UNIVERSIDAD DE BURGOS
ESCUELA POLITÉCNICA SUPERIOR
Grado en Ingeniería Informática

TFG del Grado en Ingeniería Informática

Detección de malware mediante técnicas de Inteligencia Artificial

Presentado por David Cezar Toderas
en Universidad de Burgos — 24 de junio de 2025
Tutor: Alvar Arnaiz Gonzalez

UNIVERSIDAD DE BURGOS
ESCUELA POLITÉCNICA SUPERIOR
Grado en Ingeniería Informática

D. Alvar Arnaiz Gonzalez, profesor del departamento de nombre departamento, área de nombre área.

Expone:

Que el alumno D. David Cezar Toderas, con DNI X8300130M, ha realizado el Trabajo final de Grado en Ingeniería Informática titulado título de TFG.

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

En Burgos, 24 de junio de 2025

Vº. Bº. del Tutor:

Resumen

En este primer apartado se hace una **breve** presentación del tema que se aborda en el proyecto.

Descriptores

Palabras separadas por comas que identifiquen el contenido del proyecto Ej: servidor web, buscador de vuelos, android ...

Abstract

A **brief** presentation of the topic addressed in the project.

Keywords

keywords separated by commas.

Índice general

Índice de figuras

Índice de tablas

1. Introducción

Intro

1.1. Estructura de la memoria

El trabajo se divide en dos partes principales, una memoria (este documento) con la explicación general de lo realizado y, un documento adicional de anexos que contiene diferentes apéndices relativos al proyecto que indagan en cuestiones más específicas acerca del mismo. La estructura de ambos documentos puede verse reflejada a continuación:

Memoria

La memoria (este documento) sigue la siguiente estructura:

1. **Introducción:** Presenta el proyecto, su contexto y motivación, y describe la estructura del documento.

2. **Objetivos del proyecto:** Detalla las metas específicas que el proyecto busca alcanzar, tanto a nivel de funcionalidades como de requisitos técnicos.

3. **Conceptos teóricos:** Establece el marco teórico necesario para poder comprender el documento.

4. **Técnicas y herramientas:** Describe el conjunto de librerías y otras tecnologías que se han empleado para desarrollar el proyecto.

5. **Aspectos relevantes del desarrollo del proyecto:** Narra el proceso realizado a lo largo del trabajo, explicando las fases, las decisiones tomadas y cómo se llegó a la solución.

6. **Trabajos relacionados:** Análisis del estado del arte relacionado con el proyecto.

7. **Conclusiones y líneas de trabajo futuras:** Reflexiones e ideas finales acerca del proyecto y aspectos a mejorar en un futuro.

Anexos

El documento de anexos tiene los siguientes apéndices:

A **Plan de proyecto software:** Contiene la planificación del proyecto y los estudios de viabilidad de este.

B **Especificación de requisitos:** Expone los requisitos funcionales y no funcionales del sistema, catálogo de requisitos y especificaciones adicionales.

C **Especificación de diseño:** Describe el diseño de datos y arquitectónico del proyecto.

D **Documentación técnica de programación:** Guía destinada a los desarrolladores que explica la estructura del código, las decisiones de implementación y los pasos para configurar el entorno de desarrollo.

E **Documentación de usuario:** Manual de instrucciones dirigido al usuario final, que explica cómo utilizar la aplicación y sus funcionalidades.

Recursos adicionales

Además de la memoria y los anexos, se proporciona el enlace a la aplicación creada y al repositorio del proyecto.

1. **Aplicación web:**

2. **Repositorio del proyecto:**

2. Objetivos del proyecto

Este apartado explica de forma precisa y concisa cuáles son los objetivos que se persiguen con la realización del proyecto. Se puede distinguir entre los objetivos marcados por los requisitos del software a construir y los objetivos de carácter técnico que plantea a la hora de llevar a la práctica el proyecto.

2.1. Objetivos funcionales

Los objetivos funcionales definen las capacidades y acciones que el sistema final debe ofrecer al usuario. Son, en esencia, las características con las que se podrá interactuar directamente.

- **Clasificación de aplicaciones Android:** El sistema debe ser capaz de recibir un archivo APK y analizarlo para determinar si es malicioso o benigno, devolviendo un resultado claro.

- **Desarrollo de una interfaz web de demostración:** Crear una aplicación web simple e intuitiva que permita a un usuario subir un archivo APK y obtener la predicción del modelo de forma visual.

2.2. Objetivos técnicos

Los objetivos técnicos se refieren a las metas de implementación, arquitectura y proceso necesarias para construir el sistema y asegurar su calidad y funcionalidad interna.

- **Investigación del estado del arte:** Realizar un análisis de la literatura científica existente para comprender las técnicas actuales de detección de *malware* con IA y posicionar el proyecto.

- **Creación de un dataset propio:** Diseñar y construir un conjunto de datos a medida, extrayendo características útiles de una cantidad decente de ficheros APK.

- **Implementación de un *pipeline* de datos:** Desarrollar un proceso completo y replicable que transforme los datos brutos de un APK en el formato numérico que el modelo necesita para su análisis.

- **Diseño y entrenamiento del modelo de red neuronal:** Construir un modelo de *deep learning* con PyTorch, entrenarlo con el *dataset* propio y optimizar su rendimiento para que sea capaz de clasificar *malware* correctamente.

- **Ajuste de hiperparámetros:** Utilizar herramientas de búsqueda automatizada como Optuna para encontrar la configuración de hiperparámetros que ofrezca el mejor equilibrio entre rendimiento y tiempo de entrenamiento.

- **Análisis comparativo de modelos:** Entrenar clasificadores clásicos de aprendizaje automático (como KNN, SVM, Linear Regression, Random Forest o XGBoost) usando las características procesadas por la red neuronal y comparar sus resultados.

2.3. Objetivos personales

Estos objetivos describen las competencias y conocimientos que se esperan adquirir a nivel personal durante el desarrollo del proyecto.

- **Aplicar conocimientos de IA en un proyecto real:** Llevar la teoría a la práctica, diseñando, entrenando y evaluando un modelo de red neuronal desde cero con Python.

- **Adquirir experiencia en investigación académica:** Aprender a buscar, leer y sintetizar documentación científica, tanto para fundamentar las decisiones técnicas del proyecto, como para mejorar mi compresión a la hora de leer estudios de cualquier tipo.

- **Profundizar en IA y ciberseguridad:** Ganar un mejor entendimiento acerca de estos dos grandes campos que hoy en día dominan el mundo de la informática. Comprender a su vez mejor cómo funciona el *malware* y las técnicas que pueden usarse para detectarlo y combatirlo.
- **Obtener un modelo útil y con un buen rendimiento:** Ser capaz de desarrollar un modelo que pueda competir con otros o, que al menos pueda defenderse. El objetivo principal en este caso es obtener algo que sea capaz de no fallar mucho y que dé predicciones bastante fiables.

3. Conceptos teóricos

En esta sección se definen aquellos conceptos que son necesarios conocer para comprender el resto del documento.

3.1. Conceptos de IA

Para comprender cómo una máquina puede aprender a identificar amenazas, primero debemos sentar las bases de la inteligencia artificial. Este apartado introduce las ideas fundamentales que hacen posible este proyecto, comenzando por el concepto general de la IA, centrándose en el aprendizaje automático como su motor principal. Se explora cómo los modelos computacionales pueden identificar patrones en los datos y el importante proceso de transformar información cruda, como un archivo, en un formato que la máquina pueda entender. El objetivo es ofrecer una visión clara del camino que se sigue para construir y entrenar un modelo de inteligencia artificial desde cero.

Inteligencia artificial

La inteligencia artificial (IA) es una disciplina de la informática que busca desarrollar sistemas capaces de realizar tareas que tradicionalmente asociamos con la inteligencia humana, como el razonamiento, el aprendizaje o la percepción del entorno. Más que intentar replicar la mente humana, un objetivo todavía lejano, la IA se centra en crear herramientas que puedan procesar información de forma autónoma y tomar decisiones para resolver problemas concretos. La IA nos proporciona un conjunto de técnicas para abordar estos desafíos, abriendo la puerta a nuevas formas de automatización y análisis.

El enorme interés que vemos hoy en día por la IA se debe a una combinación de factores que han coincidido en el tiempo: la disponibilidad de cantidades masivas de datos para entrenar los modelos, el diseño de algoritmos de aprendizaje cada vez más eficaces y, sobre todo, el acceso a una gran capacidad de cómputo gracias al abaratamiento de costes y el desarrollo de mejores GPUs. Estas circunstancias han permitido que la IA deje de ser un campo puramente académico para convertirse en una tecnología con aplicaciones en prácticamente cualquier sector.

Gracias a ello, la IA ha impulsado avances en un espectro muy amplio de aplicaciones. En el procesamiento del lenguaje (NLP), ha sido la base para mejorar los traductores automáticos y ha permitido el desarrollo de asistentes virtuales. En el campo de la visión por computador, las técnicas de IA son fundamentales en sistemas de reconocimiento facial o en la asistencia al diagnóstico médico mediante imágenes. Otros campos, como los sistemas de recomendación o la conducción autónoma, son desafíos complejos donde la IA es un componente esencial, aunque su desarrollo y perfeccionamiento siguen siendo un trabajo en curso.

Aprendizaje automático

El aprendizaje automático (*machine learning* o ML) es un subcampo de la inteligencia artificial que se centra en el desarrollo de algoritmos y modelos que permitan a los sistemas informáticos aprender a partir de los datos, en lugar de ser programados explícitamente para realizar una tarea concreta. La premisa fundamental del aprendizaje automático es que, al exponer a un modelo a una cantidad suficientemente grande de ejemplos, este puede identificar patrones, correlaciones y estructuras subyacentes en los datos para, posteriormente, realizar predicciones o tomar decisiones sobre datos nuevos y nunca antes vistos.

El proceso de aprendizaje se puede categorizar principalmente en tres paradigmas. El más común de ellos es el **aprendizaje supervisado**, donde el modelo se entrena con un conjunto de datos etiquetado; es decir, cada ejemplo de entrada está emparejado con una salida, comúnmente conocida como etiqueta, o resultado correcto. El objetivo es que el modelo aprenda una función que sea capaz de mapear las distintas entradas con sus correspondientes salidas correctas. Las tareas de clasificación y regresión son ejemplos típicos de este paradigma. En segundo lugar, el **aprendizaje no supervisado** se enfrenta directamente a datos no etiquetados, y el objetivo del modelo es descubrir patrones o estructuras ocultas por sí mismo, como agrupar datos similares (*clustering*) o reducir la dimensionalidad de los

datos. Por último, el **aprendizaje por refuerzo** implica a un agente que aprende a tomar decisiones interactuando con un entorno; el agente recibe recompensas o penalizaciones en función de sus acciones, y su objetivo es maximizar la recompensa acumulada a lo largo del tiempo.

El aprendizaje automático constituye el motor que impulsa a la mayoría de las aplicaciones modernas de IA, permitiendo a estas adaptarse y ser útiles en multitud de situaciones.

Extracción de características

Un modelo de aprendizaje automático no ”entiende” el mundo como nosotros; no ve una imagen, un texto o un archivo, sino que únicamente opera con números. La extracción de características es el proceso fundamental mediante el cual, se intenta traducir la información del mundo real al lenguaje matemático que el modelo puede procesar. Consiste en identificar y cuantificar las propiedades más representativas de los datos en bruto para convertirlas en un formato numérico y estructurado.

Esta ”traducción” debe capturar la esencia del problema. Por ejemplo, si queremos analizar un programa en busca de *malware*, no le pasamos el archivo binario directamente al modelo. En su lugar, extraemos ”pistas” medibles como el número de permisos que solicita, la presencia de ciertas funciones sospechosas o si intenta conectarse a direcciones de internet conocidas por ser maliciosas. Cada una de estas pistas es una característica que, en conjunto, forma una especie de perfil numérico de la aplicación que el modelo es capaz de aprender mucho mejor que el binario completo.

Sin embargo, no todas las pistas son igual de importantes. Una parte crucial del proceso es la selección de características, que consiste en filtrar el ruido y quedarse solo con la información que realmente aporta valor predictivo. Esto sería el equivalente a un detective que descarta las pistas falsas para centrarse en solo aquellas que le permiten resolver el caso. Este proceso de selección y filtrado culmina en la creación de un *dataset*, una tabla organizada donde cada fila es una muestra y cada columna es una de las características relevantes, lista para que el modelo pueda aprender de ella.

Preprocesamiento de datos

El preprocesamiento de datos es una etapa crucial en cualquier proyecto de aprendizaje automático la cual consiste en aplicar un conjunto técnicas

al *dataset* justo antes del entrenamiento del modelo. Su objetivo principal es limpiar, transformar y adecuar los datos para mejorar la calidad de la información de entrada y, en consecuencia, el rendimiento, la eficiencia y la capacidad de generalización del modelo. Un modelo solo puede ser tan bueno como los datos con los que se entrena, el preprocesamiento es la etapa que intenta que esos datos estén en la mejor forma posible para facilitar el aprendizaje del modelo. A su vez, también es útil para permitir simplificar el modelo internamente y, asegurar que los datos que le llegan a este son siempre de una forma concreta, lo cual permite optimizar enormemente el rendimiento de este.

Las tareas de preprocesamiento son variadas y dependen de la naturaleza de los datos y del modelo que se pretende utilizar. Una de las más fundamentales es la conversión de estos a un formato numérico, ya que, al final, todos los algoritmos no son más que puras matemáticas que operan exclusivamente con números. Esto implica, por ejemplo, transformar texto en una representaciones numéricas mediante técnicas como la tokenización y la creación de vocabularios, o convertir características categóricas (como "tipo de archivo") en valores numéricos mediante métodos como la discretización o el *one-hot encoding*. El resultado final suele ser una matriz numérica.

Otras tareas comunes incluyen:

- **Imputación de valores nulos:** Muchos *datasets* contienen valores faltantes. Estos deben ser tratados, ya sea eliminando las muestras o características afectadas (si son pocas) o, más comúnmente, rellenando los huecos con un valor por defecto o incluso un estadístico como la media, la mediana o la moda de dicha característica.
- **Escalado de características:** Cuando las características numéricas tienen rangos muy diferentes (p. ej., una va de 0 a 1 y otra de 0 a 1.000.000), los algoritmos sensibles a la escala, como las máquinas de soporte vectorial (SVM) o los que usan el descenso de gradiente, pueden verse negativamente afectados. Técnicas como la normalización (escalar a un rango [0, 1]) o la estandarización (centrar en media 0 y desviación estándar 1) resuelven este problema.
- **Discretización:** Consiste en convertir características numéricas continuas en categorías o *bins* que cubren un rango concreto de todos los valores posibles.
- **Filtrado de características:** Aunque a menudo se considera parte de la extracción, la selección final de características puede realizarse

también en esta fase, eliminando aquellas con baja varianza o baja correlación con la variable objetivo.

Clasificador

Un clasificador es un modelo o algoritmo que se utiliza para asignar una categoría o etiqueta a un conjunto de datos, basándose en sus características. En el contexto de la inteligencia artificial, el proceso de clasificación implica entrenar un modelo para que aprenda a predecir la clase o categoría correcta de nuevas observaciones, basándose en ejemplos previos conocidos como datos de entrenamiento.

Clasificadores clásicos (ML)

Los clasificadores clásicos de aprendizaje automático son aquellos modelos que se establecieron como fundamentales en el campo antes de la popularización masiva de las redes neuronales (*deep learning*). Estos algoritmos siguen siendo extremadamente útiles y relevantes, especialmente cuando se trabaja con datos estructurados o tabulares, o cuando la interpretabilidad del modelo es una prioridad, además, suelen requerir menos datos y recursos computacionales que los modelos de *deep learning*. A continuación se describen algunos de los más representativos:

- **Árbol de decisión (*Decision tree*):** Es uno de los modelos más intuitivos. Funciona creando una estructura similar a un diagrama de flujo, donde cada nodo interno representa una pregunta sobre una característica, cada rama representa la respuesta a esa pregunta, y cada nodo hoja representa una etiqueta de clase. Para clasificar una nueva muestra, se la hace ”descender” por el árbol desde la raíz, respondiendo a las preguntas en cada nodo hasta llegar a una hoja, la cual proporciona la predicción. Los árboles aprenden a realizar las mejores preguntas (o ”cortes”) para dividir los datos de la forma más pura posible en cada paso, basándose en métricas como la ganancia de información.
- **k-Vecinos más cercanos (*k-Nearest Neighbors* o k-NN):** A diferencia de otros modelos, k-NN es un algoritmo ”perezoso” (*lazy learner*), ya que no aprende una función como tal durante el entrenamiento. Simplemente, almacena todo el set de datos de entrenamiento y listo. Para clasificar una nueva muestra, busca las k muestras más similares (sus ”vecinos más cercanos”) en el conjunto de entrenamiento,

basándose en una métrica de distancia como puede ser la distancia euclídea, la distancia de Manhattan o la distancia de Humming. La clase de la nueva muestra se asigna por votación mayoritaria entre las clases de esos k vecinos.

- **Máquinas de soporte vectorial (*Support Vector Machines* o SVM):** El objetivo de un SVM es encontrar el hiperplano óptimo que separe las distintas clases que se desean predecir en el espacio de características del set de datos con el mayor margen posible. Dicho "margen" es la distancia entre el hiperplano de decisión y los puntos de datos más cercanos de cada clase, llamados "vectores de soporte". Al maximizar este margen, el modelo logra una mejor capacidad de generalización. Para problemas que no son linealmente separables, los SVM pueden utilizar lo que se conoce como el *kernel trick*, el cual consiste en mapear los datos a un espacio de mayor dimensión donde sí se pueda encontrar un hiperplano que separe las clases correctamente.

- **Regresión logística:** A pesar de su nombre, la regresión logística es un modelo de clasificación lineal estadístico bastante usada y que funciona sorprendentemente bien para lo simple que es. Se basa en estimar la probabilidad de que una muestra pertenezca a una clase determinada. Lo hace aplicando la función logística (o sigmoide) a una combinación lineal de las características de entrada. El resultado es un valor entre 0 y 1, que puede interpretarse como una probabilidad. Un umbral de decisión (típicamente 0.5) se utiliza para asignar la muestra a una de las dos clases.

Ensembles

En el aprendizaje automático, a menudo se cumple el dicho de que "dos cabezas piensan mejor que una". Los ensembles o conjuntos de modelos se basan precisamente en esta idea: en lugar de confiar en la predicción de un único modelo, se combinan las decisiones de múltiples modelos para obtener un resultado final más preciso y fiable. La lógica es que los errores o sesgos individuales de cada modelo tienden a ser compensados por los aciertos de los demás, resultando en una predicción colectiva más sólida.

Existen dos estrategias principales para crear estos "comités" de modelos. El *bagging* consiste en entrenar a varios modelos en paralelo, cada uno con una muestra ligeramente diferente de los datos, y luego agregar sus predicciones (por ejemplo, mediante una votación). Uno de los modelos más usados que siguen este enfoque es el RandomForest, el cual se basa en

aplicar este método de entrenamiento a varios árboles de decisión, debida su naturaleza ”inestable” (pequeños cambios en su entrenamiento cambian drásticamente el árbol final) esto funciona muy bien con ellos, porque los diferentes árboles que se obtienen son muy distintos entre sí, lo cual permite que algunos sean mejores en ciertos aspectos que otros, complementándose y mejorando su rendimiento al ser evaluados en conjunto. Por otro lado, el *boosting* es un proceso secuencial: se entrena un primer modelo y, a continuación, se entrena un segundo modelo que se centra específicamente en corregir los errores del primero, y así sucesivamente. Cada nuevo modelo se especializa aquellos ejemplos difíciles que el modelo anterior no supo clasificar correctamente, generando así una cadena de expertos cada vez mejor entrenados. El mejor modelo que representa esta técnica de entrenamiento es el XGBoost (eXtreme Gradient Boosting), el cual, se basa nuevamente en entrenar árboles de decisión pero de manera secuencial y siguiendo el proceso descrito anteriormente.

Neurona artificial y perceptrón multicapa (MLP)

La neurona artificial [2] es la piedra fundamental que sirve como predecesor a las redes neuronales. Su diseño está inspirado en una neurona biológica, la cual recibe señales a través de sus dendritas, las procesa en el soma y emite una señal a través de su axón si el estímulo acumulado supera un cierto umbral. De forma análoga, una neurona artificial recibe una o más entradas numéricas $(x_1, x_2, ..., x_n)$, a cada una de las cuales se le asigna un peso ajustable $(w_1, w_2, ..., w_n)$. Estos pesos determinan la importancia de cada entrada. La neurona calcula la suma ponderada de sus entradas, a la que se le añade un término de sesgo (*bias*, b). Este resultado agregado pasa a través de una función de activación no lineal, que determina la salida final de la neurona.

El primer modelo formal de una neurona artificial fue el perceptrón [4], desarrollado por Frank Rosenblatt en 1957. Un único perceptrón podía aprender a resolver problemas de clasificación binaria que fueran linealmente separables. Sin embargo, se demostró que un perceptrón simple era incapaz de aprender funciones no lineales tan básicas como la puerta XOR [3], lo que limitó gravemente su aplicabilidad y condujo a un período de menor interés en la investigación de redes neuronales.

La solución a esta limitación llegó con el desarrollo del perceptrón multicapa (*Multilayer Perceptron* o MLP). Un MLP consiste en apilar las neuronas artificiales en múltiples capas: una capa de entrada (*input layer*), que recibe los datos iniciales; una o más capas intermedias u ocultas

(*hidden layers*), que realizan la mayor parte del procesamiento y permiten aprender patrones complejos; y una capa de salida (*output layer*), que produce el resultado final (p. ej., una clasificación o un valor de regresión). Al interconectar las neuronas de esta manera, donde la salida de las neuronas de una capa se convierte en la entrada de las neuronas de la siguiente, la red en su conjunto es capaz de aprender representaciones jerárquicas y aproximar cualquier función continua, superando las limitaciones del perceptrón simple. El MLP es el modelo base de cualquier red neuronal actual y uno de los avances más importantes en el campo.

Red neuronal (*Neural network*)

Una red neuronal es un tipo de modelo de aprendizaje automático inspirado de forma conceptual en cómo funciona el cerebro humano. Está formada por una multitud de pequeñas unidades de procesamiento, llamadas neuronas, que están interconectadas entre sí y organizadas en capas. La idea es que, al igual que en el cerebro, la información se procesa de forma distribuida y en paralelo. Las primeras capas de la red aprenden a detectar patrones muy simples y básicos, y a medida que la información avanza hacia capas más profundas, estas combinan los patrones simples para reconocer conceptos cada vez más complejos y abstractos.

La magia de las redes neuronales reside en su capacidad para aprender automáticamente. Cada conexión entre neuronas tiene un "peso.ªsociado, que determina la importancia de esa conexión. Durante el entrenamiento, la red ajusta continuamente estos millones de pesos para minimizar el error en sus predicciones. Este proceso de ajuste fino, guiado por un algoritmo llamado *backpropagation*, es lo que permite a la red ”aprender” la estructura de los datos. Dependiendo de cómo se organicen estas capas y neuronas, obtenemos diferentes arquitecturas especializadas, como las CNN para imágenes o las RNN para datos secuenciales como las cadenas de texto.

Función de pérdida (*Loss function*)

Para que un modelo pueda aprender, necesita una forma de saber si sus predicciones son correctas o no. La función de pérdida (*loss function*), o función de coste, cumple exactamente este papel, es una fórmula matemática que estima el error que el modelo está cometiendo en sus predicciones. Después de cada predicción que este hace, la función de pérdida la compara con la respuesta correcta y calcula un número que representa qué tan

"equivocado" estaba el modelo. Si el número es alto, el error es grande; si es cercano a cero, indica que la predicción fue muy buena.

Existen diferentes tipos de funciones de pérdida y, su elección depende de la tarea específica a realizar por el modelo. Por ejemplo, para problemas de regresión se suele utilizar el error cuadrático medio (*mean squared error*), mientras que para problemas de clasificación se emplea la entropía cruzada (*cross-entropy*).

A su vez, durante el entrenamiento, es fundamental monitorizar dos métricas de pérdida:

- **Pérdida de entrenamiento (*training loss*):** Es el valor de la función de pérdida calculado sobre el conjunto de datos de entrenamiento. Este valor es el que el algoritmo de optimización intenta minimizar directamente ajustando los parámetros del modelo. Una disminución constante de la pérdida de entrenamiento indica que el modelo está aprendiendo los patrones presentes en los datos de entrenamiento.
- **Pérdida de validación (*validation loss*):** Es el valor de la función de pérdida calculado sobre un conjunto de datos separado, llamado conjunto de validación, que el modelo no utiliza para aprender. Esta métrica es un indicador mucho más fiable de la capacidad de generalización del modelo, es decir, de su rendimiento en datos nuevos y no vistos. Si la pérdida de entrenamiento sigue disminuyendo pero la de validación comienza a aumentar, es una señal clara de sobreajuste (*overfitting*). Esto significa que el modelo ha comenzado a "memorizar" los datos de entrenamiento, incluyendo su ruido, en lugar de aprender los patrones subyacentes, perdiendo así su capacidad para predecir correctamente nuevas muestras.

Sobreajuste (*overfitting*) y subajuste (*underfitting*)

El sobreajuste (*overfitting*) y el subajuste (*underfitting*) son posiblemente los dos problemas más comunes que pueden surgir durante el entrenamiento de modelos de aprendizaje automático, representando extremos opuestos del rendimiento de un modelo. Ambos fenómenos se refieren a la incapacidad del modelo para generalizar adecuadamente a partir de los datos de entrenamiento y, por lo tanto, para realizar predicciones precisas sobre datos nuevos y no vistos.

En concreto, el sobreajuste ocurre cuando un modelo aprende los datos de entrenamiento "demasiado bien", capturando no solo los patrones

subyacentes, sino también el ruido y las fluctuaciones aleatorias específicas del conjunto de datos. Dicho de otra forma, el modelo se vuelve excesivamente complejo y se ”memoriza” los ejemplos de entrenamiento en lugar de aprender a generalizar. La manera en la que esto se manifesta durante el entrenamiento es mediante un rendimiento excelente en los datos de entrenamiento (*training loss* muy baja y que cae constantemente) pero un rendimiento pobre en los datos de validación (*validation loss* alta y que se estanca o empieza a subir tras un punto en el entrenamiento).

Por otro lado, el subajuste sucede cuando un modelo es demasiado simple para capturar la complejidad y los patrones inherentes en los datos. En este caso, el modelo no tiene la capacidad suficiente para aprender relaciones significativas, lo que resulta en un mal rendimiento tanto en el conjunto de entrenamiento como en el de validación. Un modelo subajustado no aprende eficazmente, manifestándose en una *loss* alta para ambos conjuntos de datos.

Descenso de gradiente, aprendizaje y *backpropagation*

Tres conceptos intrínsecamente ligados, los cuales forman el núcleo del mecanismo que permite a las redes ”aprender” de los datos. El proceso de aprendizaje consiste en ajustar iterativamente los parámetros del modelo (pesos y sesgos) para minimizar una función de pérdida.

El descenso de gradiente [1] (*gradient descent*) es el algoritmo de optimización que permite realizar este ajuste. La función de pérdida puede visualizarse como una superficie con valles y colinas en un espacio multidimensional, donde cada punto de la superficie representa un valor de pérdida para una configuración específica de los pesos del modelo. El objetivo es encontrar el punto más bajo de esta superficie (el mínimo global de la pérdida). El ”gradiente” es un vector que apunta en la dirección del máximo crecimiento de la función en un punto dado. Por lo tanto, para ”descender” y minimizar la pérdida, el algoritmo da pequeños pasos en la dirección opuesta al gradiente. El tamaño de estos pasos está controlado por un hiperparámetro llamado tasa de aprendizaje (*learning rate*). Una tasa adecuada es crucial: si es demasiado pequeña, el aprendizaje será muy lento; si es demasiado grande, el algoritmo podría sobrepasar el mínimo y no converger nunca.

Para ahora poder realizar dicho ajuste de los pesos es necesario usar un famoso algoritmo conocido como *backpropagation* [5]. Tras realizar una pasada hacia adelante (*forward pass*) para obtener una predicción y calcular la función de pérdida, *backpropagation* aplica la regla de la cadena del cálculo

para propagar el error hacia atrás, desde la capa de salida hasta la capa de entrada. De esta manera, calcula la contribución de cada peso y sesgo al error total, es decir, el gradiente de la pérdida con respecto a cada parámetro.

El proceso de aprendizaje completo se resume a continuación:

- **Pasada hacia adelante (*forward pass*):** Se introducen los datos en la red y se propagan a través de las capas para generar una predicción.
- **Cálculo de la pérdida:** Se compara la predicción con el valor real usando la función de pérdida para cuantificar el error.
- **Pasada hacia atrás (*backward pass* / *backpropagation*):** Se calcula el gradiente de la función de pérdida con respecto a cada parámetro de la red.
- **Actualización de parámetros:** Se utiliza el descenso de gradiente para ajustar los pesos y sesgos en la dirección opuesta al gradiente, reduciendo así la pérdida.

Este ciclo se repite de forma iterativa con diferentes lotes (*batches*) de datos, permitiendo que la red refine gradualmente sus parámetros y mejore su rendimiento.

Tokenización

Los modelos de inteligencia artificial, en su más puro estado, son modelos matemáticos complejos los cuales consiguen aprender patrones de sus datos de entrada y permiten realizar posteriormente predicciones en base a dichos patrones.

El problema fundamental que esto presenta a la hora de procesar diferentes tipos de datos es principalmente que los modelos matemáticos se basan en el uso de números y la búsqueda de patrones en datos numéricos. Esto implica que si usamos datos que no sean numéricos, como puede ser el texto, caemos en un grave problema puesto que no podemos simplemente aplicar el modelo a este tipo de dato directamente puesto no está preparado para funcionar con el.

Una solución a este problema es emplear el proceso de tokenización, el cual consiste en romper el texto que se obtiene como entrada en unidades más sencillas llamadas tokens que el modelo pueda llegar a entender posteriormente. Este proceso puede realizarse de diferentes manearas, ya sea

partiendo cada palabra en un token individual, partiendo las palabras en subpalabras o incluso procesando cada carácter de manera independiente.

El proceso de tokenización parece bastante trivial a primera vista pero presenta muchos retos como pueden ser los siguientes. Procesar diferentes idiomas se vuelve rápidamente un problema en función del método que se utilice para obtener los tokens, diferentes idiomas permiten contracciones o expresiones especiales las cuales pueden ser interpretadas de diferentes maneras y han de poder ser divididas en tokens que mantengan dicho significado. También es posible que un texto presente errores gramaticales, símbolos especiales o jerga específica de un campo que ha de ser entendida y tokenizada correctamente para mantener su significado.

Vocabulario

Para que un modelo pueda entender los tokens que se obtienen en el paso de la tokenización, es necesario convertir dichos tokens en números de alguna forma. Esto puede realizarse de forma sencilla mediante el uso de un diccionario o vocabulario para el modelo.

El vocabulario del modelo no es más que la colección de todos los tokens que se pretende que el modelo pueda comprender asociados a un valor numérico que siempre será el mismo. De esta forma, el texto de entrada puede ser dividido en tokens, los cuales, a su vez, pueden ser convertidos siempre en el mismo valor numérico mediante el uso de un diccionario. Permitiendo así que el modelo procese texto y pueda encontrar patrones en este.

De forma similar al proceso de la tokenización, la creación y selección de un vocabulario presenta muchos problemas que pueden no ser tan aparentes a primera vista, algunos de ellos son, la extensión del diccionario y el número de tokens que son necesarios en este para poder procesar diferentes idiomas o, la necesidad de poder incluir valores de control que puedan ser usados en casos en los que un token no exista o se le quiera dar una información especial a ciertos tokens en algunas circunstancias.

3.2. Conceptos de ciberseguridad

Antes de poder construir una buena defensa, es imprescindible conocer a fondo la amenaza que se pretende combatir. Este capítulo se dedica a explorar el mundo del software malicioso o *malware*, el principal adversario que nuestro modelo de inteligencia artificial está diseñado para combatir.

Se comenzará por definir qué es el *malware* y se describirán sus diferentes manifestaciones, desde los virus clásicos hasta el *ransomware* moderno. A continuación, se explicarán las técnicas que se utilizan para inspeccionar programas sospechosos, sentando así las bases para comprender de dónde extraerá nuestro modelo la información para tomar sus decisiones.

Malware

El término *malware*, (proveniente de *malicious software*) o software malicioso, se refiere a cualquier programa, código o script diseñado con el propósito explícito de causar daño, comprometer la seguridad, o realizar actividades no autorizadas en un sistema informático, una red o un dispositivo. Coloquialmente, se utiliza el término *malware* para referirse a una amplia gama de diferentes subcategorías de programas que puedan considerarse dañinos, cada una con características, formas de ataque y objetivos específicos, pero todas compartiendo la intención de perjudicar al usuario, robar información, o tomar el control de aquellos sistemas afectados.

Una clasificación bastante común del *malware* es la siguiente:

Virus

Un virus es considerado el comodín de los programas maliciosos, al menos desde un punto de vista coloquial, ambas palabras son intercambiables, en cambio, desde un punto de vista técnico, la definición de un virus informático es equiparable a su equivalente biológico. Un virus es un tipo de *malware* que requiere de un huésped para poder realizar su función, es decir, suele ir incrustado o adjunto a otros archivos, generalmente, ejecutables o documentos los cuales, una vez abiertos, permiten al virus infectar el sistema y realizar acciones no deseadas. Una característica importante de los virus, es el hecho de que pueden auto replicarse, es decir, una vez infectada una máquina, pueden emplear diferentes métodos y técnicas para propagarse a otras, comúnmente, estas suelen ser mediante el uso de la red o mediante el uso de medios físicos extraibles. Los virus suelen ser detectados mediante firmas específicas, aunque las técnicas de ofuscación y polimorfismo pueden dificultar su identificación.

Gusano (*worm*)

Un gusano es un tipo de *malware* diseñado para propagarse automáticamente a través de las redes informáticas, explotando las vulnerabilidades de los diferentes sistemas o utilizando técnicas de ingeniería social para

engañar a los usuarios. A diferencia de los virus, los gusanos no necesitan de un huésped al cual adjuntarse para poder desempeñar su función y para auto replicarse, puesto que pueden propagarse de manera independiente. Su principal objetivo es infectar tantos dispositivos como sea posible, lo que puede resultar en la saturación de las redes, la degradación del rendimiento del sistema, o la creación de puertas traseras para permitir el paso a otros tipos de *malware*. Los gusanos son particularmente peligrosos en entornos corporativos, donde pueden propagarse rápidamente a través de las redes internas de una empresa o una organización.

Troyano (*trojan*)

Un troyano es un tipo de *malware* que se hace pasar por un *software* legítimo o útil para engañar a los usuarios y lograr su ejecución. Una vez instalado, un troyano permite a un atacante acceder o controlar el sistema infectado de manera remota, sin el conocimiento del usuario. Los troyanos no se replican por sí mismos, pero pueden realizar gran variedad de acciones maliciosas, principalmente, robar información confidencial, instalar otros tipos de *malware*, o convertir el sistema en parte de una *botnet*. Su nombre proviene del mito del Caballo de Troya, perteneciente a la mitología griega, ya que, al igual que sucede en la historia, el troyano parece inofensivo en un principio pero oculta una gran amenaza en su interior.

Ransomware

El *ransomware* es un tipo de *malware* diseñado para cifrar los archivos del usuario o bloquear el acceso al sistema, exigiendo un rescate, *ransom*, generalmente en criptomonedas, a cambio de restaurar el acceso. Este tipo de *malware* ha ganado mucha popularidad en los últimos años debido a su impacto devastador en individuos, empresas e incluso instituciones gubernamentales. El *ransomware* suele propagarse a través de correos electrónicos de *phishing*, descargas maliciosas o *exploits* de vulnerabilidades. Una vez este es activado, el comportamiento más típico consiste en mostrar al usuario un mensaje (*ransom note*) con las instrucciones para pagar el rescate. Es importante destacar que, aunque se pague la tasa correspondiente al rescate dentro del tiempo establecido, no hay ninguna garantía de que los atacantes cumplan con su promesa de desbloquear los archivos. Es por ello, por lo que, este tipo de *malware* es extremadamente peligroso puesto que juega con el factor de perder un recurso muy preciado, como puede ser la información, además de utilizar la desesperación de los usuarios en su contra.

Spyware o *Info stealer*

El *spyware* es un tipo de *malware* diseñado para recopilar información del usuario sin su consentimiento. Esta información puede incluir contraseñas, datos bancarios, historiales de navegación, métodos de entrada (como pueden ser las pulsaciones de las teclas de un teclado) o cualquier otro dato sensible que pueda posteriormente ser vendido o utilizado en contra de los usuarios. El *spyware* suele operar de manera sigilosa, sin mostrar signos evidentes de su presencia, lo que dificulta su detección.

En términos generales, existen dos variantes de *spyware*, la primera de ellas se instala en el sistema de forma permanente y siempre está activa, monitorizando la actividad del usuario de manera constante, mandando cualquier información sensible que este use, vea o teclee a un servidor externo para que los atacantes la puedan utilizar o vender, la única ventaja que presenta esta variante es que deja rastro y es más sencilla de detectar. Por otro lado, la segunda variante simplemente se ejecuta una vez, recopila toda la información que pueda y luego se borra a si misma para no dejar ni el más mínimo rastro de su ejecución. Para muchos, esta segunda variante es considerada incluso más peligrosa que la primera puesto que la víctima puede tardar semanas, meses o incluso años en darse cuenta de que su información a sido comprometida.

Además de robar información, algunos tipos de *spyware* pueden modificar la configuración del sistema, instalando *software* adicional o redirigiendo el tráfico de red, generalmente, para evitar su detección. Este tipo de *malware* es comúnmente distribuido a través de descargas no autorizadas, correos electrónicos de *phishing*, o *software* gratuito (*freeware*) que incluye componentes ocultos.

Adware

El *adware* es un tipo de *software* que muestra publicidad no deseada, a menudo de manera intrusiva, en el dispositivo del usuario. Aunque no siempre es malicioso, el *adware* puede ser molesto y afectar negativamente a la experiencia del usuario. En algunos casos, el *adware* incluye funcionalidades adicionales para rastrear el comportamiento del usuario y mostrar anuncios personalizados, lo que puede considerarse una violación de la privacidad. El *adware* suele distribuirse junto con *software* gratuito, y los usuarios pueden instalarlo sin darse cuenta al aceptar los términos y condiciones sin leerlos detenidamente.

PUP (*Potentially Unwanted Program* o Programa Potencialmente no Deseado)

Un PUP (*Potentially Unwanted Program*, por sus siglas en inglés) es un tipo de *software* que, aunque no es necesariamente malicioso, puede ser considerado no deseado por el usuario. Los PUPs incluyen aplicaciones como barras de herramientas (*toolbars*), optimizadores de sistema, o *software* de publicidad que se instalan sin el consentimiento explícito del usuario. Aunque no siempre son dañinos, los PUPs pueden ralentizar el sistema, mostrar anuncios no deseados, o recopilar información acerca del usuario. Muchos antivirus y soluciones de seguridad clasifican los PUPs como una categoría separada de *malware*, ya que su impacto puede variar desde simplemente molesto hasta potencialmente peligroso.

Rootkit

Un *rootkit* es un conjunto de herramientas o *software* diseñado para otorgar a un atacante acceso privilegiado y persistente a un sistema, mientras oculta su presencia tanto del usuario como de cualquier *software* de seguridad que pueda estar presente en el sistema. Los *rootkits* suelen operar a nivel de *kernel* (conocido como Ring 0 en muchos casos) o del sistema operativo, lo que les permite manipular cualquier funcionalidad del sistema, incluso aquellas de las que el usuario posiblemente desconoce puesto que están ocultas por el sistema operativo para facilitar su uso. Esto permite a los *rootkits* ser uno de los *malware* más peligrosos debido a que pueden evadir la gran mayoría de soluciones de seguridad y antivirus puesto que operan con los mismos privilegios que estos o incluso más altos. Una vez instalado, un *rootkit* puede ser utilizado para instalar otros tipos de *malware*, robar información, o convertir el sistema en parte de una *botnet*. Debido a su capacidad para ocultarse, los *rootkits* son particularmente difíciles de detectar y eliminar, y a menudo requieren herramientas especializadas o en muchos casos, la reinstalación completa del sistema para poder deshacerse de ellos.

Botnet

Una *botnet* es una red de dispositivos infectados (llamados *bots* o *zombies*) controlados de manera remota por un atacante, conocido como *botmaster*. Los dispositivos infectados pueden incluir computadoras, servidores, dispositivos IoT, y otros equipos conectados a internet. Las *botnets* son utilizadas para realizar una variedad de tareas maliciosas, como pueden ser, ataques de denegación de servicio distribuido (DDoS), envío masivo de correos no deseados (*spam*), minería de criptomonedas, o robo de información masivo.

Los dispositivos infectados suelen ser controlados a través de un servidor externo, y los usuarios generalmente no son conscientes de que su dispositivo forma parte de una *botnet*. La creación y gestión de *botnets* es una de las actividades que más dinero genera para los ciberdelincuentes, ya que les permite llevar a cabo ataques a gran escala con un impacto muy significativo.

Análisis estático

Técnica de detección de *malware* que se realiza sin la necesidad de ejecutar el programa en cuestión. Este método se basa en la obtención, inspección y evaluación de las características que se pueden extraer de un archivo binario, tales como su estructura, código fuente (si está disponible), indicios de obfuscación u otras técnicas de ocultación, cadenas de texto incrustadas en este, firmas digitales, huella digital *hash o signature* del archivo, secuencias de bytes concretas, cabeceras del programa, metadatos incrustados, desensamblado del ejecutable y otras propiedades que pueden ser extraídas directamente del archivo. Las ventajas que este enfoque presenta son, su simplicidad, rapidez y bajo coste computacional, ya que no requiere de entornos de ejecución específicos ni de hardware especializado para probar el comportamineto del programa. Sin embargo, el mayor problema de este tipo de análisis es su dificultad para detectar malware que utiliza técnicas avanzadas de ofuscación o cifrado, ya que estas prácticas dificultan la extracción de información útil del binario.

Análisis dinámico

Técnica de detección de *malware* que consiste en evaluar el comportamiento de un programa mediante su ejecución en un entorno controlado, con el objetivo de observar sus interacciones con el sistema operativo, los recursos de este y otros programas. En este enfoque, se monitorean actividades como la modificación de archivos, el tráfico de red generado, la creación de procesos o la inyección de código en estos, lo cual permite identificar patrones de comportamiento asociados con programas maliciosos. A diferencia del análisis estático, el análisis dinámico ofrece una mayor precisión, ya que puede detectar comportamientos maliciosos que no son evidentes simplemente escanenado el archivo de manera estática, como el uso de técnicas de ofuscación. Sin embargo, este tipo de análisis sigue teniendo sus incovenientes, por un lado, es más complejo, requiere de más recursos computacionales y es más costoso de implementar, dado que involucra la ejecución real del código en un entorno controlado, generalmente una máquina virtual (*sandbox*). Por otro lado, también es poco eficiente contra casos en los que el *malware*

detecta el hecho de que está siendo analizado y oculta su comportamiento malicioso. Además, puede no ser adecuado para dispositivos con recursos limitados, como dispositivos IoT o móviles, debido a sus altos requerimientos de hardware y tiempo.

Análisis híbrido

Metodo de detección de *malware* el cual combina las fortalezas tanto del análisis estático como del dinámico. En este método, el programa se ejecuta en un entorno controlado, y durante su ejecución, se realizan *dumps* de memoria de manera periódica o en respuesta a comportamientos sospechosos. Estos volcados de memoria son posteriormente analizados utilizando técnicas de análisis estático para identificar posibles patrones maliciosos, tales como la inyección de código en procesos ajenos, manipulación de memoria que no le pertenece al programa o modificaciones en partes protegidas de la memoria pertenecientes al sistema oeprativo. Este enfoque permite una detección más precisa de *malware* que utiliza técnicas avanzadas de ocultamiento, ya que combina la observación del comportamiento en tiempo real con la inspección detallada del estado de la memoria. Sin embargo, el análisis híbrido es el más complejo y costoso de implementar, ya que requiere tanto de infraestructura de virtualización como de herramientas para realizar un buen análisis de memoria. A pesar de todo, suele ofrecer los mejores resultados en términos de detección.

Huella digital (*fingerprinting*)

El fingerprinting o, la generación de huellas digitales de archivos, es una técnica utilizada para identificar de manera única un archivo mediante la aplicación de funciones criptográficas de *hashing*. Este proceso consiste en calcular un *hash* a partir del contenido completo del archivo utilizando algoritmos como MD5, SHA-1, SHA-256 u otros. El resultado es una cadena de longitud fija que actúa como un identificador único para ese archivo. Cualquier modificación, por mínima que sea, en el contenido del archivo resultará en un *hash* completamente diferente, lo que permite detectar alteraciones o corrupciones en estos.

Esta técnica es muy utilizada en la verificación de la integridad de archivos, la detección de duplicados, y la identificación de malware conocido al comparar el *hash* que este genera con una base de datos de muestras previamente catalogadas. Sin embargo, una limitación importante del *fingerprinting* es su sensibilidad extrema a cambios mínimos, lo que dificulta

la identificación de archivos que han sido ligeramente modificados pero que conservan una estructura o funcionalidad. Esto implica que incluso cambiar un bit en el *padding* del archivo, hace que este ya no se detecte como malware al tener una huella digital diferente.

Huella digital difusa (*fuzzy hashing*)

El *fuzzy hashing*, o *hashing* difuso, es una técnica que extiende el concepto del *hashing* tradicional al permitir la comparación de archivos basada en similitudes parciales en lugar de una coincidencia exacta. A diferencia del *hashing* convencional, que opera sobre el archivo completo, el *fuzzy hashing* divide el archivo en bloques o segmentos y calcula un *hash* para cada uno de ellos. Este enfoque por bloques permite identificar similitudes entre archivos incluso cuando solo una porción de su contenido ha sido modificada.

El *fuzzy hashing* es particularmente útil en el análisis forense digital y la detección de *malware*, ya que permite identificar variantes de archivos maliciosos que han sido modificaods para evadir su detección, pero que conservan partes significativas de su código original. Al comparar dos *hashes* difusos, es posible calcular un grado de similitud basado en la cantidad de bloques que coinciden entre ambos. Esto se logra mediante algoritmos especializados como SSDeep o TLSH, que están diseñados para generar *hashes* difusos y medir la similitud entre ellos.

3.3. Otros conceptos

Dadas las bases anteriores necesarias para entender el proyecto, se dejan a continuación aquellos conceptos adicionales que pueden ser de gran ayuda para comprender mejor otros aspectos relevantes del proyecto.

Framework web

Un *framework* web es un conjunto de herramientas, bibliotecas y componentes predefinidos que facilitan el desarrollo de aplicaciones y páginas web. Su propósito es ofrecer una estructura básica que los desarrolladores pueden utilizar para crear aplicaciones de manera más eficiente, sin tener que comenzar desde cero. Los *frameworks* web incluyen funcionalidades que resuelven tareas comunes, como el manejo de rutas (URLs), la conexión a bases de datos, la autenticación de usuarios y la seguridad, lo que ahorra tiempo y esfuerzo durante el desarrollo.

Además, proporcionan una organización estructurada para el código, lo que facilita la colaboración entre desarrolladores y simplifica el mantenimiento de los proyectos a largo plazo. Muchos *frameworks* también incluyen mecanismos de seguridad incorporados para proteger las aplicaciones contra amenazas comunes y herramientas que automatizan tareas repetitivas, como la gestión de dependencias y la ejecución de pruebas. Todo esto permite construir aplicaciones web escalables, seguras y fáciles de mantener.

Algunos *frameworks* web populares son: Django (para Python), Ruby on Rails (para Ruby), Angular y React (para JavaScript), y Laravel (para PHP).

4. Técnicas y herramientas

En este apartado se comentan y analizan las diferentes herramientas usadas en la realización de este proyecto.

4.1. Lenguajes y añadidos

Python

Poetry

Poetry es una herramienta de gestión de dependencias y empaquetado para Python. A diferencia de los métodos tradicionales que combinan herramientas como *pip*, *requirements.txt* y *virtualenv*, Poetry integra todas estas funcionalidades bajo una única interfaz de comandos y un archivo de configuración llamado pyproject.toml. Dicho archivo puede ser modificado para declarar las dependencias del proyecto, características del paquete final, versiones y otro tipo de configuraciones adicionales para la generación de distribuciones y reglas de resolución de paquetes. Poetry se encargará posteriormente de generar a partir de dicho archivo de configuración un fichero llamado *poetry.lock*, el cual contiene todas las versiones exactas de cada paquete y sus dependencias correspondientes, correctamente resueltas y guardadas en un formato que permitirá posteriormente reproducir el entorno de forma simple y directa.

En este caso, Poetry no solo es útil para proyectos grandes sino que es extremadamente cómodo para cualquier proyecto de Python que necesite de un par de dependencias, puesto que permite crear entornos virtuales con solo un par de comandos, permitiendo así no modificar la instalación global de Python en el sistema y permitiendo una gestión más compleja de ciertas

dependencias en función de las prestaciones del equipo que uno posee o su sistema operativo. Por ejemplo, en este proyecto, una de las librerías usadas es PyTorch, la cual cuenta con soporte para aceleración por GPU de sus operaciones, pero, este soporte viene incluido en un paquete diferente al del paquete que solo permite el uso de la CPU. Usando Poetry es posible crear una configuración dinámica que sea capaz de adaptarse al equipo de cada uno, descargando o no la versión con o sin aceleración por hardware en función de si el equipo es compatible con ello.

Jupyter Notebook

4.2. Librerías

PyTorch vs Keras

PyTorch y Keras representan dos de los *frameworks* de *deep learning* más conocidos y utilizados en la actualidad. Si bien ambos facilitan la construcción de redes neuronales, cada uno toma una filosofía de diseño distinta, lo cual los hace más adecuados para diferentes tipos de proyectos y usuarios. Keras funciona como una interfaz de alto nivel, diseñada para la simplicidad y el desarrollo rápido, mientras que PyTorch opera a un nivel más bajo, ofreciendo un control granular y una mayor flexibilidad a la hora de desarrollar modelos personalizados y específicos.

Keras se caracteriza principalmente por su facilidad de uso, debido a que permite construir y entrenar modelos estándar con muy pocas líneas de código. Su API es bastante intuitiva y abstrae gran parte de la complejidad subyacente, lo que lo convierte en una opción excelente para principiantes y para la creación rápida de prototipos y para entornos de producción donde la estandarización es clave. Sin embargo, esta simplicidad conlleva una menor flexibilidad, ya que realizar modificaciones sustanciales en la arquitectura o en el ciclo de entrenamiento de los modelos puede volverse complejo y poco intuitivo debido a no estar diseñado para ello.

Por el contrario, PyTorch proporciona un set de herramientas mucho más versátil, pensado para la investigación y para proyectos que requieren de arquitecturas personalizadas. En general, permite crear modelos de forma mucho más granular debido a que se basa en proporcionar al usuario con un conjunto de módulos y funciones que pueden ser instanciadas juntas para formar un modelo complejo y personalizado, especializado en la tarea que se desee. A su vez, permite definir de manera más concreta el set de datos a

usar, el preprocesado de esos datos y el cómo se entrena el modelo a partir de ellos.

En concreto, para este proyecto, PyTorch fue la elección más lógica. La arquitectura del modelo de clasificación de APKs no es convencional; se procesan un montón de cadenas de caracteres y vectores, los cuales requieren de un preprocesado específico y de la implementación de un *embedder* personalizado para poder usar posteriormente dichas características junto con una lógica y control sobre el proceso de entrenamiento bastante concreto.

Características	**PyTorch**
Flexibilidad	Alta, permite crear redes neuronales personalizadas y complejas.
Facilidad de uso	Requiere más código y tiene una curva de aprendizaje más pronunciada
Personalización	Excelente para redes personalizadas y modelos avanzados.
Comunidad y soporte	Muy popular en la investigación académica y proyectos avanzados.
Uso principal	Investigación, redes neuronales complejas.

Tabla 4.1: Comparativa entre PyTorch y Keras

NumPy, Pandas y Matplotlib

NumPy, Pandas y Matplotlib constituyen la trilogía fundamental de librerías sobre las que se edifica gran parte del ecosistema de paquetes científicos y de data science en Python.

- **NumPy (*Numerical Python*):** Es la librería base para la computación numérica en Python. Proporciona el concepto de ndarray, una estructura de datos para la creación de arrays N-dimensionales eficientes, y un vasto conjunto de funciones matemáticas para operar sobre ellos. Una de las mayores ventajas de NumPy es su velocidad y eficiencia tanto en tiempo, como en espacio al trabajar con grandes sets de datos. Este rendimiento se debe principalmente a que muchas de sus operaciones están implementadas en C y aprovechan la vectorización, permitiendo ejecutar operaciones complejas en arrays completos sin necesidad de bucles explícitos en Python.

- **Pandas:** Construida sobre NumPy, Pandas introduce estructuras de datos de alto nivel, principalmente el DataFrame, una tabla bidimensional heterogénea e indexada, pensada para manejar datos tabulares y series temporales. Facilita enormemente tareas como la lectura y escritura de datos, la limpieza, el filtrado, la agregación y la transformación,

siendo una herramienta muy útil estándar para el preprocesamiento de datos.

- **Matplotlib:** Es la librería de visualización de datos por excelencia en Python. Permite elegir entre un gran repertorio de gráficos comunes como pueden ser los gráficos de barras o histogramas a, la creación de gráficos personalizados, facilitando el trabajo de representar datos complejos y hacerlos agradables a la vista.

Estas tres librerías han sido de gran ayuda en diferentes etapas del proyecto. Pandas ha sido la herramienta principal para estructurar las características extraídas de los archivos APK en un DataFrame limpio y manejable, facilitando todo el preprocesamiento. NumPy ha sido utilizado de forma subyacente por Pandas y PyTorch, y directamente para realizar operaciones numéricas eficientes sobre los datos ya procesados antes de introducirlos en el modelo. Finalmente, Matplotlib ha sido usada para la evaluación del modelo, permitiendo visualizar diferentes aspectos del entrenamiento del modelo y la comparación de este con otros de una manera más visual.

scikit-learn

Optuna

Streamlit

Streamlit es un *framework* de código abierto para Python, diseñado específicamente para la creación y el despliegue rápido de aplicaciones web interactivas para proyectos de *data science* y aprendizaje automático. Su filosofía se basa en la simplicidad radical, permitiendo transformar *scripts* convencionales con código de procesamiento de datos, modelos de IA o simples funciones aisladas en aplicaciones web funcionales con un esfuerzo y conocimiento de desarrollo web mínimos.

A diferencia de *frameworks* web más tradicionales y complejos como Django o Flask, que exigen la gestión de rutas, una organización de los ficheros del proyecto específica, sintaxis inusual, plantillas HTML y lógica de servidor, Streamlit permite construir una interfaz de usuario directamente desde un script normal de Python. Con comandos sencillos, se pueden añadir elementos interactivos como botones, deslizadores, gráficos y, fundamentalmente para este caso, campos para la subida de archivos. Esto acelera drásticamente el ciclo de desarrollo puesto que uno puede simplemente

centrarse en obtener un modelo o set de funcionalidades que son correctas y dan buenos resultados sin preocuparse mucho de cómo se llevará luego esto a una interfaz gráfica o aplicación de escritorio / web puesto que la conversión es muy sencilla en la mayoría de casos.

La finalidad de la aplicación web en este proyecto es ofrecer una demostración tangible y una especie de *demo* o entorno de prueba para que cualquiera pudiera probar el clasificador de *malware* de forma sencilla. El objetivo era crear una interfaz simple donde un usuario pudiera subir un archivo .apk y recibir una predicción de manera inmediata. Streamlit fue la herramienta perfecta para esta tarea, ya que permitió desarrollar esta funcionalidad en cuestión de horas en lugar de días y el resultado es más que suficiente para el alcance deseado.

Androguard y otros analizadores

Androguard es una potente herramienta de código abierto y un paquete de Python, diseñada específicamente para el análisis estático y la ingeniería inversa de aplicaciones de Android (archivos APK). Su función principal es el proceso de diseccionar un archivo APK para extraer información detallada sobre su estructura y contenido sin necesidad de ejecutar la aplicación. Una de las mayores ventajas de Androguard es el hecho de que permite además realizar todo este proceso de extracción de características de forma automatizada puesto que, expone una API de Python bastante simple e intuitiva que proporciona acceso a todos los datos que se pueden obtener de analizar las APKs.

A través de Androguard, es posible acceder a componentes como el *manifest* de la aplicación (AndroidManifest.xml) para analizar permisos y componentes declarados, desensamblar el código Dalvik (DEX) para inspeccionar las clases y los métodos, e incluso, extraer recursos como cadenas de texto o certificados. Aunque existen otras herramientas de análisis como MobSF (que ofrece un entorno más automatizado y visual) o Jadx (un popular descompilador), Androguard destaca por su granularidad, simplicidad de uso y su naturaleza como librería de Python.

Una de las piedras fundamentales de este proyecto es la posibilidad de extraer características de forma estática de APKs y entrenar un modelo con ellas capaz de discernir entre aplicaciones benignas y malignas. La razón principal de su eso es el hecho de funcionar nativamente en Python y permitir la automatización de la creación del *dataset* de entrenamiento del modelo. A

su vez, es el componente que permite analizar muestras nuevas, obteniendo los datos que el modelo espera recibir para realizar una predicción.

4.3. Otras herramientas

Docker

Docker es una plataforma de código abierto que permite automatizar el despliegue, la ejecución, la distribución y la gestión de aplicaciones mediante el uso de la "containerización". Esta herramienta permite empaquetar una aplicación y todas sus dependencias como bibliotecas, herramientas del sistema, código, comandos de ejecución, configuraciones y todo lo que uno pueda necesitar para ejecutar su aplicación en una unidad independiente y aislada llamada contenedor.

El principal problema que Docker intenta resolver es el clásico "en mi equipo funciona", donde una aplicación se ejecuta correctamente en el entorno de un desarrollador pero falla en otro debido a diferencias en la configuración del sistema operativo o en las versiones de las dependencias, además de posibles errores en la ejecución del proyecto y diversas posibles causas. Un contenedor soluciona esto debido a que, uno puede simplemente distribuir un único archivo que describe el proceso de creación del contenedor llamado *Dockerfile* (o la misma imagen del contenedor ya creado) garantizando que cualquiera podrá ejecutar el proyecto exactamente de la misma manera en cualquier máquina que soporte Docker, desde un portátil local hasta un servidor en la nube. Esto asegura la consistencia, la reproducibilidad y simplifica enormemente los flujos de trabajo desarrollo.

GitHub

VSCode

TeXstudio

5. Aspectos relevantes del desarrollo del proyecto

5.1. Resumen del trabajo

Este trabajo aborda la creación de una herramienta automatizada para la clasificación de *malware* en Android mediante análisis estático. La estrategia consiste en examinar las aplicaciones de forma segura, sin ejecutarlas, para que un modelo de inteligencia artificial evalúe sus características y determine si son o no maliciosas. El proyecto se desarrolló en dos fases diferentes: una inicial de prototipado y una segunda de desarrollo y refinamiento de la solución final.

La primera fase exploró el estado del arte y validó, mediante un prototipo de red neuronal, que es posible distinguir aplicaciones benignas de malignas usando únicamente sus características estáticas. Sin embargo, este primer modelo se entrenó con un *dataset* preexistente cuyo proceso de creación era una "caja negra", lo que puso de manifiesto un grave problema, la incapacidad de aplicar el modelo a nuevas muestras por no saber cómo preparar los datos de entrada.

Para resolver este problema, la segunda fase se centró en la creación de un *dataset* propio, desarrollando un extractor de características replicable con Androguard. Sobre esta nueva base, se construyó y optimizó un nuevo modelo que, al ser evaluado, reveló un hallazgo interesante. Si bien el rendimiento de la red neuronal es excelente, su mayor valor reside en su *embedder*, el componente que transforma los datos en una representación que la red puede entender. Este demostró ser tan eficaz que permitió entrenar otros clasificadores más simples y ligeros con resultados igualmente buenos

o incluso mejores que los del modelo principal, convirtiendo a la red en una herramienta muy útil para habilitar la creación de otros modelos que cumplan con este propósito de detectar *malware*.

5.2. Pruebas iniciales y estado del arte

Base del proyecto

El dataset de Drebin

Creación de un modelo prototipo

Entrenamiento del modelo

Análisis de datos preliminar

5.3. Extracción de caracteísticas y *dataset* propio

Pruebas con Androguard

Automatización del proceso de creación del *dataset*

5.4. Desarrollo y optimización del modelo final

Adaptación del modelo al nuevo set de datos

Problemas imprevistos

Búsqueda de hiperparámetros óptimos

Análisis de los resultados

Cuantización del modelo

5.5. Desarrollo de la aplicación web

Creación de la web

Dockerización y despliegue

6. Trabajos relacionados

Este apartado sería parecido a un estado del arte de una tesis o tesina. En un trabajo final grado no parece obligada su presencia, aunque se puede dejar a juicio del tutor el incluir un pequeño resumen comentado de los trabajos y proyectos ya realizados en el campo del proyecto en curso.

7. Conclusiones y Líneas de trabajo futuras

Todo proyecto debe incluir las conclusiones que se derivan de su desarrollo. Éstas pueden ser de diferente índole, dependiendo de la tipología del proyecto, pero normalmente van a estar presentes un conjunto de conclusiones relacionadas con los resultados del proyecto y un conjunto de conclusiones técnicas. Además, resulta muy útil realizar un informe crítico indicando cómo se puede mejorar el proyecto, o cómo se puede continuar trabajando en la línea del proyecto realizado.

Bibliografía

[1] Shunichi Amari. A theory of adaptive pattern classifiers. *IEEE Transactions on Electronic Computers*, (3):299–307, 2006.

[2] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5:115–133, 1943.

[3] Marvin Minsky and Seymour A Papert. *Perceptrons, reissue of the 1988 expanded edition with a new foreword by Léon Bottou: an introduction to computational geometry*. MIT press, 2017.

[4] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[5] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[6] Wikipedia. Latex — wikipedia, la enciclopedia libre. https://es.wikipedia.org/w/index.php?title=LaTeX&oldid=84209252, 2015. [Internet; descargado 30-septiembre-2015].