *palabra* de la ventana):

$$q_{1,i} = x_1 W_i^Q \in \mathbb{R}^{1 \times 16}, \qquad k_{1,i}, v_{1,i} \text{ análogamente.}$$

Intuitivamente:

- **Q** (*queries*) pregunta: «¿en qué otros flujos debería fijarme para tomar mi decisión?».
- **K** (*keys*) describe de qué trata cada flujo.
- **V** (*values*) contiene la información que se combinará.

En la práctica todo se calcula en bloque con matrices; concatenamos las salidas de las h cabezas y las proyectamos de vuelta a $\mathbb{R}^{64}$, manteniendo la dimensión original.

Atención escalada producto-punto

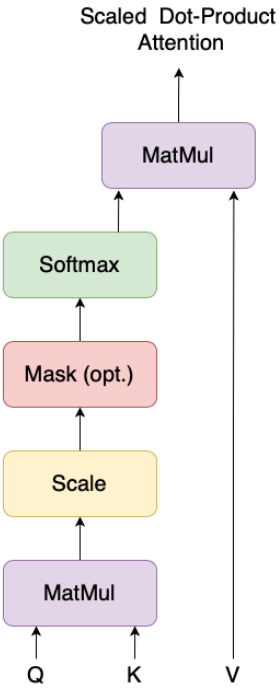

Figura 3.3: Atención escalada producto-punto entre consultas Q, claves K y valores V.

La atención de una cabeza se define como

$$\mathrm{Att}(Q, K, V) = \mathrm{softmax}\Big(\frac{QK^\top}{\sqrt{d_k}} \; + \; M\Big)V,$$

donde M es una máscara que pone $-\infty$ en las posiciones que corresponden a tokens de *padding*, evitando que influyan en el resultado.

El factor $\sqrt{d_k}$ evita que los productos punto crezcan demasiado cuando d_k es grande, estabilizando los gradientes.

Ejemplo paso a paso (vector a vector)

Sea $L = 3$ para simplificar (un `[CLS]` y dos flujos reales). Depués de proyectar con W^Q, W^K, W^V de *una* cabeza ($d_k = d_v = 16$), obtenemos:

$$Q = \begin{bmatrix} q_0 \\ q_1 \\ q_2 \end{bmatrix}, \quad K = \begin{bmatrix} k_0 \\ k_1 \\ k2 \end{bmatrix}, \quad V = \begin{bmatrix} v_0 \\ v_1 \\ v_2 \end{bmatrix} \in \mathbb{R}^{3\times 16}$$

1. Puntuaciones (escalares). Para el **primer flujo real** ($q_1 \in \mathbb{R}^{1\times 16}$) calculamos el producto punto con todas las claves del resto de flujos, incluido él mismo:

$$s_{1j} = q_1 k_j^\top \in \mathbb{R}, \qquad j = 0, 1, 2.$$

2. Escalado y normalización.

$$\alpha_{1j} = \frac{\exp\Big(s_{1j}/\sqrt{d_k}\Big)}{\sum_{m=0}^{2} \exp\Big(s_{1m}/\sqrt{d_k}\Big)} \quad \Longrightarrow \quad \boldsymbol{\alpha_1} = [\alpha_{10}, \alpha_{11}, \alpha_{12}] \in \mathbb{R}^{1\times 3}.$$

Cada α_{1j} es un **escalar** que indica cuánta información del flujo j «atiende» el flujo 1.

3. Combinación lineal.

$$z_1 = \sum_{j=0}^{2} \alpha_{1j}\, v_j = \alpha_{10} v_0 + \alpha_{11} v_1 + \alpha_{12} v_2 \quad \in \mathbb{R}^{1\times 16}$$

Así:

- v_j (valor) $\quad \in \mathbb{R}^{1\times 16}$.
- α_{1j} $\quad$ escalar multiplicando v_j.
- La suma preserva la dimensión $1 \times d_v$.

Símbolo	Dimensión
q_i, k_i	1×16
v_i, z_i	1×16
α_{ij}	escalar
Q, K, V, Z	$L \times 16$
Z^{multi}	$L \times 64$

Tabla 3.2: Dimensiones de los tensores en una cabeza de atención.

4. Matriz completa. Repitiendo para q_0 y q_2 obtenemos $Z = \text{Att}(Q, K, V) \in \mathbb{R}^{3\times16}$. Con h cabezas concatenamos:

$$Z^{\text{multi}} = \text{concat}(Z^{(1)}, \ldots, Z^{(h)}) \in \mathbb{R}^{3\times64},$$

recuperando d_{model}.

Este cálculo «vector a vector» deja claro que la salida del primer flujo $\left(z_1\right)$ no es un valor aislado, sino una mezcla ponderada de todos los valores $\{v_0, v_1, v_2\}$. En otras palabras, los pesos α_{1j} indican *cuánta atención merece cada flujo según lo útil que resulta para detectar un ataque concreto*: si dos flujos exhiben patrones fuertemente relacionados con la misma clase de ataque, su producto punto crece y el modelo les otorga más peso; por el contrario, los flujos irrelevantes para la clasificación reciben pesos cercanos a cero y su influencia se difumina.

Vector `[CLS]` y capa de clasificación

Al terminar las $n_{\text{layers}} = 2$ capas obtenemos $H^{(2)} \in \mathbb{R}^{128\times64}$. Tomamos la primera fila, $h_{\texttt{[CLS]}}$, como resumen de la ventana: contiene una "visión de conjunto" que integra la interacción entre todos los flujos. Ese vector alimenta una última capa lineal + *softmax* para predecir la clase de la ventana (p. ej., *normal* versus *anómala*).

3.8. API REST

Una **API REST** (*Representational State Transfer*) es una interfaz de programación de aplicaciones basada en el protocolo HTTP, diseñada para el intercambio eficiente y sencillo de información entre sistemas distribuidos. REST establece un conjunto de principios que definen cómo los recursos

web deben ser identificados, accedidos y manipulados utilizando métodos estándar como `GET`, `POST`, `PUT` y `DELETE`.

Cada recurso en una API REST es accesible mediante una URL específica, permitiendo una gestión clara y escalable. Las APIs REST utilizan frecuentemente formatos ligeros y ampliamente aceptados como `JSON` o `XML` para el intercambio de datos. Gracias a estas características, REST se ha convertido en un estándar predominante para desarrollar aplicaciones web modernas y servicios en la nube debido a su simplicidad, flexibilidad y eficiencia [18].

4. Técnicas y herramientas

En este apartado se hará referencia a las principales herramientas empleadas en este trabajo. Se realiza una breve explicación de las tecnologías más importantes junto a sus referencias bibliográficas para ampliar información.

Extracción de flujos Los archivos PCAP se convirtieron en datos tabulares *CSV* mediante la herramienta CICFlowMeter [23], representando cada fila un flujo de red y cada una de las columnas estadísticas del flujo de red.

Desarollo del modelo y preprocesado La implementación del Transformer se realizó desde cero en PyTorch, aprovechando su sistema de diferenciación automática y su flexibilidad a la hora de crear tensores para crear componentes como la atención multicabeza [12]; los flujos se agruparon en ventanas mediante un scrip propio implementado en Python; y la manipulación y la normalización de estos datos se realizaron con NumPy [9] y Pandas [10]; para la evaluación del modelo se empleo la librería de Scikit-learn [13].

Backend de inferencia El servicio REST se desarrolló con FastAPI [2], proporciona rutas asíncronas que devuelven las predicciones del modelo en formato JSON, listas para ser consumidas por la capa web.

Interfaz de usuario Para la interfaz gráfica, Bootstrap 5 [5] fue empleado para crear diseños responsivos y modernos; DataTables [8] fue empleado para la gestión de tablas interactivas con paginación y búsqueda mediante filtros; y Plotly.js [11] permitió la creación de gráficos dinámicos (series temporales, matrices de confusión).

Herramientas	Modelo	Web	API REST	Memoria
Git	X	X	X	X
GitHub	X	X	X	X
CICFlowMeter	X			
Python	X		X	
JavaScript	X			
Jupyter Notebook	X			
Visual Studio Code	X	X	X	
Mendeley				X
Draw.io				X
LaTeX				X
PyTorch	X			
Scikit-learn	X		X	
Pandas	X		X	
Numpy	X		X	
Matplotlib	X			
HTML5		X		
CSS3		X		
Bootstrap 5		X		
Plotly.js		X		
DataTables		X		
JSON		X	X	
FastAPI			X	

Tabla 4.1: Herramientas y tecnologías utilizadas en cada parte del proyecto

Documentación La memoria se redacto con LaTeX y la bibliografía fue gestionada con el gestor de referencias Mendeley [14]. Los diagramas fueron creados con la herramienta online Draw.io [1].

Control de versiones El control de versiones se llevó a cabo con Git [3], sincronizando los commits en un repositorio público de GitHub [4].

El resumen completo de las herramientas y tecnologías empleadas se recoge en la Tabla 4.1.

5. Aspectos relevantes del desarrollo del proyecto

El objetivo principal de este proyecto es crear un *Sistema de Detección de Intrusiones (IDS)* que pueda ser embebido en un router con recursos modestos dentro de una red local de dispositivos IoT y que sea capaz de detectar en intervalos de 5 s el tipo de tráfico que está recibiendo de otras redes. En este caso estamos trabajando con datos referentes a redes IoT pero esto se puede extrapolar a datos de redes de computadoras. La Figura 5.1 viene a graficar esto; el IDS se integra directamente en el router de la red local, donde inspecciona de forma continua ventanas de 5 s y clasifica cada ventana como *normal* o como uno de los cinco tipos de ataques definidos (DDoS-ICMP, DDoS-TCP, DoS-TCP, OS Scan y Port Scan). Esta ubicación dentro del router doméstico permite bloquear o aislar de inmediato cualquier tráfico malicioso, ya provenga de dispositivos internos comprometidos o de redes IoT externas, sin necesidad de hardware adicional ni de enviar datos a la nube.

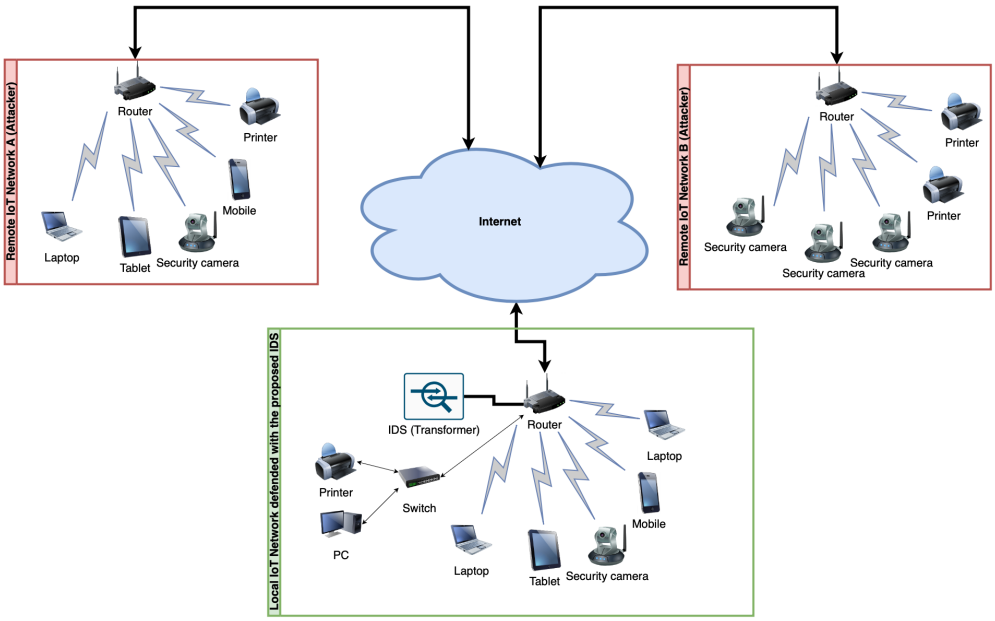

Figura 5.1: Topología que representa un entorno distribuido realista con dispositivos IoT. Dos redes IoT remotas (A y B), que incluyen dispositivos potencialmente comprometidos, generan tráfico hacia la red IoT local protegida. El IDS propuesto, basado en un modelo Transformer, se encuentra embebido en el router y clasifica en tiempo real todos los flujos entrantes para detectar posibles intrusiones.

El trabajo se ha dividido en siete partes, cada una de ellas con un propósito distinto:

- **Búsqueda de un dataset IoT:** Donde hablaremos sobre los dos tipos de datasets IoT que nos podemos encontrar, su ventajas e inconvenientes y detallaremos el dataset empleado para este trabajo.
- **Obtención de flujos individuales a partir de archivos PCAP:** A partir de los datos en bruto, se transforman los paquetes en un archivo *CSV* donde cada línea representa un flujo de red junto con sus estadísticas asociadas.
- **Agrupación de flujos en ventanas consecutivas:** Como la idea de este trabajo no es clasificar flujos individuales de red, sino clasificar ventanas temporales cada 5 s, asignaremos cada flujo a su(s) correspondiente(s) ventana(s). Se verá también que dependiendo de la

naturaleza del tráfico cada ventana tendrá mayor o menor número de flujos.

- **Generación de tensores de tamaño fijo:** El modelo Transformer requiere una entrada de tamaño fijo. Como hemos dicho antes el número de flujos por ventanas es variable, en esta sección veremos como se ha solventado este problema.
- **Implementación del Transformer en PyTorch:** Se implementará la parte del encoder de un Transformer en PyTorch desde cero.
- **Evaluación del modelo:** Se evaluará el modelo en el conjunto de test con ventanas nunca vistas anteriormente, recogiendo algunas métricas de interés como son la Precisión, el Recall, F1-Score y una matriz de confusión.
- **Despliegue de la aplicación web:** Finalmente, se mostrará una demo funcional de la aplicación web desarrollada para visualizar los resultados de predicción de forma sencilla e interactiva.

5.1. Búsqueda de un dataset IoT

El primer paso en este trabajo fue buscar un conjunto de datos que reflejara lo más fielmente posible un entorno IoT real, donde se mezclara tráfico legítimo con ataques lanzados desde dispositivos comprometidos. Aunque existen datasets sintéticos o generados en entornos controlados muy simples, el objetivo era trabajar con algo más realista, a ser posible con una topología amplia y dispositivos IoT de verdad.

En este caso se optó por el dataset **CICIoT2023: A real-time dataset and benchmark for large-scale attacks in IoT environment**, desarrollado por el Canadian Institute for Cybersecurity [25]. Este conjunto de datos recoge el tráfico de una red donde conviven hasta 105 dispositivos IoT reales (cámaras, altavoces, sensores, enchufes, etc.) y en la que se simulan más de 30 ataques de diferente naturaleza (DDoS, DoS, escaneo, fuerza bruta, spoofing, malware tipo Mirai, etc.). Lo interesante de este dataset es que los ataques son lanzados por dispositivos IoT infectados que se comportan como atacantes internos, tal y como puede ocurrir en la vida real, y todo el tráfico se captura en tiempo real.

Este tipo de dataset es perfecto para el enfoque de este TFG, ya que permite no solo detectar intrusiones externas, sino también identificar com-

portamientos anómalos dentro de la red local, que es justo el objetivo de este IDS embebido.

En las siguientes secciones se explicará cómo se ha transformado este dataset desde los archivos *PCAP* originales divididos por tipo de tráfico hasta las ventanas temporales de flujos que se usan como entrada al modelo Transformer.

5.2. Obtención de flujos individuales a partir de archivos PCAP

Tras la búsqueda y selección del dataset descrita en la Sección 5.1, el siguiente paso consiste en transformar las capturas de paquetes (archivos PCAP) en flujos bidireccionales etiquetados. Para ello se ha empleado **CICFlowMeter**, herramienta presentada en el Capítulo 3.3. CICFlowMeter agrupa los paquetes que comparten la 5-tupla $\{\text{IP}_{\text{origen}}, \text{IP}_{\text{destino}}, \text{puerto}_{\text{origen}}, \text{puerto}_{\text{destino}}, \text{protocolo}\}$ y genera para cada flujo más de 80 métricas estadísticas. El resultado final es un fichero *CSV* donde cada línea representa un flujo de red (ver la definición de "flujo" en la Sección 3.3); estas columnas serán la materia prima de la que se nutrirá el script de ventaneo temporal descrito en la Sección 5.3.

Para evitar que un flujo se extendierá a lo largo de varias ventanas temporales consecutivas (especialmente en flujos con protocolo UDP, donde no existe una señal explícita de finalización) se definieron los siguientes parámetros en la herramienta de CICFlowMeter:

- **Flow timeout:** se estableció en **10 segundos**, lo que significa que si no se observa tráfico adicional asociado a un flujo durante ese intervalo, este se considera finalizado y se genera un nuevo flujo en caso de retomarse posteriormente.
- **Activity timeout:** se fijó en **5 segundos**, lo cual limita la duración máxima de un flujo activo, incluso si sigue recibiendo paquetes. Esto controla que un mismo flujo como máximo aparezca en dos ventanas temporales consecutivas.

En la Tabla 5.1 se resumen todas las características generadas por CICFlowMeter.

Flow Duration	Duración total del flujo en microsegundos
Total Fwd Packets	Número de paquetes en la dirección origen→destino
Total Bwd Packets	Número de paquetes en la dirección destino→origen
Total Length of Fwd Packets	Suma de tamaños de los paquetes forward (bytes)
Total Length of Bwd Packets	Suma de tamaños de los paquetes backward (bytes)
Fwd Packet Length Min	Tamaño mínimo de paquete forward
Fwd Packet Length Max	Tamaño máximo de paquete forward
Fwd Packet Length Mean	Tamaño medio de paquete forward
Fwd Packet Length Std	Desviación estándar del tamaño de paquete forward
Bwd Packet Length Min	Tamaño mínimo de paquete backward
Bwd Packet Length Max	Tamaño máximo de paquete backward
Bwd Packet Length Mean	Tamaño medio de paquete backward
Bwd Packet Length Std	Desviación estándar del tamaño de paquete backward
Flow Bytes/s	Bytes por segundo en el flujo
Flow Packets/s	Paquetes por segundo en el flujo
Flow IAT Mean	Media del *Inter-Arrival Time* entre paquetes
Flow IAT Std	Desviación estándar del IAT del flujo
Flow IAT Max	Máximo IAT del flujo
Flow IAT Min	Mínimo IAT del flujo
Fwd IAT Min	IAT mínimo forward
Fwd IAT Max	IAT máximo forward
Fwd IAT Mean	IAT medio forward
Fwd IAT Std	Desviación estándar del IAT forward
Fwd IAT Total	Suma de IAT forward
Bwd IAT Min	IAT mínimo backward

continúa en la página siguiente

continúa desde la página anterior

Bwd IAT Max	IAT máximo backward
Bwd IAT Mean	IAT medio backward
Bwd IAT Std	Desviación estándar del IAT backward
Bwd IAT Total	Suma de IAT backward
Fwd PSH Flags	Recuentos del flag PSH en sentido forward (TCP)
Bwd PSH Flags	Recuentos del flag PSH en sentido backward (TCP)
Fwd URG Flags	Recuentos del flag URG en sentido forward (TCP)
Bwd URG Flags	Recuentos del flag URG en sentido backward (TCP)
Fwd Header Length	Bytes de cabeceras en sentido forward
Bwd Header Length	Bytes de cabeceras en sentido backward
Fwd Packets/s	Paquetes forward por segundo
Bwd Packets/s	Paquetes backward por segundo
Packet Length Min	Longitud mínima de paquete en el flujo
Packet Length Max	Longitud máxima de paquete en el flujo
Packet Length Mean	Longitud media de paquete en el flujo
Packet Length Std	Desviación estándar de la longitud de paquete
Packet Length Variance	Varianza de la longitud de paquete
FIN Flag Count	Número de paquetes con flag FIN
SYN Flag Count	Número de paquetes con flag SYN
RST Flag Count	Número de paquetes con flag RST
PSH Flag Count	Número de paquetes con flag PSH
ACK Flag Count	Número de paquetes con flag ACK
URG Flag Count	Número de paquetes con flag URG
CWR Flag Count	Número de paquetes con flag CWR
ECE Flag Count	Número de paquetes con flag ECE
Down/Up Ratio	Ratio bytes descargados/subidos
Average Packet Size	Tamaño medio de paquete

continúa en la página siguiente

continúa desde la página anterior	
Fwd Segment Size Avg	Tamaño medio de segmento forward
Bwd Segment Size Avg	Tamaño medio de segmento backward
Fwd Bytes/Bulk Avg	Bytes bulk medios forward
Fwd Packet/Bulk Avg	Paquetes bulk medios forward
Fwd Bulk Rate Avg	Tasa bulk media forward
Bwd Bytes/Bulk Avg	Bytes bulk medios backward
Bwd Packet/Bulk Avg	Paquetes bulk medios backward
Bwd Bulk Rate Avg	Tasa bulk media backward
Subflow Fwd Packets	Paquetes forward por subflujo
Subflow Fwd Bytes	Bytes forward por subflujo
Subflow Bwd Packets	Paquetes backward por subflujo
Subflow Bwd Bytes	Bytes backward por subflujo
Fwd Init Win Bytes	Bytes enviados en la ventana inicial forward
Bwd Init Win Bytes	Bytes enviados en la ventana inicial backward
Fwd Act Data Pkts	Paquetes con payload TCP (>0 B) forward
Fwd Seg Size Min	Tamaño mínimo de segmento forward
Active Min	Tiempo mínimo activo antes de quedar inactivo
Active Mean	Tiempo medio activo antes de quedar inactivo
Active Max	Tiempo máximo activo antes de quedar inactivo
Active Std	Desviación estándar del tiempo activo
Idle Min	Tiempo mínimo inactivo antes de reactivarse
Idle Mean	Tiempo medio inactivo antes de reactivarse
Idle Max	Tiempo máximo inactivo antes de reactivarse
Idle Std	Desviación estándar del tiempo inactivo

Tabla 5.1: Características extraídas por CICFlowMeter.

5.3. Agrupación de flujos en ventanas consecutivas

Tras obtener los flujos individuales (ver Sección 5.2), el siguiente paso fue agruparlos en intervalos de tiempo de 5 s. Cada uno de estos intervalos (o "ventana") se convierte en un *bloque* de flujos, es decir, en una lista de

todos los flujos cuyo *timestamp de inicio* y *timestamp de fin* cae dentro de dicho intervalo.

De este modo, en los análisis posteriores ya no trabajamos a nivel de flujos individuales, sino sobre una *secuencia de ventanas*, donde cada ventana contiene su conjunto de flujos ordenados cronológicamente por *timestamp de creación de cada flujo*. Este enfoque permite al modelo capturar la estructura temporal del tráfico y detectar patrones de comportamiento anómalo de forma más robusta.

El diagrama de flujo del script propio implementado en Python que agrupa estos flujos en ventanas se puede ver en la Figura 5.2.

En resumen, el procedimiento empleado es el siguiente:

1. Leer el fichero CSV con los flujos extraídos.

2. Para cada flujo, calcular la primera y la última ventana en la que puede caer, con la información que nos da sus timestamps de inicio y fin.

3. Iterar sobre cada ventana desde la primera hasta la última:

   - Si existe solapamiento temporal (`ts_inicio_flujo < fin_ventana && ts_fin_flujo > inicio_ventana`), asignar el flujo a esa ventana.

   - En caso contrario, avanzar a la siguiente ventana.

4. Repetir el proceso para el siguiente flujo.

En la Tabla 5.2 se presentan las estadísticas del número de flujos por ventana, extraídas de cada fichero de nuestro dataset y agrupadas por tipo de tráfico:

Archivo	Etiqueta	# Ventanas de 5 s	Mín flujos	Máx flujos	Media flujos	Mediana flujos
./Benign_0.csv	BENIGN	6667	12	227	58.43	59.0
./Benign_1.csv	BENIGN	5220	12	235	43.92	43.0
./Benign_2.csv	BENIGN	5785	5	145	43.07	43.0
./Benign_3.csv	BENIGN	2493	5	77	42.01	42.0
./DDoS-ICMP_0.csv	DDoS-ICMP	421	4	208	70.86	69.0
./DoS-TCP-0_00000_Flow_ventanas.csv	DoS-TCP	111	2	37587	11603.96	13071.0
./DoS-TCP-0_00001_Flow_ventanas.csv	DoS-TCP	53	352	76705	44558.38	61162.0
./DoS-TCP-0_00004_Flow_ventanas.csv	DoS-TCP	16	15751	70619	54415.19	64736.5
./DoS-TCP-1_00000_Flow_ventanas.csv	DoS-TCP	57	3	74421	47755.04	63284.0
./DoS-TCP-1_00002_Flow_ventanas.csv	DoS-TCP	330	2	64624	1295.90	62.0
./DoS-TCP-1_00004_Flow_ventanas.csv	DoS-TCP	11	37	65545	26286.91	27095.0
./DoS-TCP-2_00000_Flow_ventanas.csv	DoS-TCP	48	4	65611	19550.96	7189.0
./DoS-TCP-2_00001_Flow_ventanas.csv	DoS-TCP	100	30	69574	32695.59	34943.0
./DoS-TCP-2_00002_Flow_ventanas.csv	DoS-TCP	289	4	73915	14943.34	47.0
./DoS-TCP-2_00004_Flow_ventanas.csv	DoS-TCP	35	3	76924	52789.26	62707.0
./DDoS-TCP-0_Flow_ventanas.csv	DDoS-TCP	382	15	5717	118.17	78.0
./DDoS-TCP-1_Flow_ventanas.csv	DDoS-TCP	310	5	59261	761.38	90.5
./DDoS-TCP-2_Flow_ventanas.csv	DDoS-TCP	313	2	42720	295.05	61.0
./DDoS-TCP-3_Flow_ventanas.csv	DDoS-TCP	312	1	121	51.64	50.0
./Recon-OSScan.pcap_Flow_ventanas.csv	OSScan	1672	1	1239	122.26	66.0
./Recon-PortScan.pcap_Flow_ventanas.csv	PortScan	1522	2	1463	142.37	64.0

Tabla 5.2: Estadísticas de flujos por ventana según tipo de tráfico

5.4. Generación de tensores de tamaño fijo

Una vez agrupados los flujos en ventanas de 5 s (Sección 5.3), cada ventana contiene un conjunto de m flujos, y cada flujo consiste en un vector de d características (en nuestro caso, $d = 79$, extraídas por CICFlowMeter).

Para poder alimentar al Transformer es necesario que cada ventana esté representada como un tensor de tamaño *constante* $(L \times d)$, donde

$$L = \text{max\_seq\_len} = 128.$$

Denotemos por

$$F \in \mathbb{R}^{m \times d}$$

la matriz de características original de una ventana con m flujos. Definimos entonces el tensor fijo $\widetilde{F} \in \mathbb{R}^{L \times d}$ como:

$$\widetilde{F} = \begin{cases} F_{1:L,\,:}, & m \geq L, \\ \begin{bmatrix} F \\ \mathbf{0}_{(L-m)\times d} \end{bmatrix}, & m < L, \end{cases}$$

donde

- $F_{1:L,\,:}$ indica que, en caso de $m \geq L$, se toman únicamente las primeras L filas de F (truncamiento).
- $\mathbf{0}_{(L-m)\times d}$ es una matriz nula de tamaño $(L-m) \times d$ que sirve para rellenar con ceros las filas restantes (padding).

El pseudocódigo de la función principal que realiza este proceso es el siguiente:

```
def load_window_samples(csv_paths, max_seq_len=128):
    # ... lectura y limpieza de datos ...
    for cada ventana vid:
        subdf = df[... en esa ventana ...]
        feats = subdf[numeric_cols].to_numpy()  # shape (m, d)
        if m > max_seq_len:
            feats = feats[:max_seq_len, :]
        else:
            pad_len = max_seq_len - m
            pad = np.zeros((pad_len, d), dtype=np.float32)
            feats = np.vstack([feats, pad])
        samples.append((feats, label_idx))
    return samples
```

Una vez construido cada tensor $\widetilde{F}$, se asocia su etiqueta $y \in \{0, \ldots, C-1\}$ y se almacena como par $(\widetilde{F}, y)$. Estos pares quedan listos para escalarse (normalización de características) y cargarse en un `DataLoader` de PyTorch mediante la clase:

```
class WindowDataset(Dataset):
    def __init__(self, samples, scaler):
        self.samples = samples
        self.scaler = scaler
```

```
    def __getitem__(self, idx):
        arr, lbl = self.samples[idx] # arr: (L, d)
        flat = arr.reshape(-1, d)    # plano para normalizar
        scaled = self.scaler.transform(flat)
                    .reshape(L, d)
        return torch.from_numpy(scaled), torch.tensor(lbl)
```

De este modo, cada minibatch que reciba el modelo está formado por tensores de forma

$$(\text{batch\_size},\, L,\, d)\,,$$

con $L = 128$ y $d = 79$, garantizando entradas de tamaño fijo para el modelo Transformer.

5.5. Implementación del Transformer en PyTorch

En esta sección describimos la implementación del encoder Transformer dividida en sus módulos principales, acompañada del código correspondiente en PyTorch.

Atención Multicabeza (`MultiHeadAttention`)

Permite al modelo atender a distintos aspectos de la secuencia en paralelo.

```
class MultiHeadAttention(nn.Module):
    def __init__(self, d_model=512, num_heads=8):
        super().__init__()
        assert d_model % num_heads == 0, \
            'd_model debe ser divisible por num_heads.'
        self.d_k = self.d_v = d_model // num_heads
        self.num_heads = num_heads
        self.W_q = nn.Linear(d_model, d_model)
        self.W_k = nn.Linear(d_model, d_model)
        self.W_v = nn.Linear(d_model, d_model)
        self.W_o = nn.Linear(d_model, d_model)

    def forward(self, Q, K, V, mask=None):
        B = Q.size(0)
        # Proyección y reshape para num_heads
```

```
        Q = self.W_q(Q).view(B, -1, self.num_heads, self.d_k)\
                       .transpose(1,2)
        K = self.W_k(K).view(B, -1, self.num_heads, self.d_k)\
                       .transpose(1,2)
        V = self.W_v(V).view(B, -1, self.num_heads, self.d_k)\
                       .transpose(1,2)
        # Cálculo de scores y softmax
        scores = torch.matmul(Q, K.transpose(-2,-1)) \
                 / math.sqrt(self.d_k)
        if mask is not None:
            scores =
                scores.masked_fill(mask == 0, float('-inf'))
        attn = F.softmax(scores, dim=-1)
        # Aplicación de la atención a V
        # y combinación de cabezas
        out = torch.matmul(attn, V)
        out = out.transpose(1,2).contiguous()\
                 .view(B, -1, self.num_heads * self.d_k)
        return self.W_o(out), attn
```

Capa Feed-Forward Posicional (`PositionFeedForward`)

Transforma cada vector de dimensión d_{model} mediante un MLP simple.

```
class PositionFeedForward(nn.Module):
    def __init__(self, d_model, d_ff):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(d_model, d_ff),
            nn.ReLU(),
            nn.Linear(d_ff, d_model)
        )

    def forward(self, x):
        return self.net(x)
```

Subcapa del Encoder (`EncoderSubLayer`)

Combina Self-Attention, Dropout, Residual
+ LayerNorm y Feed-Forward.

```
class EncoderSubLayer(nn.Module):
    def __init__(self, d_model, num_heads, d_ff, dropout=0.1):
        super().__init__()
        self.self_attn =
            MultiHeadAttention(d_model, num_heads)
        self.ffn       = PositionFeedForward(d_model, d_ff)
        self.norm1     = nn.LayerNorm(d_model)
        self.norm2     = nn.LayerNorm(d_model)
        self.do1       = nn.Dropout(dropout)
        self.do2       = nn.Dropout(dropout)

    def forward(self, x, mask=None):
        attn_out, _ = self.self_attn(x, x, x, mask)
        x = self.norm1(x + self.do1(attn_out))
        ffn_out = self.ffn(x)
        x = self.norm2(x + self.do2(ffn_out))
        return x
```

Encoder Completo (`Encoder`)

Apila varias `EncoderSubLayer` y aplica una normalización final.

```
class Encoder(nn.Module):
    def __init__(self, d_model, num_heads,
                 d_ff, num_layers, dropout=0.1):
        super().__init__()
        self.layers = nn.ModuleList([
            EncoderSubLayer(d_model, num_heads, d_ff, dropout)
            for _ in range(num_layers)
        ])
        self.norm = nn.LayerNorm(d_model)

    def forward(self, x, mask=None):
        for layer in self.layers:
            x = layer(x, mask)
        return self.norm(x)
```

Embeddings Posicionales (`PositionalEmbedding`)

Genera sin y cos fijos para codificar la posición en la secuencia.

```
class PositionalEmbedding(nn.Module):
    def __init__(self, d_model, max_seq_len=512):
        super().__init__()
        pe = torch.zeros(max_seq_len, d_model)
        pos = torch.arange(0, max_seq_len)
            .unsqueeze(1).float()
        div = torch.exp(torch.arange(0, d_model, 2).float()
                        * (-math.log(10000.0) / d_model))
        pe[:, 0::2] = torch.sin(pos * div)
        pe[:, 1::2] = torch.cos(pos * div)
        self.register_buffer('pe', pe.unsqueeze(0))

    def forward(self, x):
        return x + self.pe[:, :x.size(1), :]
```

Clasificador con Token [CLS] (TransformerEncoderClassifierWithCLS)

Concadena un token [CLS], aplica embeddings posicionales, el encoder y termina en una capa lineal de salida.

```
class TransformerEncoderClassifierWithCLS(nn.Module):
    def __init__(self, d_model, num_heads, d_ff,
                 num_layers, input_dim, num_classes,
                 max_seq_len=512, dropout=0.1):
        super().__init__()
        self.input_proj = nn.Linear(input_dim, d_model)
        self.cls_token  =
            nn.Parameter(torch.randn(1, 1, d_model))
        self.pos_embed  =
            PositionalEmbedding(d_model, max_seq_len)
        self.encoder    = Encoder(d_model, num_heads,
            d_ff, num_layers, dropout)
        self.classifier = nn.Linear(d_model, num_classes)

    def forward(self, x, mask=None):
        B = x.size(0)
        x = self.input_proj(x)
        cls = self.cls_token.expand(B, -1, -1)
        x = torch.cat([cls, x], dim=1)
```

```
x = self.pos_embed(x)
x = self.encoder(x, mask)
return self.classifier(x[:, 0, :])
```

Con esta organización modular queda más claro el flujo de datos y las responsabilidades de cada componente dentro del encoder Transformer.

5.6. Evaluación del modelo

Tiempo de inferencia y estudio de viabilidad en tiempo real

El modelo tarda en total 1.21 s en procesar las 5228 ventanas del conjunto de test, cuando se ejecuta sobre una GPU NVIDIA RTX 4060 Ti con 16 GB de VRAM.

$$\frac{1{,}21\ \text{s}}{5228} \approx 2{,}3 \times 10^{-4}\,\text{s} = 0{,}23\,\text{ms}.$$

Dado que cada ventana agrupa 5 s de tráfico, el coste de inferencia por ventana (0.23 ms) es prácticamente despreciable en comparación con la duración de cada intervalo de muestreo (ventana). Por tanto, si el preprocesado y generación del tensor de entrada a partir de la ventana (paso de flujos a ventanas y aplicación de padding/truncamiento) se completa en menos de

$$5\,\text{s} - 0{,}23\,\text{ms} \approx 4999{,}77\,\text{ms},$$

el sistema sería perfectamente capaz de operar en *tiempo real*, clasificando cada ventana de 5 s prácticamente al instante.

Métricas de clasificación

La Tabla 5.3 recoge las métricas de precisión, exhaustividad (recall), F1-score y soporte (*support*) para cada clase.

Clase	Precision	Recall	F1-score	Support
Benigno	0.9988	0.9911	0.9949	4033
DDoS-ICMP	0.9753	0.9405	0.9576	84
DoS-TCP	0.9764	0.9904	0.9834	209
DDoS-TCP	0.9662	0.9772	0.9716	263
OSScan	0.9849	0.9761	0.9805	335
PortScan	0.8955	0.9868	0.9390	304
Accuracy		0.9883		5228
Macro avg.	0.9662	0.9770	0.9712	5228
Weighted avg.	0.9890	0.9883	0.9885	5228

Tabla 5.3: Classification report del conjunto de test.

Las métricas se definen como:

$$\text{Precision}_i = \frac{\text{TP}_i}{\text{TP}_i + \text{FP}_i}, \qquad \text{Recall}_i = \frac{\text{TP}_i}{\text{TP}_i + \text{FN}_i},$$

$$\text{F1}_i = 2 \cdot \frac{\text{Precision}_i \times \text{Recall}_i}{\text{Precision}_i + \text{Recall}_i}, \qquad \text{Accuracy} = \frac{\sum_i \text{TP}_i}{N}.$$

Curva ROC y AUC

La Figura 5.3 muestra las curvas ROC por clase.

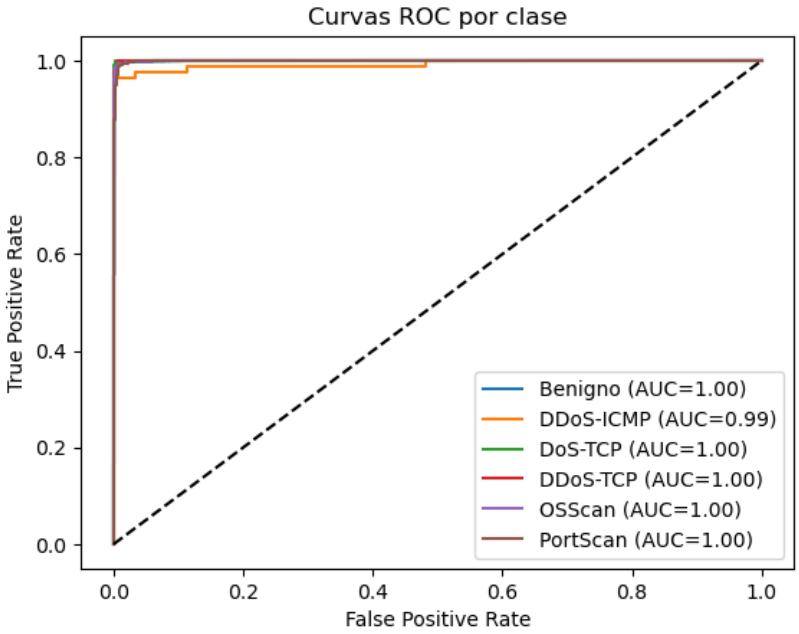

Figura 5.3: Curvas ROC por clase

Definimos para cada clase i:

$$\mathrm{TPR}_i(\tau) = \frac{\mathrm{TP}_i(\tau)}{\mathrm{TP}_i(\tau) + \mathrm{FN}_i(\tau)}, \qquad \mathrm{FPR}_i(\tau) = \frac{\mathrm{FP}_i(\tau)}{\mathrm{FP}_i(\tau) + \mathrm{TN}_i(\tau)},$$

donde τ es el umbral de decisión. La ROC traza $\mathrm{TPR}_i(\tau)$ frente a $\mathrm{FPR}_i(\tau)$ variando $\tau \in [0, 1]$.

El Área Bajo la Curva (AUC) para la clase i cuantifica la capacidad del clasificador para distinguir la clase i del resto.

Matriz de confusión

Para completar el análisis, la Figura 5.4 presenta la matriz de confusión, donde la diagonal principal contiene los conteos de clasificaciones correctas de cada ventana por clase, mientras que las celdas fuera de la diagonal vienen a representar ventanas mal clasificadas.

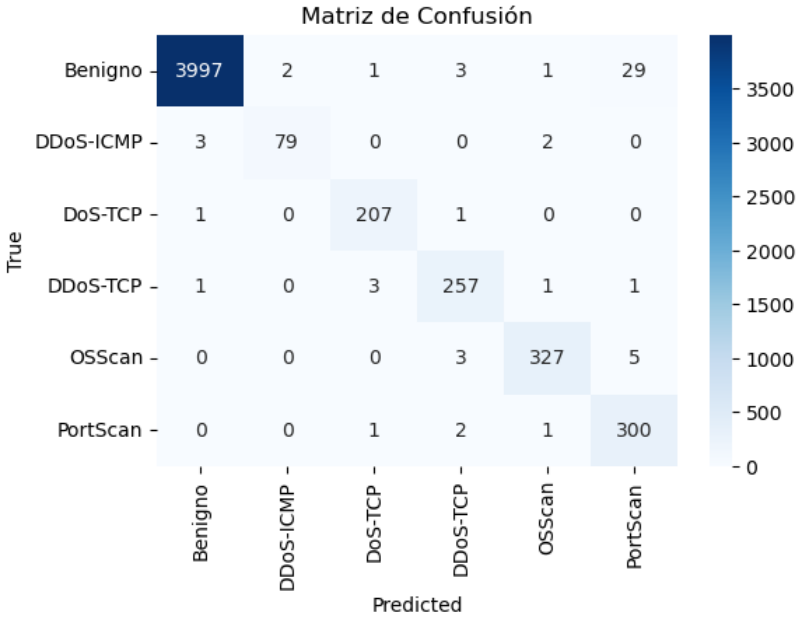

Figura 5.4: Matriz de confusión en el conjunto de test

5.7. Despliegue de la aplicación web

En este subapartado se documenta la puesta en marcha de la interfaz web desarrollada para visualizar los resultados del IDS.

Carga de una captura de red

Dadas las limitaciones de tiempo que se han tenido para este trabajo, se ha optado por cargar directamente en la web la captura en formato tabular de los flujos individuales extraida por CICFlowMeter, en lugar de los datos en brutos de red (archivo PCAP). Para futuras extensiones, sería deseable automatizar también este paso dentro del pipeline completo de preprocesado.

Para esta demo funcional, se cargará una captura de red que consiste en 15 minutos de tráfico procedente de una red local de dispositivos IoT, en la que se generan periódicamente escaneos de puertos cada 1 minuto. La tarea consiste en comprobar la capacidad del IDS para detectar correctamente estos ataques de tipo “Port Scan” frente al tráfico benigno continuo.

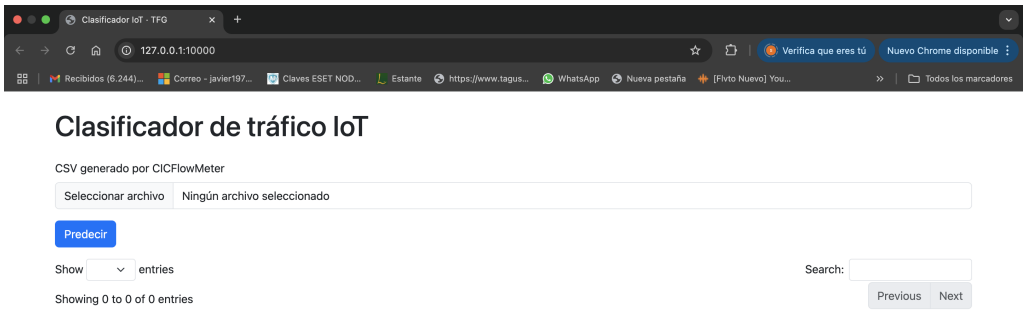

Figura 5.5: Ventana para la selección de la captura de tráfico a procesar

Tabla con resultados de la clasificación

Al procesar la captura de tráfico, lo primero que se mostrará en la web es una tabla paginada interactiva donde cada fila corresponde a una ventana de 5 s e incluye las siguientes columnas:

- **Ventana (5 s)** Intervalo temporal de la ventana.
- **Predicción** Nombre de la clase asignada por el modelo.
- **Confianza** Probabilidad (salida de la capa softmax) asociada a la clase predicha.

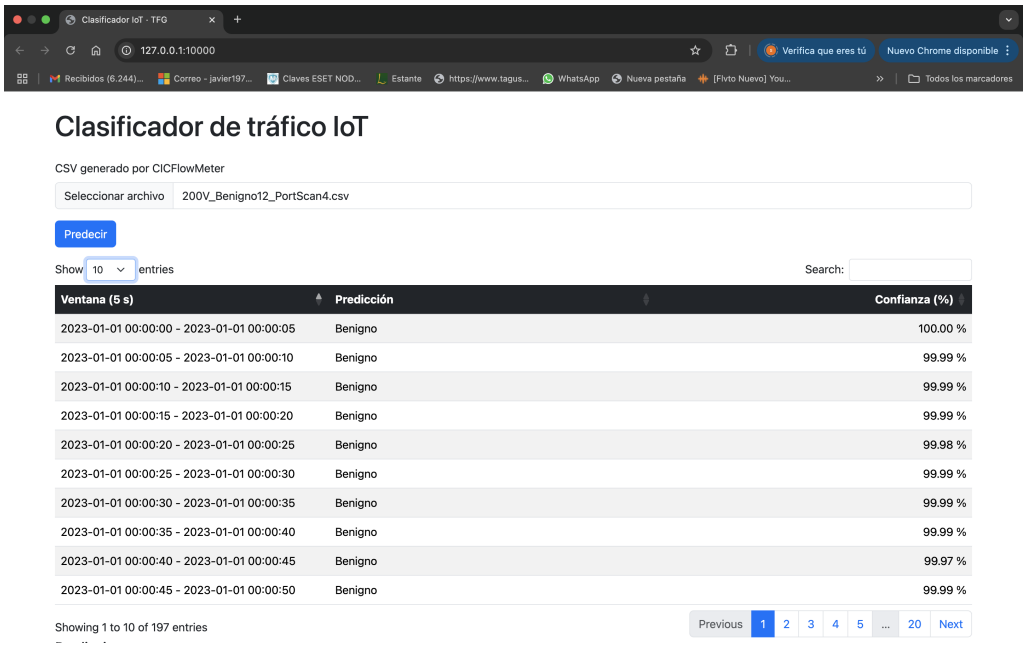

Figura 5.6: Tabla interactiva que muestra las predicciones del modelo para cada ventana de 5 s.

Comparación serie temporal de etiquetas predichas y reales

Para validar visualmente el comportamiento del IDS a lo largo del tiempo, en la web se incluyen dos gráficos de series temporales (Figura 5.7 y Figura 5.8). El primero muestra, para cada ventana de 5 s, la etiqueta predicha por el modelo, y el segundo la etiqueta real con distintos colores para cada tipo de tráfico:

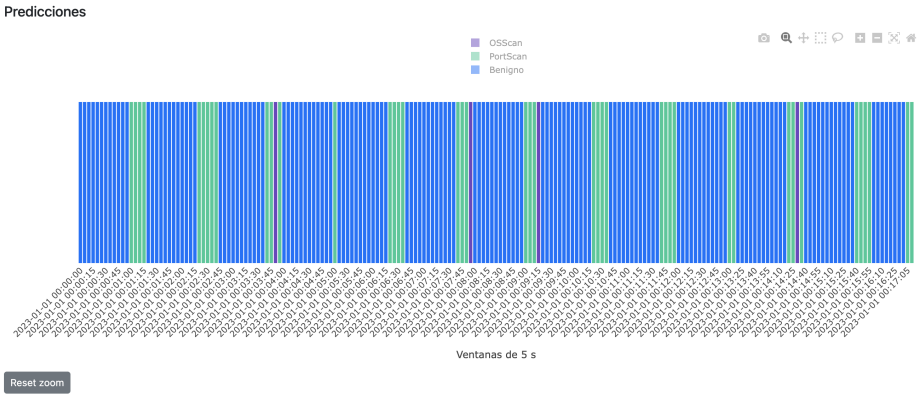

Figura 5.7: Serie temporal de etiquetas predichas por ventana de 5 s.

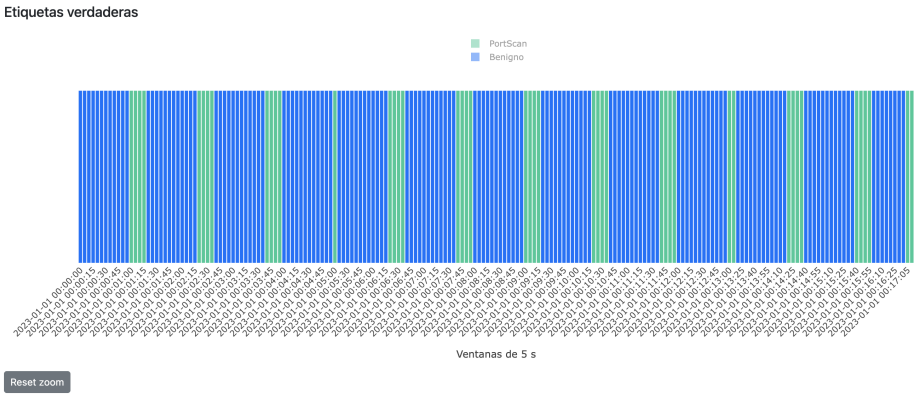

Figura 5.8: Serie temporal de etiquetas reales por ventana de 5 s.

A simple vista vemos que la predicción es bastante fiel a la realidad notando pocas ventanas en las que el modelo se confunde y predice PortScan como Benigno, y PortScan como OSScan.

Matriz de confusión y rendimiento con clases observadas

En un despliegue real sobre trazas individuales de red, suele ocurrir que solo uno o dos tipos de ataque estén presentes en un periodo dado, a pesar de que el modelo haya sido entrenado con más categorías. Por este motivo, en este caso incluiremos únicamente dos métricas globales que reflejan mejor el rendimiento sobre las clases efectivamente observadas:

- **Accuracy:** proporción de ventanas correctamente clasificadas, independiente del desbalance de clases.
- **F1-score ponderado:** media de precision y recall, ponderada por el número de ventanas de cada clase presente en la muestra.

La Figura 5.9 muestra la matriz de confusión construida solo con las clases "Benigno", "OSScan" y "PortScan" que aparecen en la demo de 15 minutos. A partir de ella, calculamos:

$$\text{Accuracy} = 97{,}46\,\%, \quad \text{F1-score ponderado} = 98{,}43\,\%.$$

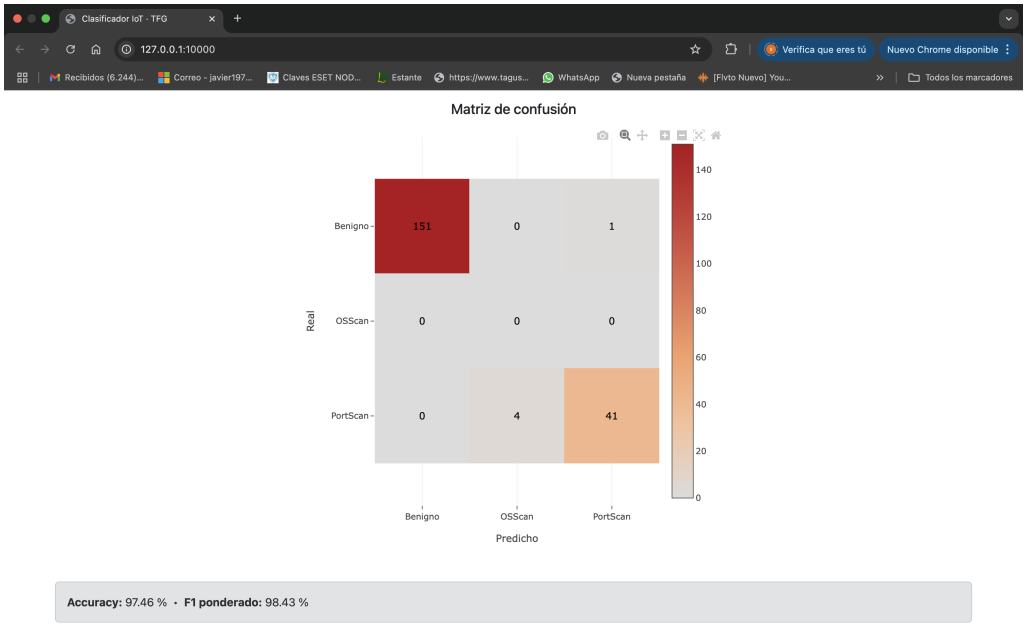

Figura 5.9: Matriz de confusión sobre las clases predichas en la demo (*Benigno*, *OSScan*, *PortScan*).

Es interesante ver en esta matriz de confusión que solo 5 ventanas de 5 s de un total de 197 ventanas fueron clasificadas erróneamente por el modelo:

- Una de ellas era "Benigna" y se clasificó como "PortScan".
- Cuatro de ellas correspondían a "PortScan" y fueron clasificadas como "OSScan".

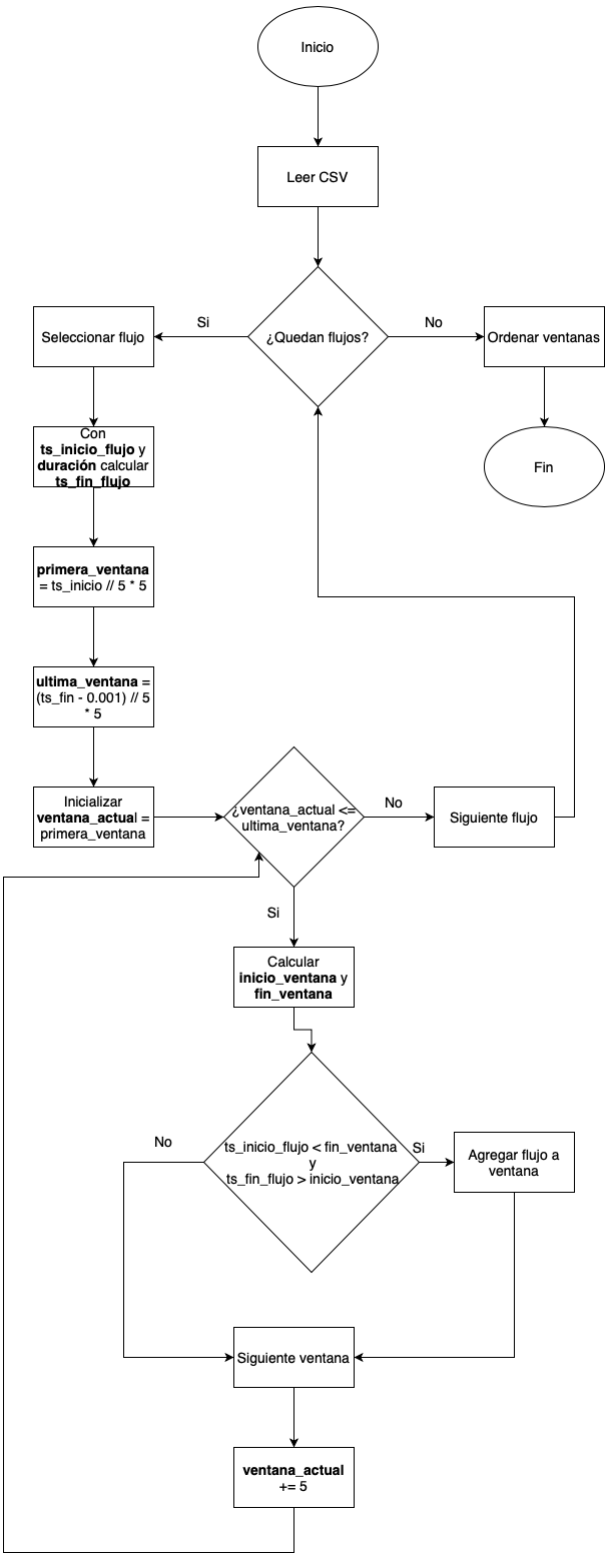

Figura 5.2: Diagrama de flujo del script de “ventaneo” de flujos de red

7. Conclusiones y Líneas de trabajo futuras

7.1. Conclusiones generales

El desarrollo de este Trabajo Fin de Grado ha permitido validar la viabilidad de utilizar modelos basados en Transformers para la clasificación de tráfico IoT en tiempo real. Partiendo de un enfoque inspirado en el procesamiento del lenguaje natural, se ha logrado transformar flujos de red en secuencias temporales estructuradas, lo que ha facilitado su tratamiento mediante arquitecturas neuronales modernas como lo son los Transformers.

Los resultados obtenidos han sido muy satisfactorios, alcanzando una precisión del 98,83 % y un F1-score ponderado del 98,85 % sobre el dataset CICIoT2023. Estas cifras reflejan no solo la eficacia del modelo en condiciones de laboratorio, sino también su potencial para ser desplegado en sistemas embebidos o entornos con recursos limitados como routers.

Además se ha comprobado que esta implementación es viable en tiempo real, al menos cuando el modelo se ejecuta desde una GPU, ya que la inferencia de una ventana de 5 segundos requiere únicamente 0,23 milisegundos.

7.2. Conclusiones técnicas

En esta sección se resumen los logros técnicos que se han logrado durante el desarrollo de este proyecto:

- Se ha demostrado que es posible representar ventanas de tráfico como tensores uniformes adecuados para modelos tipo Transformer, sin que

la perdida de flujos (en aquellas ventanas que superen más de 127 flujos) represente una pérdida de información valiosa para la clasificación de esa ventana.

- El uso de CICFlowMeter ha sido efectivo para extraer flujos útiles a partir de archivos PCAP.
- La interfaz desarrollada ha facilitado la interacción con el modelo de forma visual y sencilla, incluyendo visualización de resultados, métricas y detección de ataques en tiempo real.
- Se ha diseñado un modelo Transformer con un tamaño reducido ($d_{\text{model}} = 64$, aproximadamente 0.5 millones de parámetros), lo que permite su entrenamiento en una GPU modesta y su ejecución en dispositivos con recursos limitados.
- Se ha aprovechado el paralelismo de los Transformers para procesar los 128 flujos simultáneamente, mejorando la velocidad y eficiencia del sistema.
- El modelo ofrece una visión holística del tráfico: cada flujo en una ventana tiene en cuenta el contexto global de todos los demás flujos.
- La arquitectura es ajustable: variando parámetros como h, d_{model} o n_{layers} se puede equilibrar capacidad y coste computacional según las necesidades del entorno.

7.3. Líneas de trabajo futuras

A pesar de los resultados prometedores obtenidos, es importante señalar que tanto el entrenamiento como la validación del modelo se han realizado utilizando particiones del mismo dataset generado en un entorno de laboratorio controlado. Por tanto, una de las principales líneas de mejora consiste en evaluar la capacidad de generalización del modelo frente a datos procedentes de contextos diferentes.

Por ello, se propone como trabajo futuro la búsqueda e integración de un nuevo dataset IoT capturado en un laboratorio distinto al del CICIoT2023. Esta evaluación cruzada permitiría comprobar si el modelo es realmente extrapolable a distintas arquitecturas de red, dispositivos y configuraciones de tráfico. En caso de observarse un rendimiento aceptable, se podría aplicar una estrategia de *transfer learning* al Transformer original para adaptarlo de forma eficiente a nuevos entornos sin necesidad de entrenarlo desde cero.

Bibliografía

[1] draw.io. https://www.drawio.com/.

[2] Fastapi. https://fastapi.tiangolo.com/.

[3] Git. https://git-scm.com/.

[4] Github · build and ship software on a single, collaborative platform · github. https://github.com/.

[5] Introduction · bootstrap v5.0. https://getbootstrap.com/docs/5.0/getting-started/introduction/.

[6] Iot connections worldwide 2034| statista.

[7] Libpcap File Format. https://wiki.wireshark.org/Development/LibpcapFileFormat. Accedido: 2024-06-29.

[8] Manual. https://datatables.net/manual/index.

[9] Numpy documentation. https://numpy.org/doc/.

[10] pandas documentation — pandas 2.3.0 documentation. https://pandas.pydata.org/docs/.

[11] Plotly javascript graphing library in javascript. https://plotly.com/javascript/.

[12] Pytorch. https://pytorch.org/.

[13] scikit-learn: machine learning in python — scikit-learn 1.7.0 documentation. https://scikit-learn.org/stable/.

[14] Search | mendeley. https://www.mendeley.com/search/.

[15] tcpdump. https://www.tcpdump.org. Accedido: 2024-06-29.

[16] Wireshark. https://www.wireshark.org. Accedido: 2024-06-29.

[17] Benoit Claise, Editor, et al. RFC 7011: Specification of the ip flow information export (ipfix) protocol for the exchange of flow information. IETF Network Working Group, Internet RFC 7011, 2013. Section 3.1: Definition of a Flow.

[18] MDN Web Docs. Rest - mdn. https://developer.mozilla.org/es/docs/Glossary/REST. Accedido: 2024-06-29.

[19] Elias Dritsas and Maria Trigka. A survey on cybersecurity in iot. *Future Internet*, 17, 1 2025.

[20] Aurélien Géron. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems*. .º'Reilly Media, Inc.", 2022.

[21] Eric Gyamfi and Anca Jurcut. Intrusion detection in internet of things systems: A review on design approaches leveraging multi-access edge computing, machine learning, and datasets. *Sensors 2022, Vol. 22, Page 3744*, 22:3744, 5 2022.

[22] Hamza Kheddar. Transformers and large language models for efficient intrusion detection systems: A comprehensive survey. *Information Fusion*, 124:103347, 12 2025.

[23] Arash Habibi Lashkari et al. Cicflowmeter (iscxflowmeter) V4.0: A network traffic flow generator and analyzer. https://github.com/ISCX/CICFlowMeter, 2018. Canadian Institute for Cybersecurity, UNB.

[24] Jozef Mocnej, Adrian Pekar, Winston K.G. Seah, and Iveta Zolotova. Network traffic characteristics of the iot application use cases. Technical Report ECSTR18-01, School of Engineering and Computer Science, Victoria University of Wellington, 2018. Technical Report.

[25] Euclides Carlos Pinto Neto, Sajjad Dadkhah, Raphael Ferreira, Alireza Zohourian, Rongxing Lu, and Ali A. Ghorbani. Ciciot2023: A real-time dataset and benchmark for large-scale attacks in iot environment. *Sensors 2023, Vol. 23, Page 5941*, 23:5941, 6 2023.

[26] Michael A Nielsen. *Neural networks and deep learning*, volume 25. Determination press San Francisco, CA, USA, 2015.

[27] Giovanni Pau, Xiangjie Kong, Brunel Rolack Kikissagbe, and Meddi Adda. Machine learning-based intrusion detection methods in iot systems: A comprehensive review. *Electronics 2024, Vol. 13, Page 3601*, 13:3601, 9 2024.

[28] Shu Ming Tseng, Yan Qi Wang, and Yung Chung Wang. Multi-class intrusion detection based on transformer for iot networks using cic-iot-2023 dataset. *Future Internet 2024, Vol. 16, Page 284*, 16:284, 8 2024.

[29] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems (NIPS 2017)*, pages 5998–6008, 2017.

[30] Hongping Yan, Liukun He, Xinyu Song, Weijia Yao, Chao Li, and Qiao Zhou. Bidirectional statistical feature extraction based on time window for Tor flow classification. *Symmetry*, 14(10):2002, 2022.

[31] Ya Zhang, Ravie Chandren Muniyandi, and Faizan Qamar. A review of deep learning applications in intrusion detection systems: Overcoming challenges in spatiotemporal feature extraction and data imbalance. *Applied Sciences 2025, Vol. 15, Page 1552*, 15:1552, 2 2025.