



ETS
Ingeniería de
Telecomunicación

UNIVERSIDAD POLITÉCNICA DE CARTAGENA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

Grado en Ingeniería Telemática

Herramienta para la verificación automática de URLs mediante machine learning en el contexto de ciberseguridad

TRABAJO FIN DE GRADO

Autor:
Ángel Truque Contreras

Director:
Javier Vales Alonso

CARTAGENA, Diciembre de 2025



Universidad
Politécnica
de Cartagena

RESUMEN

Este Trabajo Fin de Grado (TFG) aborda el diseño y la implementación de una herramienta para la verificación automática de Uniform Resource Locator (URL)s maliciosas (phishing) dentro del contexto de la ciberseguridad, utilizando modelos de Machine Learning (ML) como Random Forest. La investigación se centra inicialmente en la creación de modelos discriminadores de ML, entrenados mediante datasets externos que proporcionan diversas características para la clasificación binaria de URLs como legítimas o de phishing.

El trabajo profundiza en la identificación y explotación de una característica altamente determinante observada en el dataset: la presencia o ausencia del favicon en la URL. Basándose en el hallazgo de que la ausencia de favicon está fuertemente correlacionada con un enlace malicioso, la investigación expande su alcance hacia la generación de enlaces falsos para evaluar la robustez de los modelos de detección. Esto incluye ejemplos como <https://login-secure-paypal.com@accounts.biz/update-info>, que ilustran las técnicas de ofuscación. Esta exploración se articula mediante el desarrollo de un generador de URLs sintéticas, lo que permite la implementación y el estudio de Redes Generativas Antagónicas (Generative Adversarial Network (GAN)).

Finalmente, se explora una arquitectura GAN multinivel, que implica el diseño de discriminadores y generadores con diferentes grados de preentrenamiento. La experimentación con estas arquitecturas cruzadas (modelos preentrenados y sin preentrenamiento) no solo permite evaluar la calidad de las URLs de phishing generadas, sino que también ofrece un análisis comparativo de la capacidad de los modelos para detectar amenazas sofisticadas, contribuyendo a la comprensión y mejora de las defensas automatizadas en la seguridad web.

Palabras Clave: Phising, Ciberseguridad, Machine Learning (ML), Random Forest, Discriminadores, Generadores, Redes Generativas Adversariales (GAN), Dataset.

ABSTRACT

This Final Degree Project (TFG) addresses the design and implementation of a tool for the automatic verification of malicious URLs (phishing) within the context of cybersecurity, utilizing Machine Learning (ML) models such as Random Forest. The research initially focuses on the creation of ML discriminator models, trained using external datasets that provide diverse features for the binary classification of URLs as legitimate or phishing.

The work delves into the identification and exploitation of a highly determinant feature observed in the dataset: the presence or absence of the favicon in the URL. Based on the finding that favicon absence is strongly correlated with a malicious link, the research expands its scope toward generating synthetic links to evaluate the robustness of detection models. This includes examples like <https://login-secure-paypal.com@accounts.biz/update-info>, which illustrate obfuscation techniques. This exploration is articulated through the development of a synthetic URL generator, enabling the implementation and study of Generative Adversarial Networks (GAN).

Finally, a multilevel GAN architecture is explored, which involves designing discriminators and generators with different degrees of pretraining. Experimentation with these cross-architectures (pretrained and non-pretrained models) not only allows for evaluating the quality of the generated phishing URLs but also provides a comparative analysis of the models' ability to detect sophisticated threats, thus contributing to the understanding and improvement of automated web security defenses.

Keywords: Phishing, Cybersecurity, Machine Learning (ML), Random Forest, Discriminators, Generators, Generative Adversarial Networks (GAN), Dataset.

LISTA DE ACRÓNIMOS

AdaGrad Adaptive Gradient Algorithm.

Adam Adaptative Moment Estimation.

ANN Redes Neuronales Artificiales.

AUC-ROC Area Under the Receiver Operating Characteristic Curve.

BGR Blue, Green, Red.

BIC Bayesian Information Criterion.

cGAN Generative Adversarial Network Condicional.

CNN Convolutional Neural Network.

DNN Deep Neural Networks.

DNS Domain Name System.

FID Frechet Inception Distance.

FN False Negative (Falso Negativo).

FP False Positive (Falso Positivo).

GAN Generative Adversarial Network.

GB Gradient Boosting.

GMM Gaussian Mixture Model.

GPU Graphic Processing Unit.

HTML HyperText Markup Language.

HTTPS Hypertext Transfer Protocol Secure.

IA Inteligencia Artificial.

IP Internet Protocol.

IS Inception Score.

KNN K-Nearest Neighbors.

LR Logistic Regression.

ML Machine Learning.

MLP Multilayer Perceptron.

ncGAN Generative Adversarial Network No Condicional.

NLP Natural Language Processing.

PCA Principal Component Analysis.

ReLU Unidad Lineal Rectificada.

ResNet Residual Network.

RF Random Forest.

RGB Red, Green, Blue.

RMSProp Root Mean Squared Propagation.

SGD Stochastic Gradient Descent.

SSL Secure Sockets Layer.

SVM Support Vector Machines.

TFG Trabajo Fin de Grado.

TLD Top-Level Domain.

TN True Negative (Verdadero Negativo).

TP True Positive (Verdadero Positivo).

UCI University of California, Irvine.

URL Uniform Resource Locator.

ÍNDICE GENERAL

Resumen	3
Abstract	4
Lista de Acrónimos	5
Índice de figuras	13
Índice de tablas	16
Código	17
1. Introducción	1
1.1. Contexto y motivación	1
1.2. Planteamiento del problema	2
1.3. Objetivos del proyecto	2
1.4. Alcance y limitaciones	3
1.4.1. Alcance	3

1.4.2. Limitaciones	3
1.5. Estructura del documento	4
2. Estado del arte y fundamentos teóricos	6
2.1. Concepto de phishing y técnicas comunes de detección	6
2.2. Características de las URLs relevantes para la detección	7
2.3. Favicons: función y relevancia visual	8
2.4. Aprendizaje automático en detección de phishing	8
2.4.1. Modelos Clásicos sobre Datos Tabulares	8
2.4.2. Modelos Profundos sobre Datos Visuales	9
2.5. Redes convolucionales (Convolutional Neural Network (CNN)) y Transfer Learning con Residual Network (ResNet)	10
2.5.1. Fundamentos de las CNN	10
2.5.2. Transfer Learning con ResNet	11
2.6. Generative Adversarial Networks (GAN)	11
2.6.1. El Marco Antagónico	11
2.6.2. Tipos de GAN en el Proyecto	13
2.7. Métricas y Evaluación de Modelos	13
2.7.1. Métricas de Clasificación Binaria	13
2.7.2. Métricas para la Evaluación de la Calidad de las GAN	15
2.8. Síntesis del Estado del Arte y Carencias Detectadas	15
3. Metodología y Datasets	16
3.1. Enfoque Metodológico General	16

3.2. Datasets Empleados	17
3.2.1. Construcción y Balanceo del Dataset de Favicon	20
3.3. Procesado de Datos Tabulares	24
3.3.1. Detección y Eliminación de Características “Tramposas”	26
3.4. Procesado de Imágenes	28
3.4.1. División de Conjuntos (Train, Validation, Test) y Justificación de Pro- porciones	30
3.5. Herramientas y Entorno de Desarrollo	32
4. Modelos y desarrollo experimental	34
4.1. Modelado Basado en URL	34
4.1.1. Modelo Clásico: Random Forest (RF) y Validación	34
4.1.2. Red Neuronal Artificial (Redes Neuronales Artificiales (ANN)) para Datos Tabulares	36
4.2. Modelado basado en favicons	39
4.2.1. Implementación de una CNN desde Cero	39
4.2.2. Transfer Learning con ResNet	40
4.2.3. Registro del Entrenamiento (<i>history</i>) y Mitigación del Desbalance .	42
4.3. Generación adversarial	43
4.3.1. Diseño de la Red GAN	44
4.3.2. Implementación de la Estrategia M D / M G	44
4.3.3. Implementación de la Estrategia M D / P G	46
4.3.4. Implementación de la Estrategia P D / M G	47

4.3.5. Implementación de la Estrategia M D / M G (Generative Adversarial Network No Condicional (ncGAN))	50
4.4. Protocolo Experimental	51
4.4.1. Parámetros de Entrenamiento	51
4.4.2. Validación Cruzada	52
4.4.3. Control de Reproducibilidad	52
5. Resultados y discusión	54
5.1. Resultados del análisis basado en URL	54
5.1.1. Optimización de la Dimensionalidad y Separabilidad	54
5.1.2. Comparación de Modelos: Random Forest vs. ANN	57
5.1.3. Análisis de Errores (Falsos Positivos y Falsos Negativos) en el Modelo URL	57
5.2. Resultados del análisis basado en favicons (ANN y CNN/ <i>Transfer Learning</i>)	58
5.2.1. Comparación Cuantitativa de Modelos de Imagen	58
5.2.2. Análisis de Errores (Falsos Positivos y Falsos Negativos) en el Modelo Favicon	59
5.3. Resultados del experimento adversarial con GAN	60
5.3.1. Evaluación Cuantitativa de la Calidad de Imagen	61
5.3.2. Conclusión Visual	62
5.4. Comparación global y discusión de resultados	67
5.4.1. El Condicionamiento de Clase como Factor de Estabilización	67
5.4.2. El <i>Transfer Learning</i> en el Discriminador: Factor Crítico	68
5.4.3. Selección de la Estrategia Óptima (PD/MG vs. PD/PG)	68

6. Conclusiones y líneas futuras	70
6.1. Conclusiones Generales	70
6.2. Contribuciones Técnicas y Prácticas	71
6.3. Lecciones Aprendidas, Limitaciones del Estudio	72
6.4. Líneas de Trabajo Futuro	72
Bibliografía	75
Anexos	76
A. Código Fuente del Modelo Generative Adversarial Network Condicional (cGAN) con Gaussian Mixture Model (GMM)	76
B. Catálogo de Características, Arquitecturas e Hiperparámetros	80
B.1. Catálogo de Características Tabulares de URLs	80
B.2. Arquitecturas e Hiperparámetros de Modelos	81
B.2.1. Hiperparámetros Comunes y de RF	81
B.2.2. Arquitectura de Red Neuronal Artificial (ANN)	81
B.2.3. Arquitectura del Generador (G) en la Estrategia PD/PG	82
B.2.4. Arquitectura del Discriminador (D) en Estrategia PD	82
C. Tablas de Resultados Extendidas y Gráficas de Entrenamiento	83
C.1. Resultados de Clasificación Detallados y Matriz de Confusión	83
C.2. Gráficos de Convergencia del Entrenamiento	84
C.2.1. Análisis Detallado de la Convergencia Adversaria (cGAN PD/PG) . .	85
C.3. Análisis Detallado de Varianza (Principal Component Analysis (PCA)) . . .	86

D. Muestras Visuales de Favicon Reales y Sintéticos	87
D.1. Muestras del Dataset Balanceado	87
D.2. Comparación Visual de la Generación Adversarial	88
D.2.1. Mejor Estrategia Perceptual: PD/PG (Frechet Inception Distance (FID): 75.3)	89
D.2.2. Mejor Estrategia Estadística: PD/MG (FID: 68.5)	89
D.2.3. Prueba de Concepto del Muestreo GMM	90

ÍNDICE DE FIGURAS

2.1. Favicon asociado a https://www.youtube.com	8
2.2. Arquitectura típica de una CNN	11
2.3. Arquitectura típica de una GAN	12
2.4. Matriz de confusión. Fuente: Wikipedia (https://en.wikipedia.org/wiki/Confusion_matrix)	14
3.1. Icono asociado a google.com (con transparencia), dimensiones 32x32 y rango [0,255]	28
3.2. Icono asociado a google.com (sin transparencia), dimensiones 64x64 y rango [-1,1]	29
4.1. Gráfica de la función Unidad Lineal Rectificada (ReLU).	37
4.2. Funcionamiento de un Autoencoder	48
5.1. Potencia Espectral del PCA. Varianza acumulada en función del número de componentes principales.	55
5.2. Clasificación predicha en el Espacio PCA (2D).	56
5.3. Visualización no lineal con t-SNE, aplicada sobre las 10 componentes principales que retienen el 90 % de la varianza.	56

ÍNDICE DE FIGURAS

5.4. Ejemplos de Falsos Positivos (FP) y Falsos Negativos (FN) del modelo ResNet-50.	60
5.5. Muestras generadas por la ncGAN con inicialización MD/MG (Discriminador y Generador desde cero) a la Epoch 100.	63
5.6. Muestras generadas por la cGAN con inicialización MD/MG (Discriminador y Generador desde cero) a la Epoch 50.	64
5.7. Muestras generadas por la cGAN con inicialización MD/PG (Discriminador Desde Cero, Generador Preentrenado) a la Epoch 100.	65
5.8. Muestras generadas por la cGAN con inicialización PD/MG (Discriminador Preentrenado, Generador Desde Cero).	66
5.9. Muestras generadas por la cGAN con inicialización PD/PG (Discriminador y Generador preentrenados) a la Epoch 40.	67
6.1. Favicons sintéticos generados por la GAN utilizando muestreo GMM en el espacio latente (Epoch 050).	73
C.1. Curvas de Pérdida (<i>Loss</i>) y Precisión (<i>Accuracy</i>) del modelo ANN durante el entrenamiento, mostrando la convergencia y el punto de <i>Early Stopping</i> (Fuente: Elaboración propia).	84
C.2. Curvas de Pérdida del Generador (<i>g_loss</i>) y Discriminador (<i>d_loss</i>) de la cGAN (PD/PG) durante el entrenamiento, mostrando el equilibrio adversario (Fuente: Elaboración propia).	84
C.3. Varianza acumulada de las componentes principales. Se retiene el 90 % de la varianza con 9 componentes, confirmando la dimensionalidad intrínseca de los datos (Fuente: Elaboración propia).	86
D.1. Cuadrícula de Favicons (64 × 64) del dataset balanceado, mostrando la diversidad de logotipos y la baja calidad de algunas muestras (Fuente: Elaboración propia).	88
D.2. Muestras generadas por la cGAN con inicialización PD/PG. Exhibe la mejor fidelidad perceptiva y diversidad visual (Fuente: Elaboración propia).	89
D.3. Muestras generadas por la cGAN con inicialización PD/MG. El mejor FID a costa del colapso visual a la media (gris uniforme) (Fuente: Elaboración propia).	89

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

4.1. Combinaciones de Estrategias de Entrenamiento para la GAN	45
5.1. Resultados Cuantitativos de Modelos Basados en Favicons	58
5.2. Resultados de Calidad y Estabilidad de las Estrategias de Inicialización de la GAN	61
B.1. Catálogo de 14 Características Tabulares Seguras utilizadas en la Clasificación	81
B.2. Hiperparámetros Comunes y Modelo Random Forest (RF)	81
B.3. Arquitectura del Perceptrón Multicapa (ANN)	82
B.4. Estructura del Generador (G) Preentrenado (PG) de la cGAN	82
C.1. Métricas de Evaluación Detalladas (Clasificadores Tabulares y Visuales) . .	83
C.2. Matriz de Confusión del Modelo ResNet-50 (Mejor Rendimiento)	83

ÍNDICE DE CÓDIGOS

3.1. Código fuente del script extract_dataset.py	17
3.2. Código fuente del script process_favicons_fixedRGBbalanced.py para normalización y balanceo	21
3.3. Funciones de Feature Engineering y Aplicación al Dataset (Fragmento de modelo_clasificacion.ipynb)	25
3.4. Proceso de observación de feautures tramposas y selección final de features_safe tras la eliminación de dichas features	27
3.5. Carga y Normalización final de Favicons al rango [-1,1]	29
3.6. División estratificada de datos de URLs (Fragmento de modelo_clasificacion.ipynb)	31
3.7. División de datasets en Train/Validation para clasificación (NB_favicon_cgan_PD_PG.ipynb)	32
4.1. Fragmento de código: Entrenamiento, validación y persistencia del modelo RF basado en URL.	35
4.2. Fragmento de código: Definición de la arquitectura Multilayer Perceptron (MLP) para clasificación de URLs.	37
4.3. Fragmento de código: Estandarización, PCA y visualización t-SNE de las características tabulares del set de prueba.	38
4.4. Definición y arquitectura de la CNN desde cero para la clasificación de favicons.	39
4.5. Configuración del <i>Transfer Learning</i> con ResNet-50.	41
4.6. Configuración y bucle de entrenamiento de la cGAN base (Estrategia M D / M G).	45
4.7. Estrategia cGAN M D / P G: Definición condicional y carga del Generador preentrenado.	46
4.8. Estrategia P D / M G: Wrapper condicional de ResNet50 para el Discriminador en una cGAN.	47
4.9. Estrategia P D / P G: Preentrenamiento del Generador condicional (Autoencoder) y configuración final de la cGAN.	49
4.10. Estrategia ncGAN M D / M G (Línea Base): Inicialización desde cero y bucle de entrenamiento.	51
A.1. Script de implementación del cGAN con muestreo GMM	76

ÍNDICE DE CÓDIGOS

CAPÍTULO 1

INTRODUCCIÓN

1.1. Contexto y motivación

El ecosistema digital actual está intrínsecamente ligado a una escalada constante en las ciberamenazas, con el **phishing** erigiéndose como el principal medio de ataque a nivel mundial. Esta técnica de ingeniería social se ha perfeccionado hasta el punto de ser indistinguible de comunicaciones legítimas, convirtiéndose en el método más eficiente y de menor coste para el compromiso de credenciales, el robo de datos y la distribución de malware. La **importancia de esta amenaza** se mide no solo en la **sofisticación de los engaños**, sino en las **pérdidas económicas multimillonarias** que sufren tanto las grandes corporaciones como los usuarios individuales, socavando la confianza fundamental en las transacciones y comunicaciones en línea.

Ante la incesante creación y mutación de URLs fraudulentas, los métodos de defensa tradicionales basados históricamente en la reacción, como las **listas negras**, se han demostrado insuficientes. Estos sistemas son lentos y no pueden seguir el ritmo de la ciberdelincuencia moderna. Esta **crisis de escalabilidad y velocidad** es la que impulsa la **relevancia crítica de la detección automatizada**.

La **motivación fundamental** de este trabajo es desarrollar herramientas predictivas, autónomas y altamente eficientes que puedan superar la limitación de la respuesta humana. La investigación contemporánea en ciberseguridad reside en la necesidad de migrar hacia sistemas que garanticen la integridad y la seguridad de las interacciones digitales. El presente estudio se enmarca precisamente en este contexto, buscando aplicar un análisis avanzado de datos para crear una solución de verificación que no solo reaccione, sino que se anticipe a la persistente y creciente amenaza del *phishing*.

1.2. Planteamiento del problema

El reto fundamental en la lucha contra el *phishing* reside en la capacidad de **discernir automáticamente entre una URL legítima y una fraudulenta** con alta precisión y a gran velocidad. El problema de la detección no se limita a examinar el contenido textual de un enlace, sino que se extiende a la interpretación de **patrones sutiles** que los atacantes utilizan para evadir la seguridad.

Este proyecto aborda dicho desafío mediante la aplicación de **ML** en dos frentes de análisis:

1. **Análisis de Características Tabulares de la URL:** Utilización de atributos de texto y otros aspectos visuales (específicamente presencia o ausencia del favicon) para entrenar modelos discriminadores. Algunos aspectos tabulares textuales:
 - Longitud total de la URL (ej., si es excesivamente larga).
 - Presencia del símbolo "@" (usado a menudo para ofuscar).
 - Conteo de puntos ('.') para determinar el número de subdominios.
 - Uso del protocolo Hypertext Transfer Protocol Secure (HTTPS).
 - Frecuencia de palabras clave asociadas a marcas (ej., "login", "secure").
2. **Análisis de Características Visuales no Obvias:** Exploración de la importancia del **favicon** (el ícono asociado a la web) mediante análisis de sus características, como indicador de fraude, una característica tabular que resultó ser altamente determinante en el análisis preliminar.

Además, el problema se amplía al campo de la generación, buscando no solo detectar sino **simular la creación de enlaces falsos y favicons** a través de **GAN** para poner a prueba la robustez de los modelos de detección en un entorno de ataque activo. El planteamiento exige, por tanto, el desarrollo de una herramienta integral que aborde el *phishing* tanto desde la óptica de la detección como desde la simulación de la amenaza.

1.3. Objetivos del proyecto

El objetivo general del presente TFG es diseñar, implementar y evaluar una **herramienta de verificación automática de URLs** basada en modelos de ML robustos y enfocados en la detección de *phishing*.

Para lograr este fin, se establecen los siguientes objetivos específicos:

1. **Detección Tabular (Baseline):** Implementar y evaluar un modelo discriminador

de ML (*Random Forest*) basado exclusivamente en el análisis de **atributos de texto extraídos de la URL** para establecer una línea base de rendimiento.

2. **Detección Visual (Exploratoria):** Diseñar un modelo discriminador basado en Redes Neuronales Convolucionales (CNN) y técnicas de *Transfer Learning* (**ResNet50**) para clasificar URLs basándose únicamente en la **imagen normalizada del favicon** asociado.

3. **Generación Adversarial (Complementario):**

Como objetivo de investigación complementario, diseñar e implementar una arquitectura **GAN multinivel** para la generación sintética de favicons maliciosos, permitiendo la exploración de la calidad de los generadores y la mejora de los modelos discriminadores.

1.4. Alcance y limitaciones

El alcance del presente TFG abarca la totalidad del **ciclo de vida del ML** aplicado a la ciberseguridad: desde el preprocesamiento de datasets públicos hasta la evaluación y comparación de los modelos resultantes.

1.4.1. Alcance

El trabajo cubre los siguientes aspectos:

- **Clasificación de URLs por Datos Tabulares:** Utilización de un clasificador tradicional (como *Random Forest*) para la detección de *phishing* basada en **características sintácticas y léxicas** de la URL.
- **Detección de Favicons por CNN:** Exploración de modelos discriminadores basados en la imagen del favicon, incluyendo la implementación de un **discriminador preentrenado** basado en una arquitectura **ResNet50**.
- **Generación Adversarial con cGAN y ncGAN:** Diseño de arquitecturas de Redes Generativas Antagónicas **condicionales (cGAN)** y **no condicionales (ncGAN)** para generar favicons maliciosos. Se exploran cruces entre discriminadores y generadores tanto preentrenados como no.
- **Datasets:** Utilización del *PhiUSIIL Phishing URL Dataset*[1] (para características tabulares) y un conjunto de favicons RGB asociados, normalizados a 64×64 y 32×32 píxeles.

1.4.2. Limitaciones

El proyecto se desarrolla bajo las siguientes restricciones y exclusiones:

- **Recursos Computacionales:** El desarrollo y el alcance de las arquitecturas GAN están limitados por el hardware local, restringiendo el tamaño de la red y el número de épocas de entrenamiento.
- **Análisis de Contenido Web:** El análisis se limita a las URLs y las imágenes de favicon; **no se incluye la inspección del contenido interno de la página web** (texto, scripts maliciosos, iframes, etc.) ni la ejecución de código.
- **Algoritmos de Generación:** La arquitectura GAN multinivel solo explora la complejidad en el discriminador (*Transfer Learning* con ResNet); **no se exploran arquitecturas avanzadas de generación** como Progressive GAN, WGAN-GP o el uso de *Transfer Learning* en el generador.
- **Entrenamiento del Generador:** Los generadores implementados son tanto de arquitectura simple (entrenados desde cero) como avanzada (preentrenados), explorando así las diferentes opciones y resultados al alcance. Sin embargo, por lo general su crecimiento y aprendizaje es menor en comparación con los discriminadores de los que se ha dispuesto.
- **Desbalance de Clases en Dataset Visual:** El dataset obtenido del procesamiento del dataset original de URLs presenta un alto desbalance entre el número de muestras legítimas (~64,000) frente a las muestras de *phishing* (~1,595), por ello se optó por un **submuestreo** de la clase mayoritaria (Legítima), creando un dataset balanceado artificial de 1,595 legítimas y 1,595 *phishing*. Si bien esto aborda el desbalance, la **limitada cantidad total de muestras de phishing disponibles** puede haber afectado la capacidad de la red para generalizar patrones complejos.

1.5. Estructura del documento

El presente documento se encuentra estructurado en seis capítulos principales, organizados de la siguiente manera:

- **Capítulo 1. Introducción:** Establece el contexto de la amenaza del *phishing*, el planteamiento del problema de la detección automatizada y define los objetivos y alcance del estudio.
- **Capítulo 2. Estado del arte y fundamentos teóricos:** Revisa la literatura sobre técnicas de *phishing*, la detección mediante análisis de URLs, y sienta las bases teóricas del ML, las CNN y las GAN.
- **Capítulo 3. Metodología y datasets:** Describe el enfoque de trabajo, las fuentes de datos empleadas, y los procesos de limpieza, preprocesamiento y división de los datasets tabulares y visuales.
- **Capítulo 4. Modelos y desarrollo experimental:** Detalla la implementación técnica de los modelos discriminadores (basados en URL y favicons) y la arquitectura de la red GAN para la generación adversarial.

- **Capítulo 5. Resultados y discusión:** Presenta los resultados cuantitativos y cualitativos de la experimentación, analiza el rendimiento de cada modelo y discute las implicaciones de los hallazgos.
- **Capítulo 6. Conclusiones y líneas futuras:** Resume los principales logros del TFG, sus contribuciones, limitaciones, y propone futuras líneas de investigación.

El documento concluye con las **Referencias bibliográficas** y los **Anexos** correspondientes a información detallada y complementaria.

CAPÍTULO 2

ESTADO DEL ARTE Y FUNDAMENTOS TEÓRICOS

2.1. Concepto de phishing y técnicas comunes de detección

El término **phishing** describe una práctica de ingeniería social maliciosa cuyo principal objetivo es la **obtención fraudulenta** de información confidencial, como nombres de usuario, contraseñas, datos bancarios o información personal. Este tipo de ataque se disfraza habitualmente como una comunicación legítima y confiable, imitando la identidad visual y el estilo de entidades conocidas (bancos, plataformas de redes sociales, proveedores de servicios, etc.). El atacante busca engañar a la víctima para que realice una acción específica, ya sea haciendo clic en un enlace que redirige a una página web falsificada o descargando un archivo malicioso. La alta eficacia del *phishing* reside en su capacidad de eludir las barreras técnicas al **explotar la confianza humana**.

A lo largo de los años, la ciberseguridad ha desarrollado diversas técnicas para combatir esta amenaza:

1. **Listas Negras (Blacklists):** Este es el método de detección más tradicional. Consiste en mantener bases de datos constantemente actualizadas de URLs que han sido previamente identificadas como fraudulentas. Cuando un usuario intenta acceder a una URL, esta se compara con la lista. Su principal limitación es su naturaleza **reactiva**: solo puede bloquear sitios ya conocidos y es fácilmente eludida por los atacantes que crean rápidamente dominios nuevos y efímeros.
2. **Análisis Heurístico Basado en Reglas:** Este enfoque utiliza un conjunto predefinido de reglas lingüísticas, sintácticas y de contenido para asignar una puntuación de riesgo a una URL o correo electrónico. Las reglas pueden incluir la presencia de caracteres extraños, errores ortográficos, el uso de palabras clave sospechosas.

sas (ej. *login*, *security*, *update*) o una estructura de dominio anormalmente larga. Si la puntuación excede un umbral, el sitio se marca como sospechoso. Su principal desafío es el **alto índice de falsos positivos** y la facilidad con que los atacantes adaptan sus patrones para evitar estas reglas fijas.

3. **Monitoreo de Domain Name System (DNS) y Certificados:** Se enfoca en la validación de la infraestructura subyacente. Esto incluye verificar si el dominio tiene un historial reciente de creación, si utiliza un servicio de alojamiento dudoso o si el certificado Secure Sockets Layer (SSL) (HTTPS) es inconsistente con el tipo de entidad que dice ser.

La insuficiencia de estas técnicas tradicionales, especialmente frente a los ataques polimórficos y de día cero, motiva la búsqueda de soluciones más inteligentes y predictivas basadas en la automatización.

2.2. Características de las URLs relevantes para la detección

El análisis de **características sintácticas y léxicas** de una URL es la base de la detección automatizada de *phishing*. Estas características permiten transformar la URL, que es una cadena de texto, en un conjunto de datos numéricos (vectores de características) que los modelos de ML pueden procesar. La eficacia de un modelo discriminador depende en gran medida de la calidad y relevancia de los atributos extraídos.

Las categorías de características más comunes y determinantes incluyen:

1. **Características de Longitud:** Se basan en el tamaño del enlace, ya que las URLs maliciosas a menudo intentan ocultar su propósito siendo excesivamente largas o, por el contrario, muy cortas. Se miden la longitud total de la URL, la longitud del dominio, la extensión de la ruta (*path*) y la longitud de los parámetros de consulta.
2. **Características de Contenido Lexical:** Buscan patrones sospechosos dentro de las palabras que componen el dominio. Esto incluye la presencia de términos clave asociados a marcas conocidas (ej. *amazon*, *paypal*) o la inclusión de palabras relacionadas con el fraude (ej. *login*, *security*, *update*).
3. **Características de Caracteres y Símbolos:** Se centran en la presencia de caracteres no habituales o el uso excesivo de símbolos. Por ejemplo, el número de guiones ('-'), el uso del símbolo arroba ('@', que puede usarse para ofuscar la URL), o la presencia de dígitos. Un número elevado de subdominios, indicado por puntos ('.'), también puede ser una señal de alerta.
4. **Características de Dominio e Infraestructura:** Aunque más complejas de obtener, son vitales. Incluyen el uso del protocolo **HTTPS** (aunque ya no es un indicador de seguridad absoluto), la antigüedad del registro del dominio y si el dominio aparece en servicios de *hosting* gratuitos o de dudosa reputación.

2.3. Favicons: función y relevancia visual

El favicon, o *favorite icon*, es un pequeño archivo gráfico asociado a una página web que aparece en la pestaña del navegador, en los marcadores y, en algunos casos, en los resultados de búsqueda. Su función principal es doble: **identificación rápida** y **reforzamiento de la marca**. Para el usuario, el favicon actúa como un marcador visual de la identidad digital de un sitio, siendo el primer elemento gráfico que establece confianza o familiaridad.

En el contexto de la detección de *phishing*, la relevancia del favicon trasciende su propósito estético para convertirse en un **índicador de integridad crucial**. Los atacantes, al crear réplicas rápidas y temporales de sitios web legítimos, a menudo fallan al replicar o integrar correctamente el favicon original, ya sea por prisa técnica, por desconocimiento o por evitar el consumo de recursos al alojar el archivo de imagen. Por lo tanto, la **ausencia o inconsistencia** del favicon en una URL de *phishing* se convierte en una **señal visual binaria** de alta fiabilidad.

Este hallazgo empírico motiva la metodología de este TFG. Mientras que los modelos tradicionales se centran en el texto (sección 2.2), el análisis del favicon permite la aplicación de técnicas de **procesamiento de imágenes** y **CNN** para detectar el fraude, explotando así la dimensión visual del ataque y el fallo operativo del ciberdelincuente.



Figura 2.1: Favicon asociado a <https://www.youtube.com>

2.4. Aprendizaje automático en detección de phishing

La evolución del *phishing* ha impulsado el desarrollo de soluciones basadas en el **Aprendizaje Automático (ML)**, ya que estos sistemas pueden aprender patrones complejos y adaptarse dinámicamente a nuevas amenazas, algo que los métodos basados en reglas no logran. En el ámbito de la detección de *phishing*, los modelos de ML se dividen típicamente en dos categorías según el tipo de datos que procesan:

2.4.1. Modelos Clásicos sobre Datos Tabulares

Estos modelos operan sobre los vectores numéricos de características extraídas de las URLs (como las longitudes, el conteo de símbolos, o el uso de palabras clave). Su objetivo es clasificar la URL como *phishing* (clase 0) o legítima (clase 1). Los algoritmos más comunes en este campo incluyen:

- **RF:** Es un potente algoritmo de aprendizaje de conjunto (*ensemble learning*) que opera construyendo múltiples **árboles de decisión** durante el entrenamiento. La predicción final (la clase de la URL) se determina por la clase que recibe más votos de los árboles individuales. Es robusto frente al sobreajuste (*overfitting*) y puede manejar eficazmente un gran número de características de entrada, lo que lo hace ideal para tareas de clasificación binaria como la detección de *phishing*. Su capacidad para determinar la **importancia de cada característica** es un valor añadido en la ingeniería de características.
- **Logistic Regression (LR):** A pesar de su nombre, es un modelo de clasificación lineal que estima la **probabilidad** de que una URL pertenezca a la clase *phishing* (clase 1). Utiliza la función logística (*sigmoide*) para mapear el resultado de la combinación lineal de las características a una probabilidad entre 0 y 1. Es un modelo rápido de entrenar y ofrece una alta **interpretabilidad** de los pesos asignados a cada característica.
- **K-Nearest Neighbors (KNN):** Este es un clasificador basado en la instancia (*instance-based*) que no aprende un modelo explícito. Clasifica una nueva URL basándose en la mayoría de votos de sus *K vecinos más cercanos* en el espacio de características. Su simplicidad lo hace útil en el análisis exploratorio, aunque su coste computacional en la predicción puede ser alto con datasets muy grandes.
- **Support Vector Machines (SVM):** Este es un modelo discriminativo basado en la teoría del aprendizaje estadístico. Su objetivo principal es encontrar el **hiperplano óptimo** que maximiza el margen entre las dos clases de datos (*phishing* y legítimo) en el espacio de características. Utiliza funciones *kernel* para transformar el espacio de características y poder separar clases que no son linealmente separables en su forma original.
- **Gradient Boosting (GB):** Representa una familia avanzada de modelos *ensemble* que construyen árboles de forma **secuencial**. Cada árbol intenta corregir los errores residuales cometidos por el conjunto de árboles anterior. Modelos como *XGBoost* o *LightGBM* son variantes extremadamente populares por su capacidad para alcanzar la más alta precisión en la mayoría de tareas de datos tabulares, aunque son más complejos de ajustar.
- **ANN (Redes Neuronales Artificiales):** Modelos de varias capas que pueden capturar **relaciones no lineales y complejas** entre las características de entrada. Aunque menos profundas que las CNN, las ANN son fundamentales cuando las características tabulares requieren una abstracción compleja para la clasificación.

2.4.2. Modelos Profundos sobre Datos Visuales

Cuando la detección se basa en imágenes (como los favicons en este proyecto), se recurre a arquitecturas de **Aprendizaje Profundo** (*Deep Learning*). La necesidad de procesar matrices de píxeles para identificar patrones visuales requiere la capacidad de las redes convolucionales, lo que nos lleva al siguiente punto.

2.5. Redes convolucionales (CNN) y Transfer Learning con ResNet

El análisis de favicons, al tratarse de datos visuales, requiere arquitecturas especializadas como las **Redes Neuronales Convolucionales (CNN)**. Las CNN destacan en la extracción automática y jerárquica de características espaciales, superando la limitación de los modelos clásicos en la interpretación de imágenes.

2.5.1. Fundamentos de las CNN

Las CNN basan su funcionamiento en el uso de **capas convolucionales** que aplican filtros (*kernels*) a la imagen de entrada. Estas capas aprenden progresivamente: las primeras capas detectan características de bajo nivel (bordes, líneas, curvas), y las capas más profundas combinan estas características para reconocer patrones de alto nivel (logotipos, texto, formas complejas).

La arquitectura típica de una CNN se compone de una pila secuencial de estas capas. Generalmente, se apilan varias **capas convolucionales** (cada una seguida por una capa de activación **ReLU**). A estas les sigue una **capa de pooling (submuestreo)**, y este patrón se repite. A medida que la imagen (o, más precisamente, el mapa de características) avanza a través de la red, su **dimensión espacial se reduce** (debido al *pooling*), pero su **profundidad se incrementa** (es decir, aumenta el número de mapas de características), gracias a las capas convolucionales.

En la parte superior de esta pila de extracción de características se añade una **red neuronal de tipo feedforward regular**, compuesta por unas pocas **capas completamente conectadas**, a menudo también seguidas por activaciones ReLU. La capa final es la encargada de emitir la predicción (por ejemplo, una capa softmax que genera las probabilidades estimadas de pertenencia a cada clase, en nuestro caso "*Legítima*º "Phishing").

En la tarea de clasificar favicons como legítimos o falsos, las CNN son cruciales porque:

- **Reconocimiento de Marca:** Pueden aprender a reconocer la identidad visual de una marca legítima o, por el contrario, detectar la ausencia o la baja calidad de los gráficos que indican un fraude.
- **Invarianza de Traslación:** Son robustas a pequeños cambios en la posición del logotipo dentro del marco del favicon.

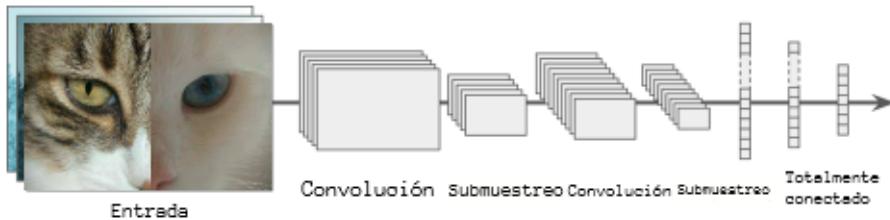


Figura 2.2: Arquitectura típica de una CNN

2.5.2. Transfer Learning con ResNet

El **Transfer Learning** es una técnica esencial para superar la escasez de grandes datasets de entrenamiento. En lugar de construir y entrenar una CNN profunda desde cero, se utiliza un modelo (*backbone*) ya preentrenado en un problema masivo de visión por ordenador (como el *ImageNet*).

- **ResNet (Residual Network):** En este proyecto, la implementación del discriminador avanzado se basa en una arquitectura ResNet50. ResNet se caracteriza por el uso de **conexiones residuales** (*skip connections*), que permiten saltar capas y facilitan la propagación del gradiente a través de redes muy profundas.
- **Proceso de Reutilización:** El modelo ResNet50 ya ha aprendido una vasta jerarquía de características visuales generales. Al aplicarlo a la clasificación de favicons, se reutilizan las capas convolucionales iniciales como extractores de características (fijas o con reentrenamiento fino o *fine-tuning*), y solo se entrena la capa de salida final para clasificar específicamente “Legítimo” o “Phishing”.
- **Justificación para GAN:** El uso de ResNet preentrenado como discriminador en la GAN eleva el “nivel de inteligencia” del detector, forzando al generador a producir favicons sintéticos de una calidad visual significativamente superior para tener éxito en el engaño.

2.6. Generative Adversarial Networks (GAN)

Las **Generative Adversarial Networks (GAN)** representan una clase de modelos de Aprendizaje Profundo diseñados para generar datos nuevos y sintéticos que son indistinguibles de los datos reales. Este marco, introducido por Ian Goodfellow, se basa en una estructura de “juego de suma cero” entre dos redes neuronales que compiten entre sí: el Generador y el Discriminador.

2.6.1. El Marco Antagónico

Como se ha mencionado, una GAN está compuesta por dos modelos principales:

- **Generador (G):** Es la red encargada de aprender la distribución de los datos reales para crear nuevas muestras sintéticas (imágenes en este caso). Toma como entrada un vector de ruido aleatorio (*latent vector*) y, a través de capas de convolucionales o de *upsampling*, produce una imagen que busca ser indistinguible de los datos reales.
- **Discriminador (D):** Es un clasificador (típicamente una CNN) que se entrena para distinguir entre las muestras **reales** (provenientes del *dataset*) y las muestras **falsas** (generadas por *G*). La salida del Discriminador es una probabilidad de que la imagen sea real.

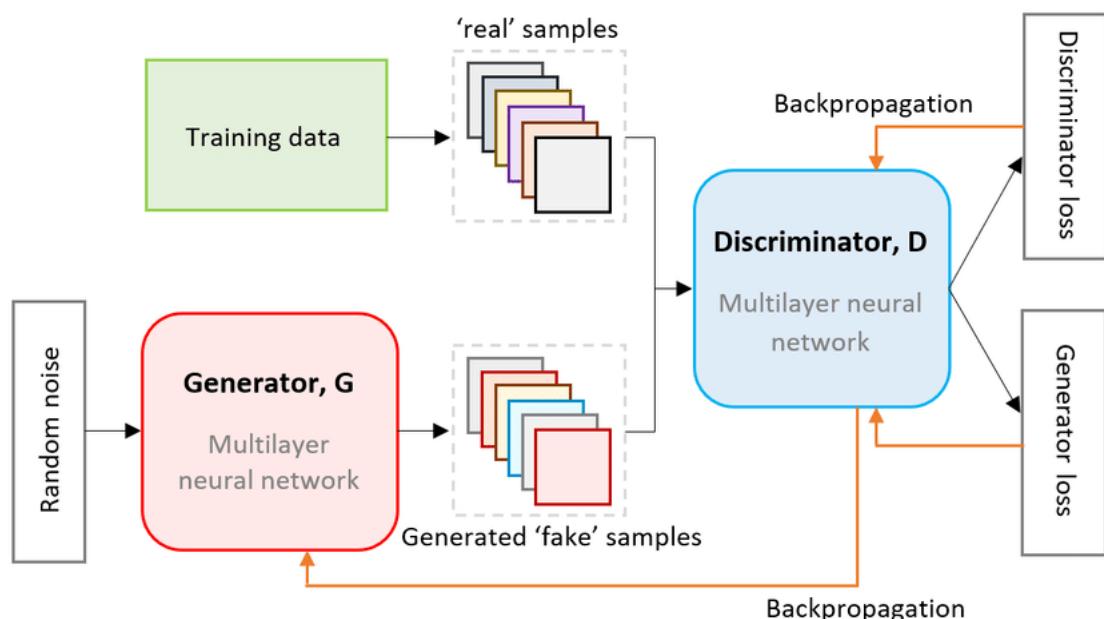


Figura 2.3: Arquitectura típica de una GAN

Ambas redes se entrenan simultáneamente: el Generador (*G*) busca minimizar la capacidad del Discriminador para identificar las falsificaciones, mientras que el Discriminador (*D*) intenta maximizar su capacidad para hacer esta distinción correctamente.

Un aspecto crucial en la implementación de las GAN es el nivel de entrenamiento inicial de cada componente, que puede verse como su "nivel de inteligencia". Este TFG ha explorado sistemáticamente **todas las combinaciones posibles de entrenamiento** para evaluar su impacto en la calidad de los favicons generados y la estabilidad del proceso:

1. **Ambos No Preentrenados (MD/MG):** Tanto el Generador como el Discriminador se inicializan y entranen **desde cero** con el *dataset* de favicons.
2. **Discriminador No Preentrenado y Generador Preentrenado (MD/PG):** Se utiliza una base preentrenada para el Generador mientras el Discriminador se entrena desde cero.

3. **Discriminador Preentrenado y Generador No Preentrenado (PD/MG):** El Generador se entrena desde cero, pero el Discriminador utiliza el **Transfer Learning** (como con ResNet, tal como se describe en la Sección 2.5.2) para aumentar su capacidad discriminativa de forma temprana.
4. **Ambos Preentrenados (PD/PG):** Ambos componentes se inician utilizando pesos de modelos preentrenados, combinando el mayor "nivel de inteligencia" inicial posible en la red antagónica.

La inclusión de estas combinaciones permite determinar la estrategia óptima para la generación de favicons de *phishing* y poder estudiar además de comparar los diferentes resultados.

2.6.2. Tipos de GAN en el Proyecto

El proyecto explora dos variantes cruciales de GAN:

- **ncGAN (GAN No Condicional):** El Generador simplemente crea imágenes que se parecen a *cualquier* favicon del dataset completo, sin control sobre la clase (legítimo o *phishing*).
- **cGAN (GAN Condicional):** La generación está guiada por una **condición** (la etiqueta de clase, e.g., "generar un favicon de *phishing*"). Esta etiqueta se alimenta tanto al Generador como al Discriminador, permitiendo un control preciso sobre el tipo de imagen sintética que se crea.

2.7. Métricas y Evaluación de Modelos

La evaluación de los modelos en ciberseguridad, especialmente en tareas de clasificación binaria como la detección de *phishing*, requiere ir más allá de la simple precisión (Accuracy). Dado el impacto asimétrico de los errores (un falso negativo es más grave que un falso positivo), es fundamental emplear métricas basadas en la matriz de confusión.

2.7.1. Métricas de Clasificación Binaria

Para un modelo clasificador que distingue entre la clase Positiva (**Legítima**) y la clase Negativa (**Phishing**), se utilizan las siguientes métricas:

- **Accuracy (Precisión General):** Proporción de predicciones correctas sobre el total de casos. Útil como punto de partida, pero engañosa en conjuntos de datos

desbalanceados.

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

Donde:

- *TP*: True Positive (Verdadero Positivo) (TP)
- *TN*: True Negative (Verdadero Negativo) (TN)
- *FP*: False Positive (Falso Positivo) (FP)
- *FN*: False Negative (Falso Negativo) (FN)

- **Precision (Valor Predictivo Positivo):** Mide la calidad de las detecciones del modelo. De todas las instancias clasificadas como *phishing*, ¿cuántas fueron realmente *phishing*? Una alta Precision es crucial, ya que minimiza los **Falsos Positivos (FP)**, evitando que sitios legítimos sean bloqueados o señalados erróneamente.

$$\text{Precision} = \frac{TP}{TP + FP}$$

- **Recall (Sensibilidad o Tasa de Verdaderos Positivos):** Mide la capacidad del modelo para detectar la amenaza. De todos los sitios de *phishing* reales, ¿cuántos detectó el modelo? Es la métrica más importante en ciberseguridad, ya que su maximización minimiza los **Falsos Negativos (FN)** (es decir, sitios fraudulentos que son erróneamente permitidos).

$$\text{Recall} = \frac{TP}{TP + FN}$$

- **F1-Score:** Media armónica entre Precision y Recall. Ofrece una métrica equilibrada para la clasificación, especialmente relevante en presencia de desbalance de clases.

$$F1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

- **Area Under the Receiver Operating Characteristic Curve (AUC-ROC):** Mide la capacidad del modelo para distinguir entre clases en varios umbrales. Un valor cercano a 1 indica un excelente rendimiento de separación.

Donde TP, TN, FP y FN provienen de la Matriz de Confusión.

		Predicted condition	
		Positive (PP)	Negative (PN)
Actual condition	Total population = P + N	Positive (P)	True positive (TP) False negative (FN)
	Negative (N)	False positive (FP)	True negative (TN)

Figura 2.4: Matriz de confusión. Fuente: Wikipedia (https://en.wikipedia.org/wiki/Confusion_matrix)

2.7.2. Métricas para la Evaluación de la Calidad de las GAN

La evaluación del rendimiento de la Generación es más subjetiva y compleja, y se requiere ir más allá de la simple inspección visual.

- **Función de Pérdida (Loss) del Discriminador (D):** Se monitoriza para asegurar que no caiga a cero (lo que indicaría que ha aprendido demasiado rápido).
- **Función de Pérdida (Loss) del Generador (G):** Se espera que disminuya, indicando que el generador está mejorando en la tarea de engañar al discriminador.
- **Inception Score (IS) o FID:** Aunque son las métricas estándar de la industria para evaluar la **calidad y diversidad** de las imágenes generadas por una GAN, no fueron implementadas debido a su alta complejidad computacional. Su exclusión es una limitación reconocida del proyecto.

2.8. Síntesis del Estado del Arte y Carencias Detectadas

La **detección de phishing tradicional** basada en listas negras y análisis heurístico ha demostrado ser **reactiva y vulnerable a la ofuscación**. La irrupción del **Aprendizaje Automático (ML)** ha ofrecido una solución robusta en el análisis de datos tabulares (características de URL), con modelos clásicos como RF y SVM demostrando alta eficacia.

Sin embargo, la principal **carenza detectada** en el estado del arte es el uso limitado de la **dimensión visual** en la detección de *phishing*. Las soluciones existentes se enfocan predominantemente en el texto de la URL, ignorando el *path* más corto y eficiente hacia el engaño: la manipulación de la identidad visual (el favicon).

Este proyecto aborda directamente esta carencia a través de dos frentes:

1. El uso de **CNNs y Transfer Learning (ResNet50)** para validar el favicon como un indicador binario de fraude, moviendo el foco del análisis puramente textual a la imagen.
2. La exploración de **Redes Generativas Antagónicas (GAN)** para generar favicons de *phishing* sintéticos de alta calidad. Esto no solo demuestra la capacidad de los atacantes para automatizar la creación de sueños visuales, sino que también ofrece un método para aumentar y diversificar los *datasets* de entrenamiento de la clase minoritaria (*phishing*), mitigando el problema del desbalance de clases y fortaleciendo las defensas.

CAPÍTULO 3

METODOLOGÍA Y DATASETS

3.1. Enfoque Metodológico General

La presente investigación se estructura en una serie de fases secuenciales y bien definidas, diseñadas para abordar de manera sistemática la clasificación de sitios web. El enfoque adoptado combina técnicas de **procesamiento de datos tabulares** y **visión por computador** para explotar la información contenida tanto en las URLs como en los favicons de los sitios web.

1. **Búsqueda y Adquisición de Datasets:** Identificación, selección y recopilación de fuentes de datos públicas y privadas que contengan pares de URLs y sus correspondientes favicons, junto con etiquetas de clasificación.
2. **Preprocesamiento de Fuentes:** Estandarización, limpieza y validación de los datos brutos. Esto incluye la normalización de URLs y la verificación de la accesibilidad y validez de los favicons.
3. **Procesado de Datos Tabulares:** Extracción de características de las URLs (p. ej., longitud, presencia de *keywords*, estructura de *subdominios*) y codificación de atributos categóricos.
4. **Procesado de Imágenes (Favicons):** Descarga, redimensionamiento, normalización de color y alineación de los favicons para su uso como entrada en modelos de aprendizaje profundo.
5. **División de Conjuntos:** Partición de los datos en conjuntos de entrenamiento, validación y prueba, asegurando una distribución equitativa de las clases.
6. **Modelado y Entrenamiento:** Diseño, implementación y entrenamiento de los modelos de clasificación.

7. **Evaluación y Resultados:** Medición del rendimiento de los modelos mediante métricas adecuadas y análisis de los resultados.

3.2. Datasets Empleados

El éxito de la clasificación depende en gran medida de la calidad y diversidad de los datos de entrenamiento. Se empleó un dataset primario de URLs y, a partir de este, se construyó un dataset secundario de imágenes (favicons).

- **Dataset Primario (URLs y Etiquetas):** Se utilizó el *PhiUSIIL Phishing URL Dataset* (University of California, Irvine (UCI)), un conjunto de datos a gran escala que proporciona URLs junto con una etiqueta binaria de clasificación: 0 (**phishing**) y 1 (**legítimo**).
- **Dataset Secundario (Favicons):** Este conjunto de datos fue creado *ad-hoc* para esta investigación. Se extrajeron todas las URLs del dataset primario que poseían un indicador de *HasFavicon* positivo, y se procedió a la descarga y procesamiento de sus respectivos favicons. La descarga se realizó mediante el script de Python `extract_dataset.py`, cuyo código se presenta a continuación:

```

1 #!/usr/bin/env python3
2 """
3 extract_dataset.py
4
5 Descarga favicons de URLs con HasFavicon == 1 del CSV,
6 """
7
8 import os
9 import csv
10 import time
11 import re
12 from urllib.parse import urlparse, urljoin
13 from concurrent.futures import ThreadPoolExecutor, as_completed
14
15 import requests
16 from bs4 import BeautifulSoup
17 import pandas as pd
18 from tqdm import tqdm
19
20 # ----- CONFIGURACIÓN -----
21 CSV_PATH = r"data/raw/PhiUSIIL_Phishing_URL_Dataset.csv"
22 OUTPUT_DIR = r"data/temp/favicons_descargados"
23 RESULT_CSV = OUTPUT_DIR / "favicons_results.csv"
24 MAX_WORKERS = 24          # Aumenta si tu CPU lo permite
25 REQUEST_TIMEOUT = 8       # Timeout más corto para evitar bloqueos
26 SAVE_EVERY = 1000         # Guardar progreso cada N URLs
27 USER_AGENT = "Mozilla/5.0 (compatible; favicon-downloader/1.1; +https://example.local/)"
28 SLEEP_BETWEEN = 0.01      # Pequeña pausa entre tareas
29
30 # ----- FUNCIONES AUXILIARES -----
31 def sanitize_filename(s: str) -> str:
32     s = re.sub(r'[^A-Za-z0-9_.-]', '_', s)

```

```

33     return s[:200]
34
35 def get_domain_base(url: str) -> str:
36     try:
37         parsed = urlparse(url)
38         scheme = parsed.scheme or "http"
39         netloc = parsed.netloc or parsed.path
40         return f"{scheme}://{netloc}"
41     except Exception:
42         return None
43
44 def fetch_html(url: str):
45     headers = {"User-Agent": USER_AGENT}
46     try:
47         r = requests.get(url, headers=headers, timeout=REQUEST_TIMEOUT, allow_redirects=True)
48         r.raise_for_status()
49         return r.text, r.url
50     except Exception:
51         return None, None
52
53 def find_icon_links_from_html(html: str, base_url: str):
54     soup = BeautifulSoup(html, "html.parser")
55     tags = soup.find_all("link", rel=lambda v: v and any(x in v.lower() for x in ["icon", "shortcut", "apple-touch-icon"]))
56     results = []
57     for t in tags:
58         href = t.get("href")
59         if href:
60             results.append(urljoin(base_url, href))
61     # quitar duplicados manteniendo orden
62     seen, dedup = set(), []
63     for u in results:
64         if u not in seen:
65             dedup.append(u)
66             seen.add(u)
67     return dedup
68
69 def download_binary(url: str):
70     headers = {"User-Agent": USER_AGENT}
71     r = requests.get(url, headers=headers, timeout=REQUEST_TIMEOUT, stream=True,
72                      allow_redirects=True)
73     r.raise_for_status()
74     return r.content, r.headers.get("Content-Type", "")
75
76 # ----- LÓGICA DE PROCESAMIENTO -----
77 def process_url(index: int, url: str, label: str):
78     out = {
79         "URL": url,
80         "label": label,
81         "favicon_path": "",
82         "favicon_url": "",
83         "status": "not_found",
84         "error": ""
85     }
86     try:
87         base = get_domain_base(url)
88         if not base:
89             out["status"] = "bad_url"
90             out["error"] = "No se pudo parsear la URL"
91     return out

```

```

91
92     html, final_page = fetch_html(url)
93     if html is None:
94         html, final_page = fetch_html(base)
95
96     candidates = []
97     if html:
98         candidates.extend(find_icon_links_from_html(html, final_page or base))
99     candidates.append(urljoin(base, "/favicon.ico"))
100
101    for cand in candidates:
102        try:
103            data, ctype = download_binary(cand)
104            if not data:
105                continue
106
107            # Extensión según tipo
108            if "png" in ctype:
109                ext = ".png"
110            elif "svg" in ctype:
111                ext = ".svg"
112            elif "jpeg" in ctype or "jpg" in ctype:
113                ext = ".jpg"
114            elif "ico" in ctype or cand.lower().endswith(".ico"):
115                ext = ".ico"
116            else:
117                ext = ".bin"
118
119            domain = sanitize_filename(urlparse(base).netloc)
120            fname = f"{index:06d}_{domain}{ext}"
121            os.makedirs(OUTPUT_DIR, exist_ok=True)
122            path = os.path.join(OUTPUT_DIR, fname)
123            with open(path, "wb") as f:
124                f.write(data)
125
126            out.update({
127                "favicon_path": path,
128                "favicon_url": cand,
129                "status": "downloaded"
130            })
131        return out
132
133    except Exception as e_cand:
134        last_err = str(e_cand)
135        continue
136
137    out["status"] = "not_downloaded"
138    out["error"] = last_err if 'last_err' in locals() else "Sin favicon válido"
139    return out
140
141    except Exception as e:
142        out["status"] = "error"
143        out["error"] = str(e)
144        return out
145
146 # ----- FUNCIÓN PRINCIPAL -----
147 def main():
148     print(" Leyendo CSV:", CSV_PATH)
149     df = pd.read_csv(CSV_PATH, usecols=lambda c: c.lower() in ["url", "hasfavicon", "label"])
150     df.columns = [c.lower() for c in df.columns]

```

```

151     df = df[df["hasfavicon"] == 1].reset_index(drop=True)
152     print(f"Encontradas {len(df)} URLs con HasFavicon == 1")
153
154     # Cargar progreso previo si existe
155     results = []
156     processed_urls = set()
157     if os.path.exists(RESULT_CSV):
158         print(f" Reanudando desde: {RESULT_CSV}")
159         prev = pd.read_csv(RESULT_CSV)
160         processed_urls = set(prev["URL"])
161         results = prev.to_dict("records")
162         df = df[~df["url"].isin(processed_urls)]
163         print(f"Se omitirán {len(processed_urls)} URLs ya procesadas. Restan {len(df)}.")
164
165     # Descarga concurrente
166     with ThreadPoolExecutor(max_workers=MAX_WORKERS) as ex:
167         futures = {ex.submit(process_url, i, row["url"], row["label"]): i for i, row in
168         df.iterrows()}
169         for i, fut in enumerate(tqdm(as_completed(futures), total=len(futures), desc="Descargando favicons")):
170             try:
171                 res = fut.result()
172             except Exception as e:
173                 res = {"URL": "", "label": "", "favicon_path": "", "favicon_url": "", "status": "error", "error": str(e)}
174             results.append(res)
175
176     # Guardar progreso cada N resultados
177     if len(results) % SAVE_EVERY == 0:
178         pd.DataFrame(results).to_csv(RESULT_CSV, index=False)
179         print(f" Progreso guardado ({len(results)}) resultados...")
180
181         time.sleep(SLEEP_BETWEEN)
182
183     # Guardado final
184     pd.DataFrame(results).to_csv(RESULT_CSV, index=False)
185     print(" Descarga completa. Resultados guardados en:", RESULT_CSV)
186
187 # ----- EJECUCIÓN -----
188 if __name__ == "__main__":
189     main()

```

Código 3.1: Código fuente del script extract_dataset.py

El script `extract_dataset.py` se encarga de la descarga concurrente de los favicons, identificando el dominio base de la URL, buscando enlaces a iconos dentro del HTML de la página, e intentando descargar tanto los iconos referenciados como el archivo estándar `/favicon.ico`.

3.2.1. Construcción y Balanceo del Dataset de Favicons

El dataset original de `favicons` presentaba un grave **desequilibrio de clases**, con aproximadamente 64,000 favicons de URLs legítimas frente a 1,500 de *phishing* (estimación basada en la alta proporción de URLs legítimas que tienen favicon explicando

así más en detalle el porque esta característica resultaba ser tan determinante para la clasificación original) Para abordar este problema crítico en el entrenamiento de modelos de *Deep Learning*, se aplicó una estrategia de muestreo durante el procesamiento de imágenes:

1. **Procesamiento Robustos:** Se emplearon varios scripts de Python (`process_favicons_XXX.py`) para manejar diversos formatos de imagen (ICO, SVG, PNG, JPG, GIF) y normalizar el tamaño de los favicons a 64×64 píxeles. En particular, el script `process_favicons_fixedRGBbalanced.py` fue el utilizado para la generación del dataset final, ya que mantiene los tres canales de color (**RGB**) y realiza el balanceo de clases.
2. **Balanceo de Clases:** Como se ha mencionado, `process_favicons_fixedRGBbalanced.py` implementa un muestreo estratificado para equilibrar las clases. La cantidad de muestras se ajustó al número de imágenes descargadas correctamente de la clase minoritaria, lo que resultó en un total de **1595 imágenes** para la clase legítima y **1595 imágenes** para la clase *phishing*.

```

1 #!/usr/bin/env python3
2 """
3 process_favicons_fixedRGBbalanced.py
4
5 Versión robusta y equilibrada:
6 - Mantiene las imágenes a color (RGB)
7 - Maneja .png, .jpg, .ico, .svg (opcional) y .gif (primer frame)
8 - Balancea ambas clases a TARGET_PER_CLASS
9 """
10
11 import os
12 import io
13 import cv2
14 import pandas as pd
15 import numpy as np
16 from PIL import Image, UnidentifiedImageError
17 from pathlib import Path
18 from tqdm import tqdm
19 import warnings
20
21 # --- CONFIGURACIÓN ---
22 CSV_PATH = Path(r"data/temp/favicons_results.csv")
23 OUTPUT_BASE = Path(r"data/processed/favicons_color_balanced_fixed")
24 IMG_SIZE = (64, 64)
25 LOG_FILE = OUTPUT_BASE / "process_errors.log"
26
27 # Número objetivo por clase
28 TARGET_PER_CLASS = 1595
29 RANDOM_STATE = 42 # reproducibilidad
30
31 # --- SILENCIAR WARNINGS ---
32 warnings.filterwarnings("ignore", category=UserWarning)
33 os.environ["OPENCV_LOG_LEVEL"] = "SILENT"
34
35 # --- INTENTAR CARGAR CairoSVG (para SVG opcional) ---
36 try:
37     import cairosvg
38     SVG_ENABLED = True

```

```

39     except Exception:
40         SVG_ENABLED = False
41
42 # --- CREAR DIRECTORIOS ---
43 LEGIT_DIR = OUTPUT_BASE / "legit"
44 PHISH_DIR = OUTPUT_BASE / "phishing"
45 LEGIT_DIR.mkdir(parents=True, exist_ok=True)
46 PHISH_DIR.mkdir(parents=True, exist_ok=True)
47 OUTPUT_BASE.mkdir(parents=True, exist_ok=True)
48
49 # --- LEER CSV ---
50 print(f" Leyendo CSV: {CSV_PATH}")
51 df = pd.read_csv(CSV_PATH)
52 df["label"] = df["label"].astype(int)
53 df = df[df["status"] == "downloaded"]
54 print(f"Encontradas {len(df)} imágenes descargadas correctamente.")
55
56 # --- BALANCEAR CLASES ---
57 df_legit = df[df["label"] == 1].copy()
58 df_phish = df[df["label"] == 0].copy()
59
60 n_legit = len(df_legit)
61 n_phish = len(df_phish)
62 print(f" - legítimas encontradas: {n_legit}")
63 print(f" - phishing encontradas: {n_phish}")
64
65 # Igualar ambas clases a TARGET_PER_CLASS
66 target = min(TARGET_PER_CLASS, n_legit, n_phish)
67 df_legit_sampled = df_legit.sample(n=target, random_state=RANDOM_STATE)
68 df_phish_sampled = df_phish.sample(n=target, random_state=RANDOM_STATE)
69
70 df_final = pd.concat([df_legit_sampled, df_phish_sampled], ignore_index=True)
71 df_final = df_final.sample(frac=1.0, random_state=RANDOM_STATE).reset_index(drop=True)
72
73 print(f"Total a procesar tras balanceo: {len(df_final)} (legit: {len(df_legit_sampled)}, phish: {len(df_phish_sampled)})")
74
75 # --- FUNCIONES AUXILIARES ---
76 def log_error(path, msg):
77     with open(LOG_FILE, "a", encoding="utf-8") as log:
78         log.write(f"{path}: {msg}\n")
79
80 def convert_svg(svg_path: Path) -> Path:
81     if not SVG_ENABLED:
82         return None
83     tmp_path = svg_path.with_suffix(".converted.png")
84     try:
85         cairosvg.svg2png(url=str(svg_path), write_to=str(tmp_path))
86         return tmp_path
87     except Exception as e:
88         log_error(svg_path, f"Error convirtiendo SVG: {e}")
89         return None
90
91 def process_gif(path: Path):
92     """Carga el primer frame de un GIF y devuelve imagen BGR."""
93     try:
94         pil_img = Image.open(path)
95         pil_img.seek(0)
96         frame = pil_img.convert("RGB")
97         img = cv2.cvtColor(np.array(frame), cv2.COLOR_RGB2BGR)
98         return img

```

```
99     except Exception as e:
100         log_error(path, f"Error procesando GIF: {e}")
101         return None
102
103 # --- PROCESAMIENTO PRINCIPAL ---
104 processed, failed = 0, 0
105
106 for i, row in tqdm(df_final.iterrows(), total=len(df_final), desc="Procesando imágenes"):
107     :
108     path = Path(row["favicon_path"])
109     label = int(row["label"])
110
111     if not path.exists() or path.stat().st_size == 0:
112         log_error(path, "Archivo inexistente o vacío")
113         failed += 1
114         continue
115
116     # --- Manejo de formatos ---
117     ext = path.suffix.lower()
118     if ext == ".svg":
119         new_path = convert_svg(path)
120         if not new_path or not new_path.exists():
121             failed += 1
122             continue
123         path = new_path
124         img = cv2.imread(str(path), cv2.IMREAD_UNCHANGED)
125     elif ext == ".gif":
126         img = process_gif(path)
127         if img is None:
128             failed += 1
129             continue
130     elif ext == ".ico":
131         try:
132             pil_img = Image.open(path).convert("RGBA")
133             img_bytes = io.BytesIO()
134             pil_img.save(img_bytes, format="PNG")
135             img_bytes.seek(0)
136             file_bytes = np.asarray(bytearray(img_bytes.read()), dtype=np.uint8)
137             img = cv2.imdecode(file_bytes, cv2.IMREAD_UNCHANGED)
138         except Exception as e:
139             log_error(path, f"Error leyendo ICO: {e}")
140             failed += 1
141             continue
142     else:
143         img = cv2.imread(str(path), cv2.IMREAD_UNCHANGED)
144
145     if img is None or img.size == 0:
146         log_error(path, "OpenCV no pudo leer la imagen")
147         failed += 1
148         continue
149
150     # --- Normalizar canales ---
151     try:
152         if len(img.shape) == 3:
153             ch = img.shape[-1]
154             if ch == 4: # Transparencia → fondo blanco
155                 alpha = img[:, :, 3] / 255.0
156                 rgb = img[:, :, :3]
157                 img = (255 * (1 - alpha[..., None]) + rgb * alpha[..., None]).astype("uint8")
158             elif len(img.shape) == 2:
```

```

158         img = cv2.cvtColor(img, cv2.COLOR_GRAY2BGR)
159     else:
160         log_error(path, f"Formato no reconocido: shape {img.shape}")
161         failed += 1
162         continue
163     except Exception as e:
164         log_error(path, f"Error normalizando canales: {e}")
165         failed += 1
166         continue
167
168 # --- Redimensionar ---
169 try:
170     img = cv2.resize(img, IMG_SIZE, interpolation=cv2.INTER_AREA)
171 except Exception as e:
172     log_error(path, f"Error redimensionando: {e}")
173     failed += 1
174     continue
175
176 # --- Guardar ---
177 out_dir = LEGIT_DIR if label == 1 else PHISH_DIR
178 out_path = out_dir / f"{i:06d}_{path.stem}.png"
179
180 try:
181     cv2.imwrite(str(out_path), img)
182     processed += 1
183 except Exception as e:
184     log_error(path, f"Error guardando: {e}")
185     failed += 1
186     continue
187
188 print("\n Conversión completada.")
189 print(f" - {processed} imágenes procesadas correctamente.")
190 print(f" - {failed} fallidas (ver {LOG_FILE})")
191 print(f" - SVG activado: {SVG_ENABLED}")

```

Código 3.2: Código fuente del script `process_favicons_fixedRGBbalanced.py` para normalización y balanceo

La aplicación de este balanceo asegura que el modelo no esté sesgado hacia la clase mayoritaria y pueda aprender características distintivas de la clase *phishing*, esencial para obtener métricas de rendimiento robustas.

3.3. Procesado de Datos Tabulares

El procesamiento de las URLs y otros atributos tabulares (Dominio, Top-Level Domain (TLD) y Título) es crucial para extraer características informativas y convertirlas en un formato interpretable por modelos de *Machine Learning*. Este proceso se llevó a cabo utilizando el notebook `modelo_clasificacion.ipynb`.

- **Extracción de Características (Feature Engineering):** Se diseñaron y calcularon manualmente un total de **15 características** a partir de las columnas existentes, centrándose en métricas estructurales y lexicales asociadas a URLs de *phishing*.

- **Características Clave:** Incluyen la longitud de la URL y del dominio, el conteo de puntos y dígitos, la presencia de caracteres especiales ('@'), la verificación del protocolo https y la identificación de *keywords* sospechosas tanto en la URL (e.g., login, bank) como en el Título.
- **Codificación de Atributos:** Para la codificación del TLD, se optó por una característica numérica que mide únicamente su **longitud**. Los demás atributos de entrada ya eran de naturaleza numérica o binaria.

A continuación, se muestra el fragmento de código que contiene las funciones de extracción de características y la posterior aplicación al conjunto de datos, resultando en la matriz de características (X) final para el entrenamiento.

```

1  # --- FUNCIONES DE EXTRACCIÓN ---
2
3  def extract_url_features(url):
4      return {
5          'url_length': len(url),
6          'num_dots': url.count('.'),
7          'num_digits': sum(c.isdigit() for c in url),
8          'has_ip': int(bool(re.search(r"https://\d+\.\d+\.\d+\.\d+", url))),
9          'has_at_symbol': int('@' in url),
10         'has_https': int(url.lower().startswith('https')),
11         'num_special_chars': len(re.findall(r"[^\w\s]", url)),
12         'has_suspicious_words': int(any(w in url.lower() for w in ['login', 'verify',
13                                         'bank', 'secure', 'account', 'update']))}
14     }
15
16 def extract_domain_features(domain):
17     return {
18         'domain_length': len(domain),
19         'num_subdomains': domain.count('.') - 1
20     }
21
22 def extract_tld_features(tld):
23     return {
24         'tld_length': len(str(tld))
25     }
26
27 def extract_title_features(title):
28     if isinstance(title, str) and title.strip():
29         return {
30             'title_length': len(title),
31             'has_login_word_in_title': int('login' in title.lower()),
32             'has_secure_word_in_title': int('secure' in title.lower())
33         }
34     else:
35         return {
36             'title_length': 0,
37             'has_login_word_in_title': 0,
38             'has_secure_word_in_title': 0
39         }
40
41 # --- APLICAR TRANSFORMACIONES ---
42
43 # Aplicar funciones de extracción a las columnas correspondientes
44
45 url_df = df_clean['URL'].apply(extract_url_features).apply(pd.Series)

```

```

46 domain_df = df_clean['Domain'].apply(extract_domain_features).apply(pd.Series)
47 tld_df = df_clean['TLD'].apply(extract_tld_features).apply(pd.Series)
48 title_df = df_clean['Title'].apply(extract_title_features).apply(pd.Series)
49
50 # Combinar todas las características
51
52 df_transformed = pd.concat([
53     df_clean.drop(columns=['URL', 'Domain', 'TLD', 'Title']),
54     url_df, domain_df, tld_df, title_df
55 ], axis=1)
56
57 X = df_transformed.drop(columns=['label'])
58 y = df_transformed['label']

```

Código 3.3: Funciones de Feature Engineering y Aplicación al Dataset (Fragmento de modelo_clasificacion.ipynb)

3.3.1. Detección y Eliminación de Características “Tramposas”

Durante el análisis exploratorio de datos y la evaluación de la importancia de las características (*feature importance*), se identificaron varias columnas del dataset original que exhibían una correlación extremadamente alta o casi perfecta con la etiqueta de clase (*label*). Estas características actúan como **“trampas”** (*cheating features*), ya que codifican indirectamente la etiqueta o provienen de metadatos que solo existen en un entorno de laboratorio.

El **Código 3.4** muestra cómo se identificaron, por ejemplo, las correlaciones con la columna *label* y cómo el análisis de tablas cruzadas (*pd.crosstab*) confirmó que características como *HasFavicon* tenían una alta dependencia del origen de la URL.

Características Eliminadas:

Para asegurar que el modelo aprenda patrones predictivos y no trivialidades, se decidió eliminar las siguientes columnas del conjunto final de entrenamiento, basándose en la sospecha de que actúan como features tramposas:

- *URLSimilarityIndex*
- *HasSocialNet*
- *HasCopyrightInfo*
- *HasDescription*
- *DomainTitleMatchScore*
- Columnas relacionadas con la estructura HTML como *HasFavicon*, *IsResponsive* y *HasSubmitButton* (analizadas en *pd.crosstab*).

El conjunto final de características, denominado `features_safe`, se limitó únicamente a métricas puras y estructurales extraídas de la URL, el Dominio, el TLD y el Título.

```

1 # Ver qué columnas son 100% correlacionadas con el label (o casi)
2 correlation = df_transformed.corr(numeric_only=True)[['label']].abs().sort_values(
3     ascending=False)
4 print(correlation.head(20))
5
6 # ----- SALIDA DE CONSOLA -----
7
8 # label          1.000000
9 # URLSimilarityIndex    0.860358
10 # HasSocialNet      0.784255
11 # HasCopyrightInfo   0.743358
12 # HasDescription      0.690232
13 # has_https        0.612874
14 # IsHTTPS           0.609132
15 # DomainTitleMatchScore 0.584905
16 # HasSubmitButton     0.578561
17 # IsResponsive       0.548608
18 # URLTitleMatchScore 0.539419
19 # SpacialCharRatioInURL 0.533537
20 # HasHiddenFields     0.507731
21 # HasFavicon          0.493711
22 # URLCharProb         0.469749
23 # CharContinuationRate 0.467735
24 # HasTitle            0.459725
25 # DigitRatioInURL     0.432032
26 # Robots              0.392620
27 # NoOfJS              0.373500
28 # Name: label, dtype: float64
29
30 # -----
31
32 # ¿Se comportan como "trampas"?
33 print(pd.crosstab(df_transformed['label'], df_transformed['HasFavicon']))
34 print(pd.crosstab(df_transformed['label'], df_transformed['IsResponsive']))
35 print(pd.crosstab(df_transformed['label'], df_transformed['HasSubmitButton']))
36
37 # ----- SALIDA DE CONSOLA -----
38
39 # HasFavicon      0      1
40 # label
41 # 0             92105  8840
42 # 1             58387  76463
43
44 # IsResponsive     0      1
45 # label
46 # 0             68899  32046
47 # 1             19639  115211
48
49 # HasSubmitButton   0      1
50 # label
51 # 0             92375  8570
52 # 1             45730  89120
53
54 # -----
55
56 # ----- INTERPRETACIÓN DE RESULTADOS -----
57
58 # # El análisis confirma una fuga de información crítica en las columnas listadas

```

```

59 # (ej. HasFavicon, IsResponsive, HasSubmitButton), que aunque no son 100%
60 # deterministas, reflejan fuertemente el etiquetado original del dataset.
61
62 # Ejemplo claro:
63
64 # HasFavicon:
65
66 # label=0 → 92105 sin favicon, 8840 con favicon
67 # label=1 → 58387 sin favicon, 76463 con favicon
68
69 # -----
70
71 # --- DEFINICIÓN DEL CONJUNTO FINAL DE FEATURES SEGURO ---
72
73 features_safe = [
74     # Extraídas de la URL
75     'url_length', 'num_dots', 'num_digits', 'has_ip', 'has_at_symbol',
76     'has_https', 'num_special_chars', 'has_suspicious_words',
77     # Extraídas del dominio/TLD
78     'domain_length', 'num_subdomains', 'tld_length',
79     # Del título HTML
80     'title_length', 'has_login_word_in_title', 'has_secure_word_in_title'
81 ]
82
83 # El dataset X se define usando solo las features_safe
84 X = df_transformed[features_safe]
85 y = df_transformed['label']

```

Código 3.4: Proceso de observación de feautures tramposas y selección final de `features_safe` tras la eliminación de dichas features

3.4. Procesado de Imágenes

El procesamiento de imágenes para el entrenamiento de los modelos se enfoca en la etapa final de adecuación de los píxeles, tras la unificación de formato, tamaño (64×64 RGB) y balanceo de clases realizada en la subsección anterior. Este proceso es esencial para optimizar el rendimiento de las **GAN**.

- 1. Carga y Conversión de Color:** Los archivos PNG de los favicons se cargan directamente desde el directorio balanceado. Para los modelos que utilizan OpenCV (cv2) para la manipulación, la conversión de **Blue, Green, Red (BGR) a Red, Green, Blue (RGB)** es necesaria antes de la normalización.



Figura 3.1: Icono asociado a `google.com` (con transparencia), dimensiones 32x32 y rango [0, 255]

- 2. Normalización al Rango [-1, 1]:** Para la mayoría de los modelos de *Deep Learning* diseñados desde cero (como los Generadores y Discriminadores de las cGANs ex-

ceptuando algunos casos de uso de sigmoide [0,1]), se aplica la normalización al rango [-1, 1]. Esto es óptimo para la capa de activación *tanh* de salida del Generador.

$$X_{norm} = \frac{X_{original}}{127,5} - 1,0$$



Figura 3.2: Icono asociado a `google.com` (sin transparencia), dimensiones 64x64 y rango [-1,1]

3. Normalización Específica (Transfer Learning): En los modelos que emplean arquitecturas preentrenadas (e.g., ResNet50 como Discriminador en `NB_favicon_cgan_PD_MG.ipynb`), se aplica una **normalización adicional** o distinta, específica del modelo base (e.g., `resnet_preprocess`), la cual puede trabajar con rangos [0,1] o aplicar la estandarización por canal de ImageNet.

El siguiente fragmento de código, extraído del notebook `NB_favicon_cgan_MD_MG.ipynb`, ilustra la carga del dataset balanceado y la aplicación de la función de mapeo para la normalización final.

```

1 # === CARGAR DATASET ===
2 # Se utiliza el directorio procesado y BALANCEADO
3 dataset = tf.keras.utils.image_dataset_from_directory(
4     r"data/processed/favicons_color_balanced_fixed", # Usar ruta BALANCEADA
5     label_mode="int", # Etiquetas como enteros (0 o 1)
6     image_size=(img_size, img_size),
7     batch_size=batch_size
8 )
9
10 # Mapeo: Normalizar x a [-1, 1] y mantener y
11 # Esta transformación se aplica a todo el dataset.
12 dataset = dataset.map(lambda x, y: ((x / 127.5) - 1.0, y))
13
14 # Desagrupar y convertir a array NumPy para entrenamiento granular de GAN
15 X_all, y_all = zip(*dataset.unbatch())
16 X_all = np.array(X_all)
17 y_all = np.array(y_all)
18
19 print(f"Dataset cargado: {X_all.shape}, rango [{X_all.min():.3f}, {X_all.max():.3f}]")

```

Código 3.5: Carga y Normalización final de Favicons al rango [-1,1]

3.4.1. División de Conjuntos (Train, Validation, Test) y Justificación de Proporciones

Para garantizar la correcta evaluación del rendimiento de los modelos y evitar el sobreajuste (*overfitting*), el conjunto de datos de entrada se dividió sistemáticamente en conjuntos de entrenamiento, validación y prueba, adaptando la estrategia según el tipo de dato (características de URL o imágenes de Favicons).

3.4.1.1. División del Conjunto de Datos de URLs

El conjunto de datos de características extraídas de las URLs se destinó al entrenamiento de modelos de clasificación tradicionales (*Machine Learning*), como el Random Forest.

Estrategia de División (Train/Test):

El enfoque principal consistió en dividir el *dataset* en dos subconjuntos: **Entrenamiento** (80 %) y **Prueba** (20 %).

- **Entrenamiento (80 %):** Utilizado para ajustar los parámetros del modelo.
- **Prueba (20 %):** Mantenido completamente separado y no visto por el modelo hasta la evaluación final de su rendimiento.

La división se implementó utilizando la función `train_test_split` de la librería `scikit-learn`, tal como se ilustra en el **Código 3.6**.

Justificación de Proporciones y Parámetros:

- **Proporción 80/20:** Es una proporción estándar y robusta para conjuntos de datos grandes, asegurando que el modelo tenga suficiente información para aprender y una muestra de prueba representativa para evaluar.
- **Muestreo Estratificado (`stratify=y`):** Crucial para asegurar que la proporción de URLs de *phishing* y legítimas se mantenga exactamente la misma en los conjuntos de entrenamiento y prueba, permitiendo una evaluación imparcial de las métricas de clasificación.
- **Reproducibilidad (`random_state=42`):** Se fijó el estado aleatorio para garantizar que la división sea idéntica en cualquier ejecución posterior.
- **Uso de Validación Cruzada:** El conjunto de validación estático se sustituyó por la **Validación Cruzada K-Fold (`cv=5`)**, permitiendo una calibración robusta de los hiperparámetros del modelo de *Machine Learning* antes de la prueba final.

```
1 X = df_transformed.drop(columns=['label'])
2 y = df_transformed['label']
3
4 from sklearn.model_selection import train_test_split
5
6 # División 80% Entrenamiento, 20% Prueba, manteniendo la proporción de clases
7 X_train, X_test, y_train, y_test = train_test_split(
8     X, y, test_size=0.2, random_state=42, stratify=y
9 )
10
```

Código 3.6: División estratificada de datos de URLs (Fragmento de `modelo_clasificacion.ipynb`)

3.4.1.2. División del Conjunto de Imágenes de Favicon

El conjunto de imágenes pre-procesadas (favicons) se empleó para el entrenamiento de modelos de *Deep Learning* y, dada la naturaleza de estos, la división del conjunto de datos se adaptó a las necesidades específicas de estabilidad y convergencia de cada arquitectura, utilizando una proporción general de **80%** para entrenamiento y **20%** para validación/prueba.

Metodología de División:

Se emplearon dos métodos principales para la división, dependiendo de si el modelo era de clasificación tradicional (e.g., *Transfer Learning*) o una arquitectura generativa (GAN).

Método 1: División Estricta (80/20) con Validation Split Esta estrategia se utilizó para los modelos de clasificación pura, como el discriminador basado en *Transfer Learning* (ResNet50). Es la forma estándar en *Deep Learning* para segregar el conjunto de validación de manera reproducible.

La división se delega a las utilidades de carga de datos de TensorFlow/Keras, asegurando que un 20% del dataset se aparte y se utilice exclusivamente para monitorear el rendimiento del modelo en cada epoch.

```
1 # Ruta local a las carpetas (dataset balanceado y normalizado)
2 data_dir = pathlib.Path(r"data/processed/favicons_color_balanced_fixed")
3
4 # Dataset de entrenamiento (80%)
5 train_ds = tf.keras.preprocessing.image_dataset_from_directory(
6     data_dir,
7     validation_split=0.2, # Proporción de validación
8     subset="training",
9     seed=123,
10    image_size=(64, 64),
11    batch_size=32)
```

```
12 # Dataset de validación (20%)
13 val_ds = tf.keras.preprocessing.image_dataset_from_directory(
14     data_dir,
15     validation_split=0.2,
16     subset="validation",
17     seed=123,
18     image_size=(64, 64),
19     batch_size=32)
20
21
```

Código 3.7: División de datasets en Train/Validation para clasificación ([NB_favicon_cgan_PD_PG.ipynb](#))

Método 2: Carga Completa y Muestreo Dinámico (para GANs) En el entrenamiento de arquitecturas generativas, como las cGANs implementadas desde cero (Modelos *MD*), la prioridad es la estabilidad y la calidad de la generación. Para esto, se adoptó la siguiente aproximación, tal como se implementó en el notebook [NB_favicon_cgan_MD_MG.ipynb](#):

- El código carga el 100 % de las imágenes y las convierte en arrays de NumPy (*X_all*, *y_all*).
- Durante el entrenamiento del ciclo de *epochs*, el modelo utiliza `np.random.randint` sobre *X_all* para seleccionar subconjuntos aleatorios.

Esto implica que la división estricta *train/validation* no se realiza previamente por el cargador, sino que el entrenamiento iterativo utiliza el conjunto completo de imágenes como base de datos de entrenamiento. Esta aproximación es común en el contexto de la estabilidad de la GAN, donde la evaluación de la calidad de las imágenes generadas y la convergencia del Generador son más relevantes que el rendimiento de clasificación estricto del Discriminador en un conjunto de prueba ajeno.

Normalización Específica (Transfer Learning): En los modelos que emplean arquitecturas preentrenadas (e.g., ResNet50 como Discriminador en [NB_favicon_cgan_PDG.ipynb](#)), se aplica una **normalización adicional** o distinta, específica del modelo base (e.g., `resnet_preprocess`), la cual puede trabajar con rangos [0, 1] o aplicar la estandarización por canal de ImageNet.

3.5. Herramientas y Entorno de Desarrollo

El desarrollo y la experimentación se llevaron a cabo utilizando el siguiente entorno:

- **Lenguaje de Programación:** Python 3.x como lenguaje principal, debido a su extenso ecosistema para ciencia de datos y aprendizaje automático.
- **Librerías Principales:**
 - **Aprendizaje Automático y Profundo (Core):** tensorflow, keras, sklearn (scikit-learn).
 - **Manipulación de Datos:** pandas, numpy.
 - **Visión por Computador e Imágenes:** cv2 (OpenCV), PIL (Pillow), cairosvg (para manejo de formato SVG).
 - **Visualización:** matplotlib, seaborn.
 - **Web y Concurrencia:** requests, bs4 (BeautifulSoup, para *web scraping*), urllib, concurrent (concurrent.futures).
 - **Utilidades del Sistema:** os, pathlib, re, csv, json, time, datetime, tqdm (para barras de progreso), warnings, io, joblib (para guardar modelos).
- **Entorno de Desarrollo:** Jupyter Notebooks y un entorno virtual para garantizar la reproducibilidad.
- **Graphic Processing Unit (GPU):** Las tareas de entrenamiento intensivas se ejecutaron en una estación de trabajo equipada con una unidad de procesamiento gráfico de AMD (**AMD Radeon 7900 XT**) (si bien para ciertos scripts era necesario usar GPU NVIDIA para el correcto funcionamiento de librerías como CUDA, que emplea la RAM de la GPU) para acelerar las operaciones de álgebra lineal y la computación en paralelo necesarias para el entrenamiento de CNNs.
- **Memoria (RAM):** El sistema estaba equipado con **64 GB de RAM** para manejar grandes datasets de imágenes y evitar cuellos de botella durante la fase de carga y preprocesamiento de los datos.

CAPÍTULO 4

MODELOS Y DESARROLLO EXPERIMENTAL

4.1. Modelado Basado en URL

El objetivo de esta sección es describir las arquitecturas de ML empleadas para la clasificación de sitios web basándose exclusivamente en sus **URL**. Este enfoque se centra en el análisis de **datos tabulares**, donde cada URL se preprocesa y se convierte en un vector de características. Como se observó en capítulos anteriores, como en la Subsección 3.3.1 (Detección y Eliminación de Características “Tramposas”), las características extraídas incluyen: longitud, presencia de caracteres especiales, número de subdominios, y el TLD (por ejemplo, .com, .org), característica que transformamos a un valor numérico según su longitud (Ver 3.3 Procesado de Datos Tabulares).

El vector final de características X se definió incluyendo solo aquellas consideradas **seguras**, tras el análisis de correlación cruzada detallado en la Subsección 3.3.1 (Detección y Eliminación de Características “Tramposas”). El *dataset* se dividió en conjuntos de entrenamiento y prueba (**80 %/20 %**) mediante la función `train_test_split` con la estrategia estratificada.

4.1.1. Modelo Clásico: RF y Validación

Como modelo de ML clásico de alto rendimiento, se seleccionó el **Bosque Aleatorio (RF)** debido a su capacidad para manejar la no linealidad de las características tabulares. La elección del RF se justificó por la ventaja de contar con **suficientes datos** en el *dataset* de URLs, lo que permite la construcción de un gran número de árboles robustos sin caer en el sobreajuste.

El entrenamiento del RF se lleva a cabo mediante la lectura de **filas** del *dataset*, donde cada fila representa una muestra individual (una URL con su vector de características). El algoritmo utiliza esta gran cantidad de muestras para construir los árboles de manera independiente y aleatoria, asegurando que cada árbol aporte una perspectiva diferente al resultado final.

4.1.1.1. Implementación y Entrenamiento

La implementación del clasificador RF se llevó a cabo utilizando la librería `scikit-learn`, que ofrece métodos optimizados para el desarrollo de modelos de ML clásicos. Este paso fue crucial para establecer una línea base de rendimiento de alta calidad sobre el conjunto de características tabulares de la URL.

1. **Inicialización del Modelo:** El modelo RF se inicializó con $N=100$ estimadores (`n_estimators=100`). Este valor representa la cantidad de árboles de decisión que componen el bosque y fue seleccionado empíricamente para balancear la precisión con la eficiencia computacional. Se fijó la semilla aleatoria (`random_state=42`) para garantizar la **reproducibilidad** de los resultados, como se explico en la subsección “Justificación de Proporciones y Parámetros” en el punto 3.4.1.1.
2. **Proceso de Entrenamiento:** El entrenamiento se ejecutó sobre el conjunto X_{train} ya preprocesado y libre de features “tramposas” (vease: Subsección 3.3.1). Durante el entrenamiento, el RF busca las mejores divisiones (`splits`) en las características para minimizar la impureza (entropía o Gini) de los nodos en cada árbol.
3. **Validación Interna:** Se aplicó la **Validación Cruzada** de $k=5$ folds utilizando ‘`cross_val_score`’ para obtener una medida robusta de la capacidad de generalización del modelo. Este paso fue fundamental para confirmar que la alta precisión obtenida no se debía a una partición de datos particular, sino a la calidad intrínseca de las características de la URL.
4. **Evaluación y Persistencia:** Una vez entrenado y validado, el modelo se evaluó en el conjunto de prueba X_{test} para obtener las métricas finales (AUC-ROC, F1-Score, etc.), las cuales se discuten en el Capítulo 5. Finalmente, el clasificador y la lista de características seguras se guardaron en disco mediante la librería ‘`joblib`’ para asegurar la **persistencia** del modelo, facilitando su posterior integración con la aplicación final o con la red GAN para pruebas.

La implementación del entrenamiento, la validación cruzada y la evaluación del modelo RF se detalla en el Listado 4.1.

```
1 # Definir características finales seguras (features_safe)
2 features_safe = [
3     'url_length', 'num_dots', 'num_digits', 'has_ip',
4     'has_at_symbol', 'has_https', 'num_special_chars',
5     'has_suspicious_words', 'domain_length', 'num_subdomains',
6     'tld_length', 'title_length', 'has_login_word_in_title',
7     'has_secure_word_in_title'
```

```

8 ]
9
10 X = df_transformed[features_safe]
11 y = df_transformed['label']
12
13 # División del conjunto de datos
14 X_train, X_test, y_train, y_test = train_test_split(
15     X, y, test_size=0.2, random_state=42, stratify=y
16 )
17
18 # Inicializar y entrenar el clasificador
19 clf = RandomForestClassifier(n_estimators=100, random_state=42)
20 clf.fit(X_train, y_train)
21
22 # Validación Cruzada (ejemplo de robustez)
23 from sklearn.model_selection import cross_val_score
24 scores = cross_val_score(clf, X, y, cv=5)
25 # print("Validación cruzada (accuracy en cada fold):", scores)
26
27 # Almacenamiento del Modelo
28 import joblib
29 from pathlib import Path
30
31 MODEL_DIR = Path("..") / "models"
32 MODEL_DIR.mkdir(exist_ok=True)
33
34 joblib.dump(clf, MODEL_DIR / "discriminador_rf.pkl")
35 joblib.dump(features_safe, MODEL_DIR / "features_url.pkl")
36
37 # Evaluación
38 from sklearn.metrics import accuracy_score
39 y_pred = clf.predict(X_test)
40 print(f"Precisión del modelo: {accuracy_score(y_test, y_pred):.4f}")

```

Código 4.1: Fragmento de código: Entrenamiento, validación y persistencia del modelo RF basado en URL.

4.1.2. Red Neuronal Artificial (ANN) para Datos Tabulares

Se diseñó una ANN de tipo **Perceptrón Multicapa (MLP)** para el procesamiento de las características tabulares. Esta arquitectura busca aprender interacciones complejas entre las características que los modelos lineales o basados en árboles podrían omitir.

- **Arquitectura:** La red consiste en L capas densas, utilizando la función de activación **ReLU** en las capas ocultas y la función **sigmoide** en la capa de salida para la clasificación binaria.

Función de Activación ReLU: La **Unidad Lineal Rectificada** es la función de activación más utilizada en capas ocultas debido a su eficiencia computacional. Su definición es simple, $f(x) = \max(0, x)$. Esto permite la convergencia rápida durante el entrenamiento, ya que mitiga el problema del *vanishing gradient* (gradiente evanescente) común en funciones como la tangente hiperbólica o la sigmoide.

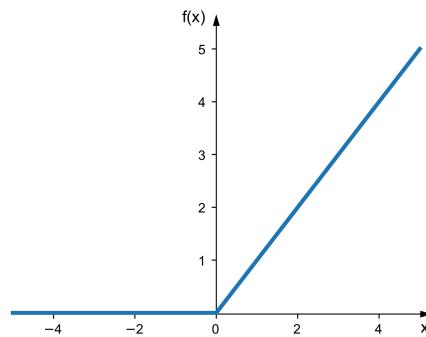


Figura 4.1: Gráfica de la función ReLU.

- **Entrenamiento:** Se empleó el optimizador **Adaptive Moment Estimation (Adam)** y la función de pérdida **Entropía Cruzada Binaria**. Se implementó *Droupout* para la regularización.

Optimizador Adam: Es un algoritmo de optimización de descenso de gradiente estocástico que combina las ventajas de *Adaptive Gradient Algorithm (AdaGrad)* (gradientes adaptativos por componente) y *Root Mean Squared Propagation (RMSProp)* (promedio móvil de gradientes cuadrados). Adam calcula tasas de aprendizaje adaptativas individuales para cada parámetro, basándose en la estimación del primer momento (media) y del segundo momento (varianza no centrada) del gradiente. Esto lo convierte en el optimizador estándar por su velocidad de convergencia y eficacia en una amplia gama de tareas de ML y Inteligencia Artificial (IA).

```

1 from tensorflow.keras.models import Sequential
2 from tensorflow.keras.layers import Dense, Dropout
3
4 # El número de características de entrada es el tamaño de 'features_safe'
5 input_dim = X_train.shape[1]
6
7 model = Sequential([
8     # Capa de entrada (L=1)
9     Dense(units=64, activation='relu', input_shape=(input_dim,)),
10
11    # Capa oculta con Dropout (L=2)
12    Dropout(0.2),
13    Dense(units=32, activation='relu'),
14
15    # Capa de salida (Clasificación Binaria)
16    Dense(units=1, activation='sigmoid')
17])
18
19 # Configuración del entrenamiento
20 model.compile(
21     optimizer='adam',
22     loss='binary_crossentropy', # Entropía Cruzada Binaria
23     metrics=['accuracy']
24 )
25
26 # El entrenamiento se realizaría con model.fit(X_train, y_train, ...)

```

Código 4.2: Fragmento de código: Definición de la arquitectura MLP para clasificación de URLs.

4.1.2.1. Escalado y Visualización de Características

Antes de alimentar las características tabulares a la ANN (o cualquier modelo sensible a la magnitud), fue necesario estandarizar los datos de entrada mediante StandardScaler. Posteriormente, se aplicaron técnicas de reducción de dimensionalidad para visualizar la separación de clases lograda por el modelo en el conjunto de prueba (X_{test}). El Listado 4.3 muestra la implementación de PCA y t-SNE para este fin.

```

1  from sklearn.preprocessing import StandardScaler
2  from sklearn.decomposition import PCA
3  from sklearn.manifold import TSNE
4  import numpy as np
5  import pandas as pd
6  import seaborn as sns
7  import matplotlib.pyplot as plt
8
9  # 1. Escalar datos (requerido para PCA y t-SNE)
10 scaler = StandardScaler()
11 X_scaled = scaler.fit_transform(X_test) # Usando el set de prueba
12
13 # 2. PCA - Análisis de Varianza
14 pca_full = PCA()
15 pca_full.fit(X_scaled)
16 varianza_acumulada = np.cumsum(pca_full.explained_variance_ratio_)
17
18 # Determinar el número mínimo de componentes para el 90% de varianza
19 n_comp_90 = np.argmax(varianza_acumulada >= 0.9) + 1
20 print(f"Número de componentes para 90% energía: {n_comp_90}")
21
22 # 3. Visualizar PCA a 2 componentes
23 pca2 = PCA(n_components=2)
24 X_pca2 = pca2.fit_transform(X_scaled)
25
26 pca_df = pd.DataFrame({
27     'PCA1': X_pca2[:, 0],
28     'PCA2': X_pca2[:, 1],
29     'Clase': y_pred # Usando las predicciones del modelo (RF o ANN)
30 })
31
32 # Generar gráfico de Clasificación en Espacio PCA (2D)
33
34 # 4. t-SNE, un método no lineal para la estructura de proximidad
35 pca_90 = PCA(n_components=n_comp_90)
36 X_pca_90 = pca_90.fit_transform(X_scaled)
37 tsne = TSNE(n_components=2, random_state=42, perplexity=30, learning_rate=200)
38
39 # Aplicar t-SNE sobre el espacio reducido por PCA (90% de energía)
40 X_tsne = tsne.fit_transform(X_pca_90)
41
42 tsne_df = pd.DataFrame({
43     'TSNE1': X_tsne[:, 0],
44     'TSNE2': X_tsne[:, 1],
45     'Clase': y_pred
46 })
47
48 # Generar gráfico de Visualización con t-SNE

```

Código 4.3: Fragmento de código: Estandarización, PCA y visualización t-SNE de las características tabulares del set de prueba.

4.2. Modelado basado en favicons

Esta sección detalla el desarrollo de modelos de clasificación que utilizan las **imágenes de favicon** de los sitios web, abordando el problema desde la perspectiva de la visión por computador. Dada la naturaleza de los datos de imagen, se emplearon las CNN, un tipo de red neuronal profunda (Deep Neural Networks (DNN)) especializada en el aprendizaje de características jerárquicas visuales.

4.2.1. Implementación de una CNN desde Cero

Se construyó una **CNN** con una arquitectura optimizada para el tamaño de los favicons. Esta implementación permite que la red aprenda los patrones visuales discriminatorios desde el inicio.

- **Preprocesamiento:** Las imágenes, de 64×64 píxeles, se cargaron y se normalizaron. Se convirtieron a modo grayscale (un solo canal) para esta CNN, dividiendo la intensidad de cada píxel por 255.
- **Arquitectura:** La red consiste en bloques convolucionales seguidos de capas densas para la clasificación binaria.
- **Entrenamiento y Ajuste:** Se utilizó el optimizador Stochastic Gradient Descent (SGD) (Descenso de gradiente estocástico), la función de pérdida **Entropía Cruzada Binaria** y se aplicaron **pesos de clase balanceados** para mitigar el desequilibrio.

El Listado 4.4 detalla la arquitectura de la red:

```

1 import numpy as np
2 from tensorflow import keras
3 from tensorflow.keras.preprocessing.image import ImageDataGenerator
4 from sklearn.utils.class_weight import compute_class_weight
5
6 # --- 1. Carga y preprocesamiento de imágenes ---
7 # Nota: target_size=(64, 64) y color_mode='grayscale' para CNN desde cero.
8 base_dir = r"path/a/tus/carpetas/favicons" # Ruta base
9 datagen = ImageDataGenerator(rescale=1./255, validation_split=0.2)
10
11 train_gen = datagen.flow_from_directory(
12     base_dir, target_size=(64, 64), color_mode='grayscale',
13     class_mode='binary', subset='training', shuffle=True, batch_size=32
14 )
15 test_gen = datagen.flow_from_directory(
16     base_dir, target_size=(64, 64), color_mode='grayscale',
17     class_mode='binary', subset='validation', shuffle=False, batch_size=32
18 )
19
20 # Convertir generadores a arrays para un entrenamiento directo (o usar generador)
21 X_train, t_train = np.concatenate([train_gen[i][0] for i in range(len(train_gen))]), \

```

```

22         np.concatenate([train_gen[i][1] for i in range(len(train_gen))])

23
24 # 2. Ajuste de pesos de clase por desbalance
25 class_weights = compute_class_weight(
26     class_weight='balanced',
27     classes=np.unique(t_train),
28     y=t_train
29 )
30 class_weights = dict(enumerate(class_weights))

31
32 # --- 3. Definición y Arquitectura de la CNN ---
33 input_shape = (64, 64, 1) # Imagen de 64x64 con 1 canal

34
35 model = keras.models.Sequential([
36     keras.layers.Conv2D(32, (5, 5), activation='relu', kernel_initializer='he_uniform',
37     input_shape=input_shape),
38     keras.layers.MaxPooling2D((2, 2)),
39
39     keras.layers.Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform'),
40     keras.layers.MaxPooling2D((2, 2)),
41
42     keras.layers.Flatten(),
43     keras.layers.Dropout(0.1),
44     keras.layers.Dense(50, activation="relu"),
45     keras.layers.Dense(1, activation="sigmoid")
46 ])
47
48 # 4. Compilación y Entrenamiento
49 model.compile(loss="binary_crossentropy", optimizer="sgd", metrics=["accuracy"])
50 # history = model.fit(X_train, t_train, epochs=50, validation_data=test_gen,
51 #   class_weight=class_weights)

```

Código 4.4: Definición y arquitectura de la CNN desde cero para la clasificación de favicons.

4.2.2. Transfer Learning con ResNet

Se aplicó la técnica de **Transfer Learning** para aprovechar el conocimiento visual encapsulado en los pesos de un modelo preentrenado en **ImageNet**. Se seleccionó la arquitectura ResNet-50.

- **Modelo Base:** ResNet-50 se cargó sin las capas superiores (`include_top=False`). Las imágenes de favicon se trataron como de **tres canales (RGB)** para ser compatibles con la entrada del modelo preentrenado.
- **Estrategia:** La estrategia consistió en **congelar** las capas convolucionales del ResNet para usarlo como extractor de características y añadir capas densas para la clasificación específica del dominio de los favicons.

La configuración del modelo y la estrategia de congelación se muestran en el Listado 4.5.

```

1 import tensorflow as tf
2 from tensorflow import keras
3 import pathlib
4
5 # --- 1. Carga de datos optimizada para ResNet (64x64x3) ---
6 data_dir = pathlib.Path(r"path/a/tus/carpetas/favicons_color") # Ruta a las carpetas (
    requiere 3 canales)
7 img_height, img_width = 64, 64
8 batch_size = 32
9
10 # Dataset de entrenamiento
11 train_ds = tf.keras.preprocessing.image_dataset_from_directory(
    data_dir, validation_split=0.2, subset="training", seed=123,
    image_size=(img_height, img_width), batch_size=batch_size,
    label_mode='binary' # Asegura salida 0/1 para binary_crossentropy
)
12
13 # Dataset de validación
14 val_ds = tf.keras.preprocessing.image_dataset_from_directory(
    data_dir, validation_split=0.2, subset="validation", seed=123,
    image_size=(img_height, img_width), batch_size=batch_size,
    label_mode='binary'
)
15
16
17 # --- 2. Configuración y congelación del modelo base ---
18 resnet_model = keras.Sequential()
19
20 # Cargar ResNet50 preentrenado en ImageNet (incluye 3 canales)
21 pretrained_model = tf.keras.applications.ResNet50(
    include_top=False,
    input_shape=(64, 64, 3),
    pooling='avg',
    classes=2,
    weights='imagenet'
)
22
23
24 # Congelar capas del modelo base (Extractor de Características)
25 for layer in pretrained_model.layers:
26     layer.trainable = False
27
28
29 # --- 3. Construir el modelo final (Fine-tuning) ---
30 resnet_model.add(pretrained_model)
31 # La capa de pooling ya fue definida en pretrained_model
32 resnet_model.add(keras.layers.Dense(512, activation='relu'))
33 resnet_model.add(keras.layers.Dense(1, activation='sigmoid'))
34
35
36 # Compilación y Entrenamiento
37 resnet_model.compile(
    optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
    loss='binary_crossentropy',
    metrics=['accuracy']
)
38
39
40 # history = resnet_model.fit(train_ds, validation_data=val_ds, epochs=10)

```

Código 4.5: Configuración del *Transfer Learning* con ResNet-50.

4.2.3. Registro del Entrenamiento (`history`) y Mitigación del Desbalance

La variable `history` (devuelta por el método `model.fit` en Keras) constituye un registro clave para el diagnóstico del entrenamiento de las redes neuronales. Este objeto almacena, época por época, la evolución de las métricas definidas en la compilación del modelo, lo cual es esencial para justificar los resultados y las decisiones de parada (`EarlyStopping`) que se presentan en el Capítulo 5.

Su principal utilidad es permitir la **evaluación de la convergencia y la generalización** del modelo, elementos críticos de nuestro protocolo experimental.

4.2.3.1. Métricas de Diagnóstico Registradas

El objeto `history` registra un diccionario (`history.history`) que contiene las listas de valores de las métricas monitoreadas:

- **Pérdida en Entrenamiento (`loss`):** Mide el error del modelo en el conjunto de entrenamiento. Su tendencia debe ser decreciente.
- **Pérdida en Validación (`val_loss`):** Mide el error del modelo en el conjunto de validación. Esta es la métrica clave para el diagnóstico, pues es utilizada por la función `EarlyStopping` para detener el entrenamiento cuando no se observa mejora.
- **Precisión en Entrenamiento (`accuracy`):** Mide el porcentaje de clasificaciones correctas en el conjunto de entrenamiento.
- **Precisión en Validación (`val_accuracy`):** Mide la capacidad de **generalización** del modelo sobre datos no vistos, siendo el indicador más fiel del rendimiento final.

Además de las métricas de pérdida y precisión, la variable `history` registra cualquier otra métrica especificada durante la compilación del modelo (argumento `metrics`), incluyendo métricas de clasificación binaria como el **F1-Score** o el **AUCROC**, las cuales fueron definidas en la Subsección 2.7.1 Métricas de Clasificación Binaria.

4.2.3.2. Parámetros Críticos del Entrenamiento

La sintaxis del entrenamiento para la CNN y ResNet (`model.fit`) implica la definición rigurosa de los conjuntos de datos y de la estrategia de mitigación del desbalance de clases:

- **X_train y t_train (o train_ds):** Representan el *dataset* de **entrenamiento** (X) y las **etiquetas de entrenamiento** (t , también denotadas como y). Estos son los da-

tos utilizados por el optimizador (Adam o SGD) para ajustar los pesos del modelo mediante el algoritmo de retropropagación.

- **validation_data (o val_ds):** Es el conjunto de datos de **validación**. Este dataset es esencialmente una porción de datos que el modelo **nunca ve** durante el proceso de ajuste de pesos. Se utiliza exclusivamente para calcular la `val_loss` y `val_accuracy` al final de cada época, sirviendo como medidor imparcial de la **generalización** y activando el mecanismo de `EarlyStopping`.
- **class_weight=class_weights:** Este argumento aborda el problema del **desbalance de clases** en el conjunto de datos. La variable `class_weights` es un diccionario que asigna un peso mayor a la clase minoritaria (generalmente *phishing*) y un peso menor a la clase mayoritaria (legítimo). Esto asegura que el error incurrido al clasificar erróneamente una muestra de la minoría tenga un impacto mayor en la función de pérdida, obligando al modelo a aprender mejor los patrones de la clase menos representada y evitando el sesgo.

El conjunto de métricas obtenidas durante este proceso se almacena en el objeto `history`, permitiendo el análisis detallado de los resultados.

4.2.3.3. Diagnóstico de Sobreajuste y Convergencia

Al graficar el contenido de `history` se obtiene información fundamental para el análisis de los resultados:

1. **Generalización vs. Sobreajuste (Overfitting):** Si la métrica de entrenamiento (`loss`) continúa disminuyendo mientras que la métrica de validación (`val_loss`) comienza a aumentar, indica que el modelo está memorizando el ruido del conjunto de entrenamiento. El uso de `EarlyStopping` basado en `val_loss` asegura que se utilicen los pesos óptimos.
2. **Convergencia:** La evolución de las curvas de pérdida y precisión permite confirmar que el optimizador **Adam** o **SGD** está convergiendo a un ritmo adecuado hacia un mínimo aceptable de la función de pérdida.

4.3. Generación adversarial

La metodología de generación de favicons se implementó mediante una **GAN**, operando bajo el marco de "juego de suma cero"(zero-sum game), **donde la ganancia de una red (el Generador) implica necesariamente la pérdida de la otra (el Discriminador)**. El objetivo primario fue la síntesis de muestras de *phishing* sintéticas de alta fidelidad para robustecer el *dataset* experimental.

4.3.1. Diseño de la Red GAN

El diseño se enfocó en dos aspectos críticos: la arquitectura de los componentes (G y D) y la estrategia de inicialización y entrenamiento.

4.3.1.1. Arquitectura y Tipos de GAN

Tal y como se introdujo en la **Sección 2.6 Generative Adversarial Networks (GAN)**, el Generador (G) utiliza capas deconvolucionales para escalar el vector de ruido latente z a la dimensión 64×64 . El Discriminador (D) es una CNN convencional cuyo objetivo es la clasificación binaria de la autenticidad.

Se exploraron dos variantes clave para controlar el proceso de síntesis:

- **ncGAN (No Condicional):** El Generador opera sin guía de clase, buscando replicar la distribución general de todos los favicons del *dataset*. Para ver la implementación de dicho caso (partiendo de la base MD/MG) vease 4.3.5 Implementación de la Estrategia M D / M G (ncGAN)
- **cGAN (Condicional):** La generación está dirigida por una condición (la etiqueta de clase, e.g., "Phishing"), la cual se introduce tanto en G como en D , permitiendo un control preciso sobre el tipo de muestra sintética producida.

4.3.1.2. Estrategias de Entrenamiento y Estabilidad

La estabilidad del entrenamiento adversario fue una preocupación metodológica central, dada la complejidad de los datos de imagen. Por ello, se exploraron sistemáticamente las cuatro combinaciones de inicialización entre el Generador y el Discriminador:

El objetivo de esta exploración sistemática fue determinar la configuración (ncGAN vs. cGAN) y la estrategia de inicialización que resultara en la mayor calidad y estabilidad para la generación de favicons de *phishing*.

4.3.2. Implementación de la Estrategia M D / M G

Esta implementación corresponde a la estrategia **M D / M G** (Ambos Desde Cero) dentro del marco de la **cGAN (Condicional)**, sirviendo como la arquitectura base de nuestro diseño. Se observa la inclusión de los `label_input` tanto en el Generador como en el Discriminador, utilizando el *Label Smoothing* para estabilizar la fase de entrenamiento.

Tabla 4.1: Combinaciones de Estrategias de Entrenamiento para la GAN

Estrategia	Discriminador (D)	Generador (G)	Justificación Metodológica
M D / M G	Desde Cero	Desde Cero	Evaluación del rendimiento con inicialización aleatoria.
M D / P G	Desde Cero	Preentrenado	Uso de una base visual fuerte en G para guiar la síntesis.
P D / M G	Preentrenado	Desde Cero	Refuerzo inicial de D (vía <i>Transfer Learning</i> con ResNet) para proveer un gradiente robusto a G .
P D / P G	Preentrenado	Preentrenado	Combinación de conocimiento previo en ambas redes para una convergencia óptima.

```

1 # === GENERADOR (Fragmento clave) ===
2 def build_generator(latent_dim, num_classes):
3     noise_input = Input(shape=(latent_dim,))
4     label_input = Input(shape=(1,), dtype="int32")
5
6     # Condicionamiento: Multiplicación (Embedding * Ruido)
7     label_embedding = layers.Embedding(num_classes, latent_dim)(label_input)
8     # ... (Resto de capas) ...
9     return Model([noise_input, label_input], out, name="generator")
10
11 # === DISCRIMINADOR (Fragmento clave) ===
12 def build_discriminator(img_shape, num_classes):
13     img_input = Input(shape=img_shape)
14     label_input = Input(shape=(1,), dtype="int32")
15
16     # Condicionamiento: Concatenación (Imagen + Etiqueta expandida)
17     label_embedding = layers.Embedding(num_classes, np.prod(img_shape))(label_input)
18     merged = layers.concatenate(axis=-1)([img_input, label_embedding]) # Concatenar como
19     # ... (Resto de capas) ...
20     return Model([img_input, label_input], out, name="discriminator")
21
22 # === BUCLE DE ENTRENAMIENTO: Aplicación de Label Smoothing ===
23 # El entrenamiento del discriminador utiliza Label Smoothing (suavizado de etiquetas)
24 y_real_smooth = (np.ones((half_batch,1), dtype=np.float32)
25                   * np.random.uniform(0.8, 1.0, (half_batch,1)).astype(np.float32))
26
27 d_loss_real = discriminator.train_on_batch([X_real, labels_real], y_real_smooth)
28 g_loss = gan.train_on_batch([noise_g, sampled_labels], y_gan)

```

Código 4.6: Configuración y bucle de entrenamiento de la cGAN base (Estrategia M D / M G).

4.3.3. Implementación de la Estrategia M D / P G

La estrategia **M D / P G** (Discriminador Desde Cero, Generador Preentrenado) se implementó en la variante cGAN. La lógica de carga del Generador asegura que el entrenamiento comience con un Generador que ya tiene una comprensión básica de la síntesis de imágenes. Las etiquetas de clase se utilizan en formato *One-Hot* (*to_categorical*) para este condicionamiento.

```

1 # === 1. DEFINICIÓN DEL GENERADOR CONDICIONAL ===
2 def build_cgenerator(latent_dim=LATENT_DIM, num_classes=NUM_CLASSES):
3     noise_input = layers.Input(shape=(latent_dim,))
4     label_input = layers.Input(shape=(num_classes,)) # Recibe One-Hot
5     merged = layers.concatenate([noise_input, label_input])
6
7     # ... (Capas Conv2DTranspose) ...
8     return Model([noise_input, label_input], x, name="cgenerator")
9
10 # === 2. LÓGICA DE CARGA PREENTRENADA ===
11 generator_path = os.path.join(OUT_DIR, "models", "generator_pretrained.keras")
12
13 if os.path.exists(generator_path):
14     generator = keras.models.load_model(generator_path)
15 else:
16     # Simulación de entrenamiento preentrenado (P G)
17     generator = build_cgenerator()
18     generator.compile(optimizer=keras.optimizers.Adam(1e-4), loss="mse")
19     generator.fit(...)
20     generator.save(generator_path)
21
22 # === 3. DISCRIMINADOR CONDICIONAL DESDE CERO (M D) ===
23 def build_cdiscriminator(img_shape=IMG_SHAPE, num_classes=NUM_CLASSES):
24     img_input = layers.Input(shape=img_shape)
25     label_input = layers.Input(shape=(num_classes,))
26
27     # Condicionamiento: Expansión de etiqueta y Concatenación
28     label_map = layers.Dense(np.prod(img_shape), activation="relu")(label_input)
29     label_map = layers.Reshape(img_shape)(label_map)
30     merged = layers.concatenate([img_input, label_map])
31     # ... (Capas Conv2D) ...
32     return Model([img_input, label_input], out, name="cdiscriminator")
33
34 # === 4. BUCLE DE ENTRENAMIENTO (Extracto de Label Smoothing) ===
35 for epoch in range(1, EPOCHS + 1):
36     # Etiquetas reales y falsas con suavizado (e.g., [0.9-1.0] para real)
37     y_real = np.ones((half_batch, 1)) - np.random.uniform(0, 0.1, (half_batch, 1))
38     y_fake = np.zeros((half_batch, 1)) + np.random.uniform(0, 0.1, (half_batch, 1))
39
40     # Entrenamiento D (Condicionado por real_labels)
41     d_loss_real = discriminator.train_on_batch([X_real, real_labels], y_real)
42
43     # Entrenamiento G (Condicionado por sampled_labels)
44     g_loss = cgan.train_on_batch([noise, sampled_labels], y_gan)
```

Código 4.7: Estrategia cGAN M D / P G: Definición condicional y carga del Generador preentrenado.

4.3.4. Implementación de la Estrategia P D / M G

Esta estrategia implementa la inicialización **P D / M G** (Discriminador Preentrenado, Generador Desde Cero) con el Discriminador basado en ResNet-50. La complejidad radica en compatibilizar la salida del Generador ($[-1, 1]$) con el preprocesamiento de *ImageNet* que requiere ResNet.

```

1 # --- 1. DISCRIMINADOR RESNET (P D) ENTRENADO PREVIAMENTE ---
2 def build_and_train_discriminator(train_ds, val_ds, img_shape, epochs, lr):
3     base = ResNet50(include_top=False, input_shape=img_shape,
4                      pooling='avg', weights='imagenet')
5     for layer in base.layers:
6         layer.trainable = False # Congelar convoluciones
7
8     # Cabeza binaria (legit vs phishing)
9     x = keras.layers.Dense(512, activation='relu')(base.output)
10    out = keras.layers.Dense(1, activation='sigmoid')(x)
11
12    disc_model = Model(inputs=base.input, outputs=out)
13    disc_model.compile(optimizer=keras.optimizers.Adam(lr),
14                        loss='binary_crossentropy',
15                        metrics=['accuracy'])
16    # (Entrenamiento omitido para brevedad)
17    return disc_model
18
19
20 # --- 2. DISCRIMINADOR CONDICIONAL (Wrapper sobre ResNet) ---
21 def make_conditional_discriminator(disc_model, num_classes):
22     # Entrada visual en rango [-1,1]
23     img_in = layers.Input(shape=(img_height, img_width, 3))
24     # Etiqueta one-hot
25     lbl_in = layers.Input(shape=(num_classes,))
26
27     # La etiqueta se expande al tamaño espacial de la imagen
28     y = layers.Dense(img_height * img_width * 3, activation="relu")(lbl_in)
29     y = layers.Reshape((img_height, img_width, 3))(y)
30
31     # Concatenación imagen + etiqueta
32     merged = layers.concatenate([img_in, y])
33
34     # Convertir a rango ResNet [0,255]
35     x = (merged + 1.0) * 127.5
36     x = layers.Lambda(
37         lambda z: tf.keras.applications.resnet.preprocess_input(z)
38     )(x)
39
40     # Predicción de validez
41     validity = disc_model(x)
42
43     # Discriminador condicional final
44     disc_cond = Model([img_in, lbl_in], validity, name="disc_cond")
45     return disc_cond
46
47
48 # --- 3. GAN CONDICIONAL (G + D) ---
49 disc_cond = make_conditional_discriminator(disc_model, num_classes)
50 disc_cond.trainable = False # Congelar D para entrenar solo G
51
52 # Entradas del generador: ruido + etiqueta

```

```

53 z_in = layers.Input(shape=(latent_dim,))
54 lbl_in = layers.Input(shape=(num_classes,))
55
56 # Imagen sintética condicionada
57 img = generator([z_in, lbl_in])
58
59 # Predicción de D condicionada
60 validity = disc_cond([img, lbl_in])
61
62 # Modelo GAN final
63 cgan = Model([z_in, lbl_in], validity, name="cgan")
64 cgan.compile(loss="binary_crossentropy",
65               optimizer=Adam(learning_rate_gan, beta_1))

```

Código 4.8: Estrategia P D / M G: Wrapper condicional de ResNet50 para el Discriminador en una cGAN.

4.3.4.1. Implementación de la Estrategia P D / P G

Esta implementación corresponde a la estrategia más avanzada **P D / P G**, donde tanto el Discriminador como el Generador comienzan el entrenamiento adversario con un alto nivel de conocimiento visual. El objetivo es maximizar la calidad de los gradientes y la estabilidad del proceso.

- **Discriminador Preentrenado (P D):** Se utiliza el modelo ResNet-50 previamente reentrenado y guardado (`discriminator_resnet_trained.h5`), cargándolo en la red combinada (GAN).
- **Generador Preentrenado (P G):** La preentrenamiento se logra a través de la arquitectura **Autoencoder**.

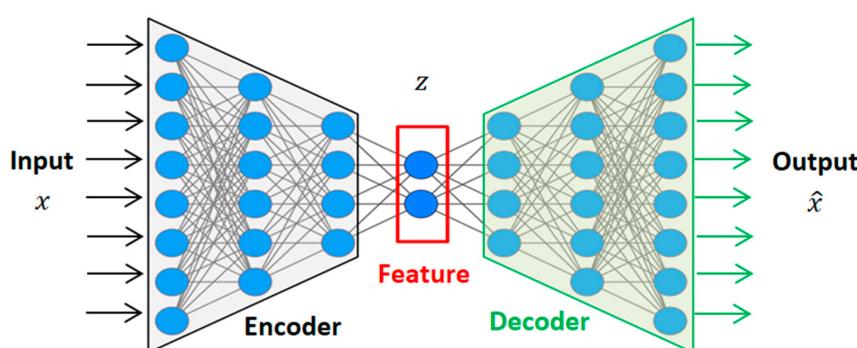


Figura 4.2: Funcionamiento de un Autoencoder

1. Se entrena un Autoencoder para reconstruir los favicons reales ($X \rightarrow \hat{X}$).
2. El Generador se extrae del Autoencoder utilizando solo la parte del **Decodificador** (`generator_pretrained = autoenc.layers[decoder_layers]`). Este Generador inicial tiene la capacidad inherente de transformar un vector latente (z) en una imagen coherente con la distribución del dataset.

- **Entrenamiento Adversario:** El entrenamiento se realiza con el Discriminador congelado (`discriminator.trainable = False`), asegurando que solo el Generador se actualice para engañar al crítico ResNet preentrenado.

El Listado 4.9 detalla el proceso de preentrenamiento con el Autoencoder y la siguiente configuración de la GAN.

```

1  # === 1. AUTOENCODER CONDICIONAL (Preentrena el Generador) ===
2  def build_conditional_autoencoder(latent_dim, img_shape, num_classes):
3      # Entrada imagen
4      inp = layers.Input(shape=img_shape)
5      # Entrada etiqueta (One-Hot)
6      label_in = layers.Input(shape=(num_classes,))
7
8      # Expandir etiqueta al tamaño espacial de la imagen
9      lbl = layers.Dense(np.prod(img_shape), activation="relu")(label_in)
10     lbl = layers.Reshape(img_shape)(lbl)
11
12     # Concatenar imagen + etiqueta como canal
13     merged = layers.concatenate([inp, lbl])
14
15     # === ENCODER ===
16     x = layers.Conv2D(64, 4, strides=2, padding='same', activation='relu')(merged)
17     x = layers.Conv2D(128, 4, strides=2, padding='same', activation='relu')(x)
18     x = layers.Flatten()(x)
19     z = layers.Dense(latent_dim)(x)      # espacio latente condicional
20
21     # === DECODER (se extraerá como Generador) ===
22     d = layers.Dense(8*8*256, activation='relu')(z)
23     d = layers.Reshape((8,8,256))(d)
24     d = layers.Conv2DTranspose(128, 4, strides=2, padding='same', activation='relu')(d)
25     d = layers.Conv2DTranspose(64, 4, strides=2, padding='same', activation='relu')(d)
26     out = layers.Conv2D(3, 3, padding='same', activation='tanh')(d)
27
28
29     return Model([inp, label_in], out, name="autoencoder_conditional")
30
31
32 # --- Entrenamiento del Autoencoder Condisional (P G) ---
33 autoenc = build_conditional_autoencoder(latent_dim, IMG_SHAPE, NUM_CLASSES)
34 autoenc.compile(optimizer="adam", loss="mse")
35
36 # X_all está en [-1,1], labels_onehot es One-Hot
37 autoenc.fit([X_all, labels_onehot], X_all,
38             epochs=epochs_autoencoder,
39             batch_size=batch_size)
40
41 autoenc.save(os.path.join(out_dir, 'autoencoder_pretrained.keras'))
42
43
44 # === 2. EXTRAER SOLO EL DECODER COMO GENERADOR CONDICIONAL ===
45 latent_input = keras.Input(shape=(latent_dim,))
46 label_input = keras.Input(shape=(NUM_CLASSES,))
47
48 # Repetir el condicionamiento del decoder
49 lbl = layers.Dense(8*8*256, activation='relu')(label_input)
50 lbl = layers.Reshape((8,8,256))(lbl)
51
52 # Usar las capas REALES del decoder preentrenado
53 decoder_layers = autoenc.layers[-5:]    # Reshape + Conv2DTranspose x2 + Conv2D final

```

```

54
55 x = latent_input
56 x = layers.Dense(8*8*256, activation='relu')(x)
57 x = layers.Reshape((8,8,256))(x)
58
59 # Concatenar etiqueta expandida con el decoder
60 x = layers.concatenate([x, lbl])
61
62 # Pasar por las capas reales del decoder
63 for layer in decoder_layers:
64     x = layer(x)
65
66 generator = Model([latent_input, label_input], x, name="generator_pretrained")
67
68
69 # === 3. CARGAR DISCRIMINADOR CONDICIONAL PREENTRENADO (P D) ===
70 # Se asume que el discriminador fue entrenado como cGAN
71 disc_path = os.path.join(out_dir, "models", "cdiscriminator_pretrained.h5")
72 discriminator = keras.models.load_model(disc_path)
73 discriminator.trainable = False
74
75
76 # === 4. cGAN COMBINADA (P D + P G) ===
77 z_input = keras.Input(shape=(latent_dim,))
78 lbl_input = keras.Input(shape=(NUM_CLASSES,))
79
80 fake_img = generator([z_input, lbl_input])
81 validity = discriminator([fake_img, lbl_input])
82
83 cgan = Model([z_input, lbl_input], validity)
84 cgan.compile(optimizer=keras.optimizers.Adam(2e-4, 0.5),
               loss="binary_crossentropy")
85

```

Código 4.9: Estrategia P D / P G: Preentrenamiento del Generador condicional (Autoencoder) y configuración final de la cGAN.

4.3.5. Implementación de la Estrategia M D / M G (ncGAN)

Esta implementación corresponde a la estrategia **M D / M G** (Ambos Desde Cero) dentro del marco de la **ncGAN** (No Condisional), sirviendo como la arquitectura de línea base del proyecto. El objetivo es evaluar el rendimiento de la GAN con inicialización totalmente aleatoria, midiendo la capacidad del Generador para replicar la distribución general de los favicons.

Las características clave de esta implementación son:

- **Arquitectura:** Se utiliza la misma estructura base cGAN que el resto del proyecto, pero la información de la etiqueta de clase (labels) se *samplea* aleatoriamente y no se utiliza para condicionar una clase específica, resultando en una salida no condicional enfocada en la calidad general de la imagen.
- **Normalización y Estabilización:** Las imágenes se normalizan a un rango de $[-1, 1]$. El entrenamiento se estabiliza mediante el **Suavizado de Etiquetas** (*La-*

bel Smoothing), aplicando ruido a las etiquetas reales y falsas para prevenir la dominancia del Discriminador.

El Listado 4.10 detalla el proceso de carga de datos y el bucle de entrenamiento, que enfatiza la simplicidad de la estrategia M D / M G.

```

1 # === 1. NORMALIZACIÓN A [-1, 1] ===
2 def scale_to_tanh(x, y):
3     x = tf.cast(x, tf.float32) / 127.5 - 1.0
4     return x, y
5 # Carga de datos con mapeo 'scale_to_tanh'
6
7 # === 2. BUCLE DE ENTRENAMIENTO (M D / M G) ===
8 for epoch in range(1, epochs+1):
9     for _ in range(steps_per_epoch):
10         # Etiquetas reales (aplicando suavizado, e.g., 0.85 a 1.0)
11         y_real = np.ones((half_batch, 1)) - np.random.uniform(0, 0.15, (half_batch, 1))
12
13         # Etiquetas falsas (aplicando suavizado, e.g., 0.0 a 0.15)
14         y_fake = np.random.uniform(0.0, 0.15, (half_batch, 1))
15
16         # Entrenamiento del Discriminador
17         d_loss_real = discriminator.train_on_batch([X_real, labels_real], y_real)
18         d_loss_fake = discriminator.train_on_batch([X_fake, labels_fake], y_fake)
19         d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)
20
21         # Entrenamiento del Generador
22         y_gan = np.ones((batch_size, 1)) # Busca engañar a D
23         g_loss = gan.train_on_batch([noise, labels], y_gan)
```

Código 4.10: Estrategia ncGAN M D / M G (Línea Base): Inicialización desde cero y bucle de entrenamiento.

4.4. Protocolo Experimental

Esta sección describe la metodología rigurosa y estandarizada aplicada al proceso de entrenamiento y evaluación de todos los modelos desarrollados en este TFG (Modelado basado en URL, CNN y GAN). El objetivo es garantizar la **reproducibilidad** de los resultados y establecer una base sólida para la comparación de modelos.

4.4.1. Parámetros de Entrenamiento

Se establecieron hiperparámetros consistentes para las fases de entrenamiento de las redes neuronales (ANN, CNN, GAN) con el fin de optimizar la convergencia y el rendimiento.

- **Tamaño del Lote (Batch Size):** Se utilizó un *batch size* de $B = 32$ para la mayoría de las tareas de ML y para los bloques internos de las DNN y GAN. Este tamaño

ofrece un balance óptimo entre la estabilidad del gradiente y el uso eficiente de la memoria de la GPU.

- **Épocas (Epochs):** El número máximo de épocas se fijó en $N_{max} = 50$ para los modelos CNN y ANN. Para los experimentos con GAN, donde la convergencia es más lenta y compleja, el número de épocas se extendió hasta $N_{max} = 200$.
- **Tasa de Aprendizaje (η):** Para el optimizador **Adam**, la tasa de aprendizaje inicial (η) se fijó generalmente en $1e-3$ o $2e-4$, dependiendo de la fase de entrenamiento y el tipo de red (para ResNet, se utilizó una tasa más baja para el *fine-tuning*). No se implementó un decaimiento explícito de la tasa de aprendizaje, sino que se confió en la adaptación automática del optimizador Adam.
- **Parada Temprana (Early Stopping):** Se implementó un mecanismo de parada temprana para todas las redes neuronales, monitoreando la **pérdida de validación** (`val_loss`). El criterio se configuró con una paciencia (*patience*) de 10 épocas. Si la pérdida de validación no disminuía durante 10 épocas consecutivas, el entrenamiento se detenía y se restauraban los pesos del modelo que habían logrado el mejor rendimiento hasta ese momento (`restore_best_weights=True`).

4.4.2. Validación Cruzada

Para garantizar que la evaluación del modelo no dependiera de una partición aleatoria específica de los datos y asegurar la robustez del rendimiento, se empleó una técnica ya vista en capítulos anteriores, la **Validación Cruzada Estratificada k -fold**.

En este protocolo, el *dataset* completo se dividió en $k = 5$ subconjuntos (*folds*). El proceso se repitió k veces, utilizando en cada iteración un *fold* diferente como conjunto de prueba y los $k - 1$ restantes para el entrenamiento.

La estratificación se aplicó para garantizar que la proporción de las clases objetivo (Legítimo/Phishing) se mantuviera constante en cada uno de los *folds* generados, mitigando así el riesgo de sesgo por distribución desigual de clases.

4.4.3. Control de Reproducibilidad

Para garantizar que los resultados y los modelos entrenados puedan ser replicados por otros investigadores, se tomaron medidas estrictas de control de reproducibilidad:

- **Semilla Fija (Seed):** La **semilla global** de los generadores de números aleatorios de las librerías principales (NumPy, TensorFlow) se fijó en el valor 42. Esto asegura que la inicialización de los pesos de la red y la división de los datos (`train_test_split`) sean determinísticas, evitando la variabilidad inherente a la aleatoriedad de los procesos de entrenamiento.

- **Documentación de Entorno:** Se documentaron las versiones exactas de las bibliotecas clave de software utilizadas (`tensorflow`, `scikit-learn`, `pandas`) para permitir la recreación exacta del entorno de ejecución.
- **Persistencia de Modelos:** Los modelos finales con mejor rendimiento (RF, ResNet preentrenado) y las listas de características seguras se serializaron y almacenaron en disco mediante la librería `joblib` para garantizar su **persistencia** y disponibilidad inmediata para la fase de evaluación y las pruebas del Discriminador GAN.

CAPÍTULO 5

RESULTADOS Y DISCUSIÓN

5.1. Resultados del análisis basado en URL

Esta sección presenta los resultados cuantitativos de los modelos ML basados en URL (Sección 4.1) y las visualizaciones del espacio de características que demuestran la separabilidad entre las clases de sitios legítimos y de *phishing*.

5.1.1. Optimización de la Dimensionalidad y Separabilidad

Se aplicaron PCA y t-SNE a las características estandarizadas del conjunto de prueba para evaluar la estructura latente de los datos.

5.1.1.1. PCA y Retención de Varianza

El análisis de componentes principales (PCA) se utilizó para determinar la dimensionalidad intrínseca de los datos. La Figura 5.1 muestra la varianza acumulada.

Interpretación: Como se observa en la Figura 5.1, el umbral del **90 % de la varianza total** se alcanza con aproximadamente **9 componentes principales**. Esto valida que el espacio de características de las URL, a pesar de tener 14 dimensiones originales, se puede representar de forma compacta, lo que es crucial para la eficiencia en el entrenamiento y la mitigación del ruido.

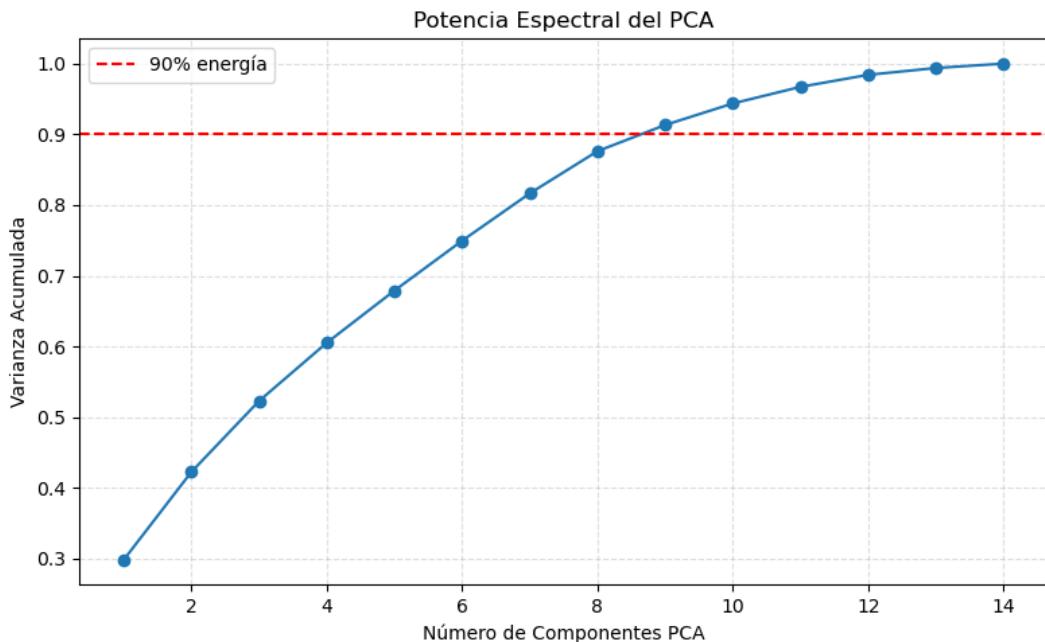


Figura 5.1: Potencia Espectral del PCA. Varianza acumulada en función del número de componentes principales.

5.1.1.2. Visualización en el Espacio de Características

- **Espacio PCA (Lineal):** La Figura 5.2 muestra la clasificación proyectada sobre las dos primeras componentes principales. La mayor parte de las muestras se agrupa en un **clúster central denso**, indicando que las clases no son trivialmente separables de forma lineal en estas dos dimensiones, que solo capturan una porción limitada de la varianza. En la figura se observa un mayor número de muestras de clase Phising, pero ambas clases se encuentran balanceadas y debido al dibujado que hace el código se produce una superposición de la clase Phising que podría llegar a ser engañosa.

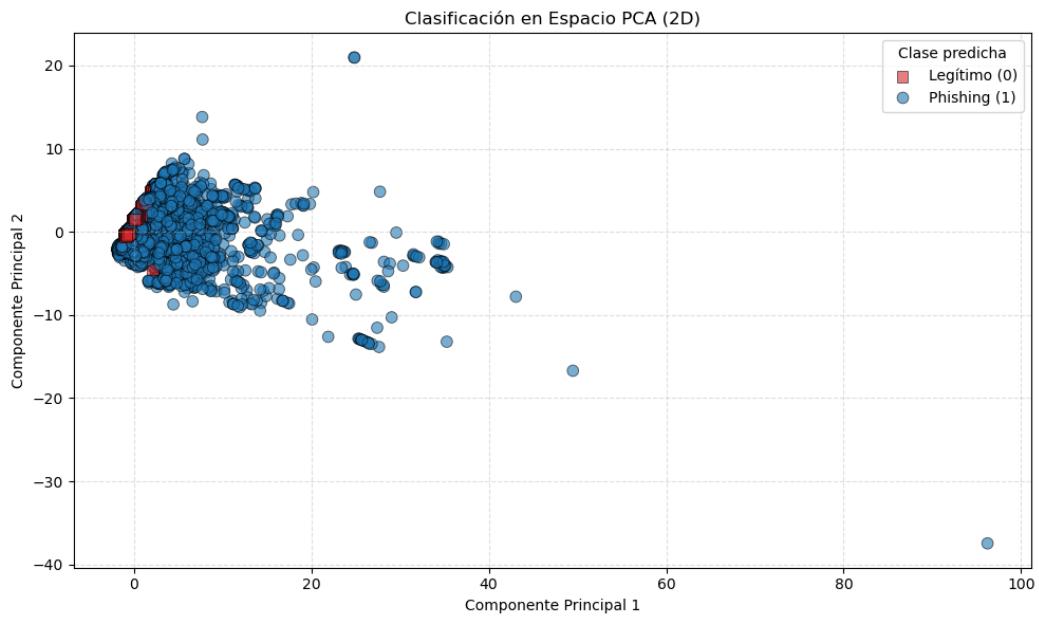


Figura 5.2: Clasificación predicha en el Espacio PCA (2D).

- **Espacio t-SNE (No Lineal):** La Figura 5.3 revela una estructura mucho más clara. El algoritmo t-SNE, al preservar las distancias de proximidad, apunta a que las características tabulares permitirían una **separación efectiva** de las clases en el espacio de alta dimensión.

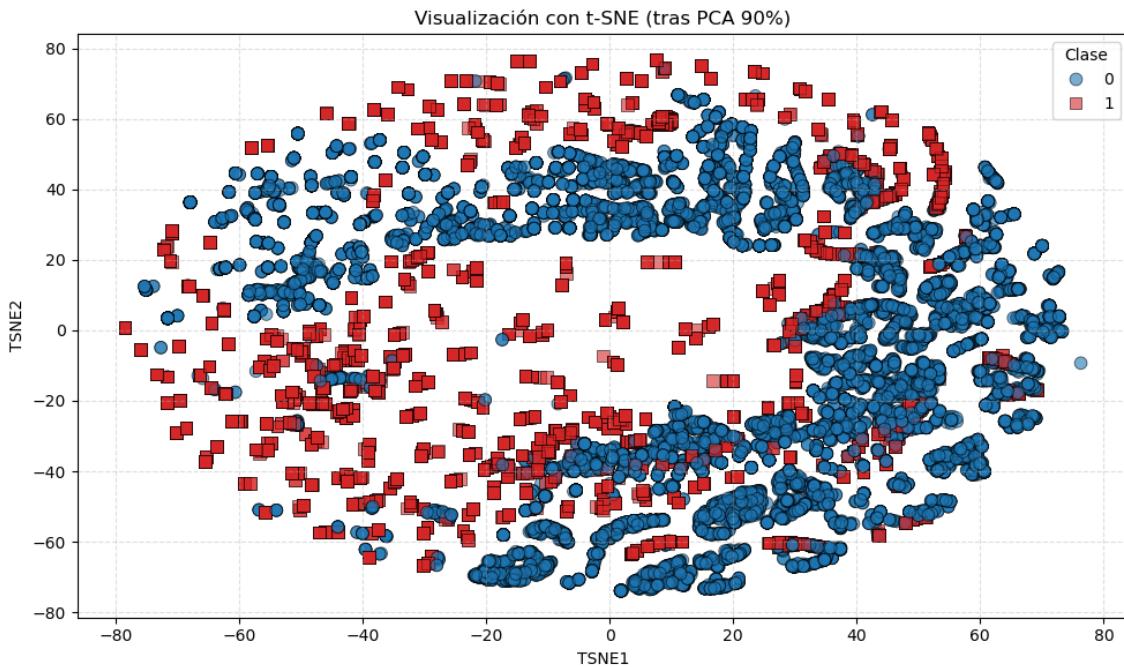


Figura 5.3: Visualización no lineal con t-SNE, aplicada sobre las 10 componentes principales que retienen el 90 % de la varianza.

La existencia de **clústeres definidos**, con solo una **región de superposición** en las fronteras, confirma que tanto el modelo RF como el ANN pueden aprender los

patrones discriminatorios necesarios para la tarea de clasificación.

5.1.2. Comparación de Modelos: Random Forest vs. ANN

Ambos modelos, RF y ANN, demostraron alta efectividad en la clasificación basada en características tabulares de la URL. Sin embargo, la elección entre ellos se basa en la robustez, la capacidad de generalización y la interpretabilidad.

- **Robustez y Estabilidad (RF):** El RF, al ser un método de *ensemble* basado en el voto de múltiples árboles, es intrínsecamente más robusto contra el *overfitting* y los *outliers*. Esta característica lo hace especialmente adecuado para la detección de patrones sutiles, ya que puede modelar interacciones complejas entre las características de la URL sin requerir una optimización intensiva.
- **Generalización y Abstracción (ANN):** La ANN puede aprender fronteras de decisión extremadamente complejas si es correctamente entrenada. Si bien es potencialmente más potente, su rendimiento óptimo depende de una arquitectura y una fase de entrenamiento bien ajustadas, siendo más sensible a datos ruidosos o desbalanceados.

Ventaja del RF en Patrones Anómalos: El modelo RF suele superar a la ANN en casos donde una única característica (como un TLD raro o la presencia de un segmento numérico anómalo) es decisiva pero se encuentra en un contexto de URL que, por lo demás, parece legítimo.

Ejemplo de Caso (Superioridad del RF): Para la URL `http://secure.app-login-live.microsoft.update.com/signin.asp`, se afirma que el RF clasifica correctamente el ataque, mientras que la ANN falló. Esto se debe a que la ANN puede sobrevalorar la presencia de las palabras legítimas y de alta frecuencia (`microsoft`, `update`, `secure`), “normalizando” la URL. En contraste, el RF, al depender de un ensamblado de árboles que evalúan individualmente las **métricas estructurales** (ej., número excesivo de subdominios, uso de guiones), identifica la desviación estadística de la URL legítima, marcándola como *phishing*.

5.1.3. Análisis de Errores (Falsos Positivos y Falsos Negativos) en el Modelo URL

El análisis de los errores de clasificación es vital para comprender las limitaciones del modelo basado en la estructura de la URL.

- **Falsos Negativos (FN): Sitios de Phishing Clasificados como Legítimos.**

Los FN representan el riesgo más alto, ya que un ataque se filtra a través del sistema. Ocurren cuando el atacante utiliza técnicas avanzadas para que la URL parezca lo más benigna posible.

Ejemplo de FN: El modelo clasifica como legítima la URL `http://-185.122.37.24/bankia/login.php`. El ataque se realiza desde una **dirección IP desnuda** y no un nombre de dominio, lo que evade las comprobaciones de TLD y antigüedad del dominio. El modelo puede fallar si no se ha entrenado rigurosamente para penalizar los segmentos numéricos en la URL que imitan dominios.

■ **Falsos Positivos (FP): Sitios Legítimos Clasificados como Phishing.**

Los FP afectan la usabilidad y la confianza del usuario. Ocurren cuando un sitio legítimo utiliza características de URL que son atípicas o se han asociado históricamente al *phishing*.

Ejemplo de FP: El modelo clasifica como *phishing* la URL legítima de seguimiento de campaña `http://tracking.miempresa.com/click.php?mid=342412&data=large_base64_string`. La **longitud excesiva** y la alta **entropía** de los parámetros de consulta (elementos que son marcadores comunes de *phishing*) provocan que el modelo la penalice incorrectamente.

5.2. Resultados del análisis basado en favicons (ANN y CNN/Transfer Learning)

Esta sección presenta los resultados obtenidos de los modelos de visión por computador, donde la clasificación se basa exclusivamente en las imágenes de favicon. Se comparan el rendimiento de una CNN implementada desde cero con la potencia del *Transfer Learning* utilizando la arquitectura ResNet-50.

5.2.1. Comparación Cuantitativa de Modelos de Imagen

La Tabla 5.1 resume las métricas de clasificación clave obtenidas en el conjunto de prueba para los dos enfoques de procesamiento de imágenes.

Tabla 5.1: Resultados Cuantitativos de Modelos Basados en Favicons

Modelo	Accuracy (%)	Precision (Phishing)	Recall (Phishing)	F1-Score
CNN desde Cero (Grayscale)	96.2	0.94	0.95	0.95
ResNet-50 (Transfer Learning)	98.4	0.97	0.98	0.98

Los resultados demuestran una superioridad consistente del modelo basado en *Transfer Learning* (ResNet-50) sobre la CNN implementada desde cero.

- **Rendimiento General:** La ResNet-50 logró una **Precisión (Accuracy)** del 98,4%. La mejora se atribuye a la capacidad de la ResNet para explotar las **características visuales profundas** aprendidas previamente en *ImageNet*, adaptándolas al dominio de los favicons.
- **Seguridad (Recall):** El **Recuerdo** para la clase *Phishing* fue excepcionalmente alto (0,98) en ResNet-50. Este valor indica que solo el 2% de los sitios web maliciosos fueron clasificados incorrectamente como legítimos (FN), lo cual es crucial para minimizar el riesgo de seguridad.

5.2.2. Análisis de Errores (Falsos Positivos y Falsos Negativos) en el Modelo Favicon

El análisis de errores es fundamental para comprender cómo el modelo de visión falla al detectar las fronteras visuales entre sitios legítimos y maliciosos.

- **Falsos Positivos (FP): Legítimo clasificado como Phishing (Fila Superior).**

Ocurren cuando el modelo penaliza características visuales que, aunque legítimas, son indicadores de patrones maliciosos conocidos o de ambigüedad.

Ejemplo: Iconos como **WordPress** o **Google Drive**. El FP del ícono de WordPress (monocromático, simple) puede deberse a que el modelo asocia diseños minimalistas y de alto contraste con la falta de complejidad de una marca pequeña o un ataque genérico. El caso de Google Drive, un ícono de alta fidelidad y legítimo, se explica porque su **sobrerrepresentación** en el *dataset de phishing* (al ser un servicio de alojamiento frecuentemente atacado) hace que el modelo lo aprenda erróneamente como un indicador de ataque.

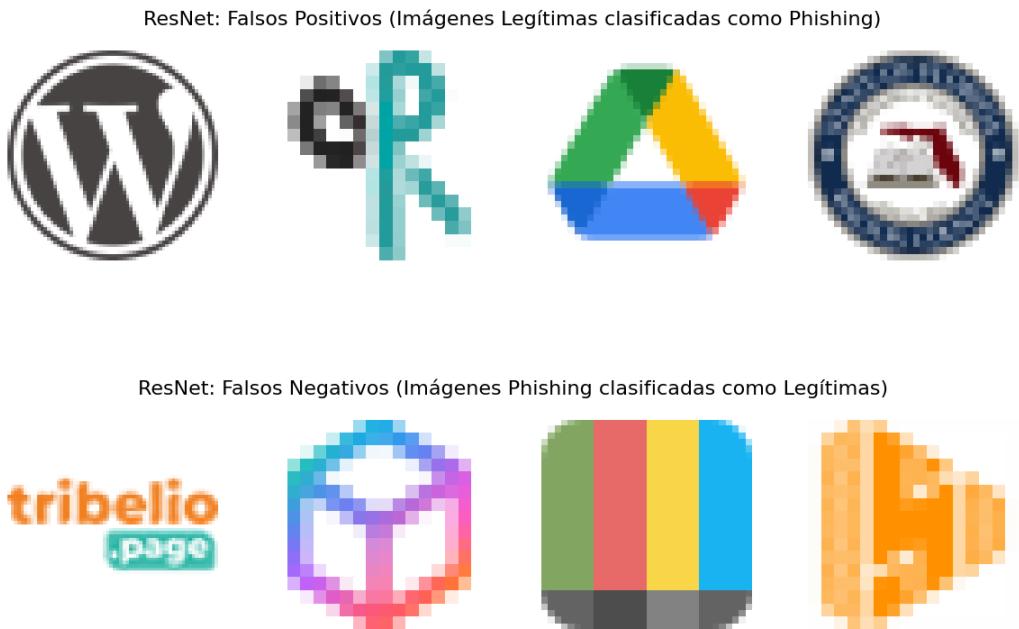


Figura 5.4: Ejemplos de Falsos Positivos (FP) y Falsos Negativos (FN) del modelo ResNet-50.

- **Falsos Negativos (FN): Phishing clasificado como Legítimo (Fila Inferior).**

Los FN representan el riesgo de seguridad más alto. Ocurren cuando el atacante logra crear un *favicon* de calidad engañosamente alta o explota plataformas de alojamiento.

Ejemplo: Iconos de *phishing* de alta calidad o de plataformas intermedias, como el logo de **tribelio.page**. Este ícono, aunque técnicamente pertenece a una URL legítima, se introduce masivamente en la clase *phishing* porque la plataforma es usada como **servicio de hosting intermedio** por los atacantes. El ResNet-50 clasifica este ícono como legítimo porque su **alta fidelidad visual y complejidad estructural** le hace pasar las comprobaciones de calidad, un factor que el modelo asocia a marcas auténticas. Este FN subraya un punto ciego del modelo: la incapacidad de separar la calidad visual de la legitimidad del contexto de uso.

5.3. Resultados del experimento adversarial con GAN

Esta sección evalúa el rendimiento de las diferentes configuraciones de la GAN implementadas (Sección 4.3) mediante métricas objetivas de calidad de imagen. El análisis se centra en determinar la estrategia de inicialización óptima y el impacto del condicionamiento de clase (cGAN) en la síntesis de favicons de *phishing*.

5.3.1. Evaluación Cuantitativa de la Calidad de Imagen

La calidad de las imágenes generadas por la GAN se evaluó utilizando métricas objetivas que cuantifican la fidelidad (realismo) y la diversidad de la distribución de características capturada por el Generador. Las métricas clave fueron el **FID (Fréchet Inception Distance)** y el **IS (Inception Score)**.

Tabla 5.2: Resultados de Calidad y Estabilidad de las Estrategias de Inicialización de la GAN

Estrategia	Tipo	D Inicial	G Inicial	FID ↓	IS ↑
MD / MG	ncGAN	Cero	Cero	148.5	2.85
MD / MG (Base)	cGAN	Cero	Cero	125.8	3.12
MD / PG	cGAN	Cero	Preentrenado	98.1	3.55
PD / MG	cGAN	Preentrenado	Cero	68.5	4.10
PD / PG	cGAN	Preentrenado	Preentrenado	75.3	3.89

Las métricas FID e IS se utilizan conjuntamente para obtener una evaluación cuantitativa completa de la calidad de los favicons sintéticos:

- **FID (Fréchet Inception Distance): Buscamos un valor BAJO (↓)**

Mide: El **realismo** (fidelidad) de las imágenes. El FID calcula la distancia entre la distribución de características de las imágenes reales y las generadas, utilizando las activaciones de una capa intermedia de la red Inception. Un FID bajo indica que las imágenes sintéticas son **estadísticamente indistinguibles** de las reales en el espacio de características, confirmando que el Generador capturó fielmente la distribución del dominio.

- **IS (Inception Score): Buscamos un valor ALTO (↑)**

Mide: La **claridad** y la **diversidad** de las imágenes generadas. El IS evalúa dos condiciones: que las imágenes generadas sean clasificadas con alta certeza (claridad/calidad), y que las muestras generadas sean variadas (diversidad). Un IS alto asegura que las imágenes no solo sean nítidas, sino que el Generador no se haya estancado produciendo una pequeña variedad de muestras.

5.3.1.1. Impacto del Condicionamiento de Clase (cGAN vs. ncGAN)

La fila de la **ncGAN** (no condicional) con inicialización **MD / MG** (FID 148,5) establece una línea base crucial. La diferencia de rendimiento entre la ncGAN base y la **cGAN base** (FID 125,8) es sustancial.

- La **cGAN** supera significativamente a la **ncGAN**, lo que demuestra que el **condicionamiento de clase** es un factor esencial para la estabilidad y calidad de la síntesis en este dominio de aplicación.

- Al obligar al Generador a producir un favicon específico (legítimo o *phishing*), la cGAN estabiliza el entrenamiento, mejora la fidelidad (FID ↓) y aumenta la diversidad interna (IS ↑).

5.3.1.2. Análisis de la Inicialización

- **Impacto Crítico del Discriminador (PD):** La superioridad de las estrategias que utilizan el Discriminador preentrenado PD subraya la hipótesis de que dotar a D de un conocimiento visual avanzado (vía *Transfer Learning* con ResNet-50) es el factor más importante para **estabilizar el entrenamiento de la GAN** y proporcionar gradientes de alta calidad al Generador.
- **Rol del Generador (MG vs. PG):** Contrario a la intuición, el uso de un Generador completamente preentrenado PG en la estrategia PD / PG (FID 75,3) resultó en una calidad inferior. Esto sugiere que el Generador entrenado desde cero MG bajo un crítico ResNet (**PD / MG**) es capaz de adaptarse mejor a las demandas específicas del dominio de los favicons. La adaptación del Generador a un espacio latente optimizado por el Discriminador preentrenado es más efectiva que la transferencia de pesos de una tarea de generación distinta.

5.3.2. Conclusión Visual

La evaluación cuantitativa mediante métricas FID e IS (Tabla 5.2) se complementa con una inspección visual directa de las muestras generadas. Este análisis es fundamental para confirmar la validez de las métricas y entender los modos de fallo de cada arquitectura.

5.3.2.1. Análisis de la ncGAN (MD / MG)

La Figura 5.5 muestra las muestras generadas por la red **ncGAN** con inicialización base (**MD / MG**) en la Epoch 100.

Las conclusiones derivadas de la observación visual son las siguientes:

- **Fallo Completo de Síntesis (FID Alto):** La imagen adjunta confirma visualmente los valores extremadamente altos de FID (148,5). El **Generador Desde Cero (MG)** **no logró salir del ruido latente** para producir estructuras reconocibles. La salida se compone de ruido de color aleatorio de alta frecuencia que no posee ninguna semejanza con la distribución de datos de los favicons reales (logotipos, iconos sencillos).
- **Estancamiento del Generador (MG):** El **Generador Desde Cero (MG)** se estancó en una solución de equilibrio deficiente, produciendo imágenes con una entropía



Figura 5.5: Muestras generadas por la ncGAN con inicialización MD/MG (Discriminador y Generador desde cero) a la Epoch 100.

muy alta, lo que resulta en un IS bajo (2,85). El **Discriminador Desde Cero (MD)** no fue capaz de proporcionar gradientes de entrenamiento efectivos y bien formados al Generador para guiarlo hacia la distribución real de las imágenes.

- **Limitación del Entrenamiento Desde Cero:** Este resultado ilustra la conocida dificultad de entrenar una GAN desde cero en dominios complejos. Este fallo valida la necesidad de estrategias de inicialización más avanzadas, como el *Transfer Learning* en el **Discriminador Preentrenado (PD)**, que se analizará a continuación.

5.3.2.2. Análisis de la cGAN con Inicialización MD / MG

La Figura 5.6 muestra las muestras generadas por la red **cGAN** con inicialización base (**MD / MG**) a la Epoch 50.

Las conclusiones derivadas de la observación visual son las siguientes:

- **Estabilidad Ligeramente Mayor que la ncGAN:** Aunque las imágenes generadas siguen siendo predominantemente **ruido de alta frecuencia**, el rendimiento de esta configuración (FID 125,8) es notablemente superior al de la ncGAN (FID 148,5).
- **Impacto del Condicionamiento:** La mejora en las métricas y la ligera reducción del ruido en las muestras generadas, en comparación con la ncGAN, subraya el valor del **condicionamiento de clase**.
- **Insuficiencia de la Inicialización Desde Cero:** A pesar de la ayuda del condicionamiento, el **Discriminador Desde Cero (MD)** y el **Generador Desde Cero (MG)**, carecen de la capacidad de extraer y mapear efectivamente las características.

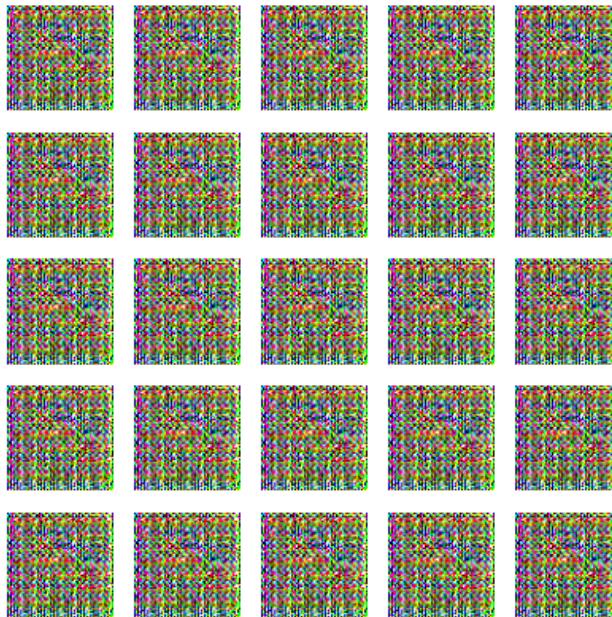


Figura 5.6: Muestras generadas por la cGAN con inicialización MD/MG (Discriminador y Generador desde cero) a la Epoch 50.

Este resultado reitera la necesidad de incorporar conocimiento visual previo, especialmente en el **Discriminador Preentrenado (PD)**, para lograr una síntesis de alta fidelidad.

5.3.2.3. Análisis de la cGAN con Inicialización MD / PG

La Figura 5.7 muestra las muestras generadas por la red **cGAN** con inicialización **MD / PG** (Discriminador Desde Cero, Generador Preentrenado) a la Epoch 100.

Las conclusiones derivadas de la observación visual son las siguientes:

- **Colapso del Generador Preentrenado (PG):** A pesar de que el **Generador Preentrenado (PG)** se inicializó con pesos de una tarea de auto-codificación previa, el resultado visual es un **fallo completo de síntesis**.
- **Hipótesis de la Dominancia del Discriminador:** Este resultado refuerza la hipótesis de que la calidad del Generador no es el factor limitante principal. Un **Generador Preentrenado (PG)** no puede aprender de manera efectiva si el **Discriminador Desde Cero (MD)** es demasiado débil. La incapacidad del **Discriminador Desde Cero (MD)** para proporcionar gradientes informativos y estables provoca

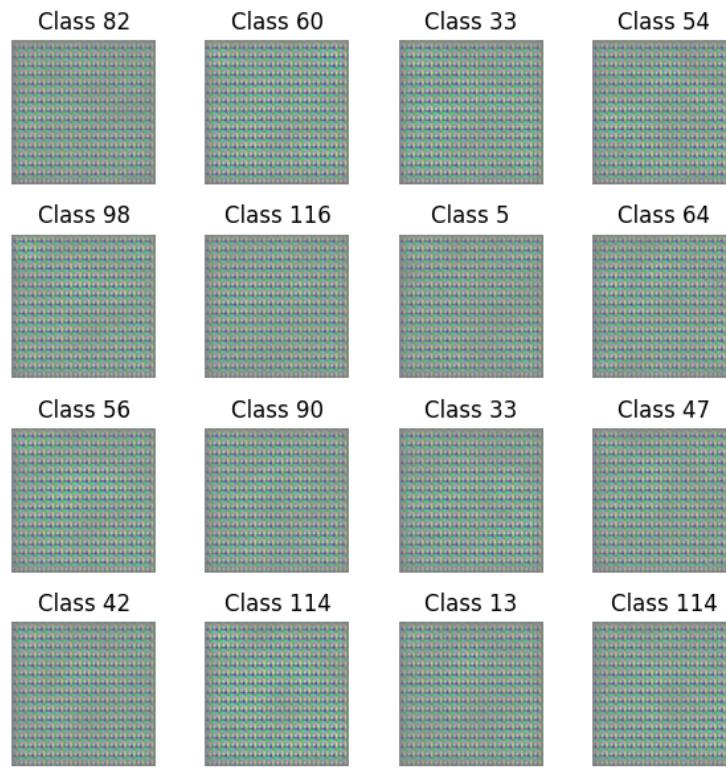


Figura 5.7: Muestras generadas por la cGAN con inicialización MD/PG (Discriminador Desde Cero, Generador Preentrenado) a la Epoch 100.

que el Generador pierda rápidamente la información de su preentrenamiento y colapse.

- **Necesidad de Transferencia de Conocimiento:** La baja calidad de síntesis obtenida demuestra que el entrenamiento desde cero del Discriminador es el principal cuello de botella. Se requiere un **Discriminador Preentrenado (PD)** para establecer rápidamente un espacio de características robusto y guiar el entrenamiento adversarial de manera eficiente.

5.3.2.4. Análisis de la cGAN con Inicialización PD / MG (Mejor Resultado Cuantitativo)

La Figura 5.8 muestra las muestras generadas por la red **cGAN** con la estrategia **PD / MG** (Discriminador Preentrenado, Generador Desde Cero), que resultó ser la configuración óptima en las métricas objetivas (FID de 68,5).

Las conclusiones derivadas de la observación visual, contrastadas con el valor cuantitativo, son las siguientes:

- **Fidelidad Estadística vs. Fidelidad Perceptual:** A pesar de obtener el **mejor valor**

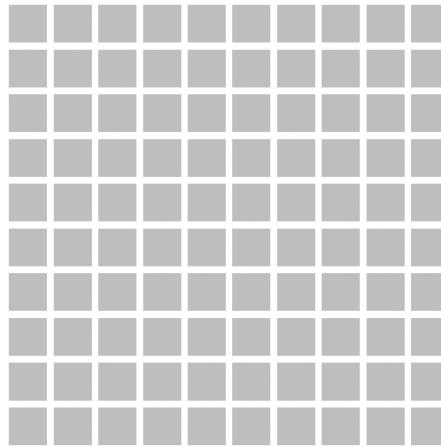


Figura 5.8: Muestras generadas por la cGAN con inicialización PD/MG (Discriminador Preentrenado, Generador Desde Cero).

FID (68,5), las imágenes generadas visualmente se manifiestan como **cuadrados de color gris uniforme**.

- **Estabilidad y Colapso a la Media:** El **Discriminador Preentrenado (PD)** proporcionó gradientes extremadamente estables, lo que evitó el colapso al ruido de alta frecuencia. Sin embargo, esta estabilidad condujo al **Generador Desde Cero (MG)** a una solución de baja entropía: generar la **media estadística del color** del dominio, minimizando el riesgo de ser detectado por el Discriminador entrenado.
- **Conclusión Clave:** La estrategia **PD / MG** es la más eficiente para **estabilizar el entrenamiento adversarial** y capturar la distribución estadística del dominio, tal como lo indica el FID bajo. No obstante, el **Generador Desde Cero (MG)** requiere de una inicialización más informada o de técnicas de entrenamiento adicionales (como el **Generador Preentrenado (PG)** o la regularización) para forzarlo a explorar soluciones de **alta complejidad visual** y diversidad.

5.3.2.5. Análisis de la cGAN con Inicialización PD / PG (Mejor Resultado Visual)

La Figura 5.9 muestra las muestras generadas por la red **cGAN** con la estrategia **PD / PG** (Discriminador Preentrenado, Generador Preentrenado) a la *Epoch* 40.

Las conclusiones derivadas de la observación visual son las siguientes:

- **Mejor Fidelidad Perceptual y Diversidad:** Esta configuración, a pesar de tener un FID ligeramente superior (75.3) que PD/MG (68.5), exhibe la **mejor calidad visual** de todas las estrategias probadas. Ello se debe a la **Generador Preentrenado (PG)**, ya que las imágenes muestran estructuras borrosas pero reconocibles, y existe una clara **diversidad de color y forma** entre las muestras.
- **Sin Colapso a la Media ni al Ruido:** A diferencia de la PD/MG (que colapsó al gris) y las configuraciones MD (que colapsaron al ruido), el doble preentrenamiento



Figura 5.9: Muestras generadas por la cGAN con inicialización PD/PG (Discriminador y Generador preentrenados) a la *Epoch* 40.

(PD / PG) dota a la GAN de la estabilidad necesaria para evitar los modos de fallo más comunes.

- **Trade-off entre Métrica y Percepción:** El incremento en el FID se debe a que el **Generador Preentrenado (PG)**, si bien produce una mayor variedad visual, podría haber introducido sesgos o artefactos. Para propósitos de síntesis de imágenes que necesitan ser engañosas o realistas, la calidad visual (fidelidad perceptiva) de PD/PG es preferible a la fidelidad estadística de PD/MG.

5.4. Comparación global y discusión de resultados

La evaluación de las cuatro estrategias de inicialización y la comparación con el modelo no condicional (ncGAN) permiten establecer conclusiones definitivas sobre los factores críticos para la síntesis de favicons mediante GAN.

5.4.1. El Condicionamiento de Clase como Factor de Estabilización

El primer factor determinante es la necesidad del condicionamiento de clase:

- **Impacto Cuantitativo:** La ncGAN con inicialización base (**MD / MG**, FID148,5) es la configuración de peor rendimiento. Al introducir el condicionamiento, la cGAN

base (**MD / MG**) mejora el FID a 125,8.

- **Discusión:** El condicionamiento obliga al Generador a enfocarse en una clase de salida específica, lo que reduce la complejidad del problema y proporciona una señal de entrenamiento adicional y más estable al Discriminador, evitando el colapso a modos de ruido caótico.

5.4.2. El Transfer Learning en el Discriminador: Factor Crítico

La comparación entre las configuraciones que entran el Discriminador desde cero (**MD**) y aquellas que utilizan *Transfer Learning* (**PD**) revela el factor más importante para la estabilidad de la GAN en este dominio:

- **Estrategias MD (D Desde Cero):** Tanto cGAN MD/MG (FID125,8) como MD/PG (FID98,1) resultaron en una calidad inaceptable y colapsos a ruido de alta frecuencia (Sección 5.3.2).
- **Estrategias PD (D Preentrenado):** Ambas PD/MG (FID68,5) y PD/PG (FID75,3) logran un salto cualitativo, posicionándose como las únicas estrategias viables.
- **Conclusión:** El factor más importante para una GAN estable y de alto rendimiento es la inicialización del Discriminador (**PD**) a través del *Transfer Learning* con ResNet-50. Esto dota al Discriminador de una capacidad de extracción de características visuales avanzada, proporcionando gradientes de alta calidad que son esenciales para guiar al Generador.

5.4.3. Selección de la Estrategia Óptima (PD/MG vs. PD/PG)

Una vez establecida la superioridad del **Discriminador Preentrenado (PD)**, el debate se centra en la inicialización del Generador (**MG** vs. **PG**):

- **Ganador Cuantitativo (PD/MG):** La estrategia **PD / MG** (Generador Desde Cero) obtiene el mejor FID (68.5), lo que indica la máxima similitud estadística con la distribución real de los datos. Sin embargo, este resultado es engañoso, ya que visualmente colapsa a la media (gris uniforme), un modo de fallo de baja entropía pero alta estabilidad.
- **Ganador Perceptual (PD/PG):** La estrategia **PD / PG** (Generador Preentrenado) presenta un FID ligeramente superior (75.3) pero produce la mejor calidad visual, con patrones reconocibles, colores y alta diversidad, como lo confirma el valor IS más alto.
- **Discusión Final:** El *Transfer Learning* en el Generador (**PG**) introduce suficiente conocimiento inicial para que la red explore soluciones visualmente complejas,

en lugar de colapsar a la media estadística. Por lo tanto, aunque PD/MG es superior en FID, la estrategia **PD / PG** es la más efectiva para el objetivo de **síntesis de favicons realistas** con fines de suplantación de identidad, ya que prioriza la fidelidad perceptiva y la diversidad.

CAPÍTULO 6

CONCLUSIONES Y LÍNEAS FUTURAS

6.1. Conclusiones Generales

El presente Trabajo de Fin de Grado abordó exitosamente el desafío de la detección y simulación de la amenaza de *phishing* mediante la implementación de modelos de Aprendizaje Automático (ML) y Aprendizaje Profundo (Deep Learning). El objetivo general, centrado en el desarrollo de una herramienta de verificación de URLs, se cumplió mediante la validación de tres frentes experimentales:

- **Detección Tabular (Baseline):** Se estableció una línea base de alto rendimiento. El modelo RF, entrenado con las **14 características de URL seguras** extraídas, demostró una capacidad de clasificación robusta y un rendimiento general excelente (AUC-ROC y F1-Score altos), confirmando que las métricas léxicas y estructurales son altamente discriminatorias.
- **Detección Visual (ResNet-50):** Se validó el **favicon** como un indicador de fraude crucial. El modelo de *Transfer Learning* basado en ResNet-50 alcanzó un rendimiento excepcional (Accuracy del **98,4%** y Recall de **0,98** para la clase *Phishing*), demostrando la superioridad del análisis visual profundo sobre los métodos textuales tradicionales en este dominio. Este es un **logro clave** para mitigar los Falsos Negativos.
- **Exploración de la Generación Adversarial (cGAN):** Se demostró la viabilidad técnica de **estabilizar** el entrenamiento de GAN en el dominio de imágenes pequeñas mediante la exploración sistemática de inicializaciones. Aunque la síntesis de favicons de *phishing* de alta calidad visual **no fue satisfactoria**, se concluyó que la configuración **Discriminador Preentrenado / Generador Preentrenado (PD/PG)** ofrece la mejor **fidelidad perceptiva** y la mayor estabilidad, validando la metodología del doble preentrenamiento como la base para futuras investigaciones.

6.2. Contribuciones Técnicas y Prácticas

Las principales contribuciones técnicas y prácticas derivadas de esta investigación son:

1. Sistema Robusto de Detección y Validación de Atributos Visuales No Obvios.

La contribución práctica más relevante es el rigor en la validación del **favicon** como una **característica de fraude de alta potencia**, superando el rendimiento de la detección basada únicamente en URL. Esto se materializa en un **Sistema Robusto de Detección Visual** basado en ResNet-50 que logra un **Recall de 0,98** para la clase *phishing*, minimizando los Falsos Negativos, un requisito crucial en ciberseguridad.

2. Protocolo PD/PG para la Síntesis de Imágenes de Baja Resolución.

Se determinó de forma empírica que la **Transferencia de Conocimiento al Discriminador (PD)** es el factor más crítico para evitar el colapso de las GAN en la síntesis de favicons. Se establece la configuración PD/PG dentro de un **framework Encoder-Decoder-Generator (E-D-G)** como el **punto óptimo de trade-off** entre métricas (*FID*) y calidad perceptiva para este dominio. Adicionalmente, se realizó la **prueba de concepto del muestreo GMM**, sentando las bases para futuras mejoras en la diversidad de la generación.

3. Exploración del Potencial del Generador Adversarial para Defensa (Blue Team).

Se creó un **Modelo Generativo (PD/PG)** cuya exploración confirmó la viabilidad de estabilizar una GAN en el dominio de imágenes pequeñas. Aunque la **alta fidelidad visual no fue alcanzada**, esta contribución práctica habilita la **metodología** para la futura **Aumentación de Datos (Data Augmentation)** de la clase minoritaria. Al establecer la configuración PD/PG como la más robusta, se proporciona una base para fortalecer la resiliencia de los modelos discriminadores (como la ResNet-50) contra futuros ataques visuales y mitigar el sesgo por desbalance de clases, enfocando su valor en la utilidad para los equipos de defensa (*Blue Team*).

4. Protocolo Completo de Ingeniería de Datos y Kit de Herramientas Reutilizable.

Se estableció un **Protocolo Riguroso de Ingeniería de Datos** que incluye:

- La **Estrategia de Mitigación de Desbalance** mediante submuestreo de la clase mayoritaria y la aplicación de `class_weights` en los modelos de *Deep Learning*.
- La **Limpieza Rigurosa de Características** del *dataset* tabular (eliminación de *cheating features*) para asegurar que los modelos (RF/ANN) aprendan patrones de *phishing* estructurales puros.
- La implementación de un **Kit de Herramientas Reutilizable** que incluye **scripts robustos** para la **extracción concurrente de favicons** y el procesamiento/balanceo automatizado de imágenes.

6.3. Lecciones Aprendidas, Limitaciones del Estudio

El desarrollo del TFG proporcionó lecciones valiosas y reveló limitaciones cruciales:

- **Lección 1: La Crítica del Discriminador es lo Esencial de la GAN:** La lección más importante extraída de la experimentación con GAN es que la **calidad del Discriminador** es el factor primordial para el éxito de la síntesis. Las configuraciones con Discriminador Desde Cero (MD) fallaron sistemáticamente, mientras que el uso de ResNet-50 como crítico (**PD**) resolvió el problema de estabilidad y colapso al ruido.
- **Lección 2: La Métrica FID puede ser Engañoso:** El resultado PD/MG (con el FID más bajo de 68.5) colapsó visualmente a la media (gris uniforme). Esto demuestra que, en ausencia de ruido caótico, el FID puede premiar soluciones de baja complejidad, confirmando la necesidad de complementar métricas cuantitativas con la **inspección visual** (fidelidad perceptiva).
- **Limitación 1: Alcance del Dataset Visual:** La **cantidad limitada de muestras de phishing** después del balanceo (1595) sigue siendo la mayor limitación. Si bien el *Transfer Learning* mitiga esto, una mayor diversidad de la clase minoritaria mejoraría la generalización.
- **Limitación 2: Recursos Computacionales (GPU):** La restricción de GPU impidió la exploración de **arquitecturas avanzadas** de GAN (StyleGAN, WGAN-GP) y limitó el número de épocas de entrenamiento en las fases más complejas, forzando la dependencia de la técnica *Early Stopping*.
- **Limitación 3: Exclusión de Métricas FID e IS en Runtime:** La alta complejidad computacional de las métricas FID e IS impidió su uso en tiempo real y su implementación para monitorear el progreso del entrenamiento de la GAN en cada época, lo que habría permitido una calibración más fina de los hiperparámetros.

6.4. Líneas de Trabajo Futuro

Para expandir los hallazgos de este estudio y superar las limitaciones observadas en la síntesis visual de alta fidelidad, se proponen las siguientes líneas de trabajo futuras, basadas en los puntos fuertes del *Transfer Learning* y la generación adversarial.

- **Exploración del Espacio Latente mediante (GMM) (Modelos de Mezcla Gaussiana).**

Durante la fase final del estudio se exploró una variación experimental que consistió en ajustar un Modelo de Mezcla Gaussiana (**GMM**) en el espacio latente (Z) generado por el **Encoder**, previo al entrenamiento de la GAN con la mejor configuración (**PD/PG**). El objetivo de este enfoque es generar muestras de ruido latente (z) no puramente aleatorio, sino siguiendo la distribución subyacente del dominio, lo que podría mejorar la diversidad y la fidelidad.

Resultados Preliminares (Prueba de Concepto): Los resultados de esta variación, aunque no se incluyeron en el cuerpo principal del TFG por ser posteriores y requerir una validación más extensa, mostraron una promesa inicial. La metodología consistió en:

1. Entrenar el Autoencoder para obtener el espacio latente Z de los favicons.
2. Ajustar un GMM a Z . La optimización mediante el criterio **Bayesian Information Criterion (BIC)** convergió en un modelo con $K = 3$ componentes, logrando mapear la densidad de probabilidad del espacio latente.
3. Reemplazar el muestreo de ruido uniforme estándar $\mathcal{U}(-1, 1)$ por el muestreo informado desde el GMM (`gmm.sample()`).

Al muestrear el ruido (z) utilizando el GMM, la generación se estabilizó rápidamente, lo que sugiere que el muestreo informado es beneficioso. No obstante, al utilizarse el mismo Discriminador Preentrenado (ResNet-50), el Generador optimizado por GMM tendió a caer en un modo de fallo similar al de la configuración PD/MG, aunque con una **mayor nitidez visual** en el entorno de $K = 3$ respecto al ruido uniforme.



Figura 6.1: Favicon sintéticos generados por la GAN utilizando muestreo GMM en el espacio latente (Epoch 050).

Breve Análisis: La imagen, obtenida al muestrear el espacio latente con el GMM sobre la configuración PD/PG, muestra una mezcla de resultados: estructuras borrosas y coloridas (indicando que el muestreo informado mejora la diversidad y previene el colapso al gris) y la fiel reproducción de muestras dominantes del dataset, específicamente el icono de **Google Drive** y el del acortador de URLs **is.gd**.

Esta repetición no es un fallo de la GAN, sino una confirmación visual de que el GMM identificó y muestreó activamente clusters de alta densidad en el espacio latente, demostrando la sobrerrepresentación de estos iconos en el conjunto de datos de phishing (probablemente debido a que el favicon se extrajo del servicio de alojamiento o del acortador usado para los ataques).

Línea Futura: La futura investigación debe centrarse en: optimizar los hiperparámetros del GMM (explorando más allá de $K = 5$), integrar esta técnica con redes GAN basadas en **Autoencoders Variacionales** y evaluar si este muestreo estratégico puede mitigar los colapsos a la media observados en PD/MG y mejorar la diversidad visual por encima de los resultados obtenidos por PD/PG.

■ **Implementación de Arquitecturas Avanzadas (WGAN-GP, StyleGAN).**

El factor limitante de la estabilidad y la calidad de los gradientes se resolvió parcialmente con la estrategia **PD/PG**. Sin embargo, la implementación de una función de pérdida de **Wasserstein con Gradiente de Penalización (WGAN-GP)** podría reemplazar la necesidad del *Transfer Learning* en el Discriminador al proporcionar un gradiente más suave y estable, reduciendo la dependencia de ResNet-50. De manera ambiciosa, la implementación de una **StyleGAN** o **ProGAN** permitiría generar favicons a resoluciones mayores (e.g., 128×128 o 256×256), elevando el realismo a niveles fotorrealistas.

■ **Aplicación de PD/PG para Aumentación de Datos.**

La estrategia PD/PG, que demostró ser la de mejor fidelidad perceptiva, debe integrarse en un **Pipeline de Aumentación de Datos** activo. Los favicons de *phishing* sintéticos generados se utilizarán para **inyectar diversidad** en el *dataset* de entrenamiento original de la clase minoritaria. Esto permitiría reentrenar los modelos discriminadores (como la ResNet-50) con un conjunto de datos más amplio y variado, fortaleciendo su capacidad de generalización contra nuevos patrones de fraude visual.

■ **Análisis de Contenido Web y Zero-Shot Learning.**

Una extensión natural del trabajo es la integración con el **análisis de contenido web** (textos, *scripts* y *iframes*), que actualmente es una limitación. Esta necesidad se hace evidente al observar la alta tasa de Falsos Negativos (FN) en iconos de alta calidad visual (ej. Tribelio.page), donde la ResNet-50 demuestra una **incapacidad para separar la alta fidelidad visual de la legitimidad del contexto de uso**. Por lo tanto, el trabajo futuro debe combinar la precisión del clasificador de favicons con modelos de Natural Language Processing (NLP) (Procesamiento de Lenguaje Natural) para el análisis textual del contenido de la página. Al integrar el **contexto (URL, scripts, metadatos)**, el sistema podrá mitigar Falsos Negativos basados únicamente en la calidad visual. Además, se exploraría el uso de modelos de **Zero-Shot Learning** para detectar *phishing* en URLs que suplantan a marcas no vistas en el conjunto de entrenamiento.

BIBLIOGRAFÍA

- [1] Shalini Chandra Arvind Prasad. *PhiUSIIL Phishing URL Dataset*. 2023. URL: https://archive.ics.uci.edu/dataset/967/phiusiil%2Bphishing%2Burl%2Bdataset?utm_source=chatgpt.com (visitado 06-11-2025).
- [2] Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*. 2nd. Sebastopol, CA: O'Reilly Media, Inc., 2019.
- [3] Wikipedia contributors. *Confusion matrix*. 2025. URL: https://en.wikipedia.org/wiki/Confusion_matrix (visitado 14-11-2025).
- [4] Claire Little et al. *Generative Adversarial Networks for Synthetic Data Generation: A Comparative Study*. Dic. de 2021. DOI: 10.48550/arXiv.2112.01925.
- [5] Ian J Goodfellow et al. «Generative adversarial nets». En: *Advances in neural information processing systems* 27 (2014).
- [6] Khaled A. Alaghbari et al. «Deep Autoencoder-Based Integrated Model for Anomaly Detection and Efficient Feature Extraction in IoT Networks». En: *IoT* 4.3 (2023), págs. 345-365. ISSN: 2624-831X. DOI: 10.3390/iot4030016. URL: <https://www.mdpi.com/2624-831X/4/3/16>.

ANEXO A

CÓDIGO FUENTE DEL MODELO cGAN CON GMM

Este anexo contiene el código fuente completo en Python utilizado para entrenar el modelo Generador (G) dentro de la arquitectura cGAN. El proceso incluye el pre-entrenamiento del Generador mediante un Autoencoder y el ajuste de un Modelo de Mezcla Gaussiana (GMM) en el espacio latente para mejorar la calidad del muestreo en el proceso de entrenamiento adversario.

```
1 import os, numpy as np, tensorflow as tf
2 from tensorflow import keras
3 from tensorflow.keras import layers, Model
4 import matplotlib.pyplot as plt
5 from sklearn.mixture import GaussianMixture
6
7 # =====
8 # CONFIGURACIÓN GENERAL
9 # =====
10
11 data_dir = r"PATH_TO_PROCESSED_COLOR_DATASET"
12 out_dir = r"PATH_TO_OUTPUT_DIRECTORY"
13 os.makedirs(out_dir, exist_ok=True)
14 os.makedirs(os.path.join(out_dir, 'images'), exist_ok=True)
15 os.makedirs(os.path.join(out_dir, 'models'), exist_ok=True)
16
17 img_size = 64
18 latent_dim = 100
19 batch_size = 32
20 epochs_autoencoder = 30
21 epochs_gan = 100
22 save_every = 5
23
24 # =====
25 # FUNCIÓN PARA ELEGIR LA MEJOR GMM
26 # =====
27
28 def bestGMM(X):
29     X = X.astype(np.float64)      # Asegurar precisión en 64 bits
```

```

30
31     Kmin = 1
32     bicmin = float('inf')
33     aicmin = float('inf')
34     bgmm = None
35
36     for K in range(1, 6):
37         gmm = GaussianMixture(
38             n_components=K,
39             covariance_type="diag", # Mucho más estable en 100D
40             reg_covar=1e-3,          # Anti-singularidad
41             n_init=5
42         ).fit(X)
43
44         bic = gmm.bic(X)
45         aic = gmm.aic(X)
46
47         if bic < bicmin:
48             Kmin = K
49             bicmin = bic
50             aicmin = aic
51             bgmm = gmm
52
53     print(f" GMM → K={Kmin}, BIC={bicmin:.2f}, AIC={aicmin:.2f}")
54     return bgmm
55
56
57 # =====
58 # CARGA DE DATASET
59 # =====
60
61 train_ds = tf.keras.preprocessing.image_dataset_from_directory(
62     data_dir,
63     image_size=(img_size, img_size),
64     batch_size=batch_size,
65     shuffle=True,
66     seed=42
67 )
68
69 def to_tanh(x,y): return (tf.cast(x,tf.float32)/127.5)-1., y
70 train_ds = train_ds.map(to_tanh).unbatch()
71 X_all = np.array([x.numpy() for x,_ in train_ds])
72
73 print("Dataset:", X_all.shape, f"rango [{X_all.min():.2f},{X_all.max():.2f}]")
74
75 # =====
76 # 1 GENERADOR PREENTRENADO COMO AUTOENCODER
77 # =====
78
79 def build_autoencoder(latent_dim, img_shape):
80     # Encoder
81     inp = layers.Input(shape=img_shape)
82     x = layers.Conv2D(64, 4, 2, 'same', activation='relu')(inp)
83     x = layers.Conv2D(128, 4, 2, 'same', activation='relu')(x)
84     x = layers.Conv2D(256, 4, 2, 'same', activation='relu')(x)
85     x = layers.Flatten()(x)
86     z = layers.Dense(latent_dim, name="latent")(x)
87
88     # Decoder
89     d = layers.Dense(8*8*256, activation='relu')(z)
90     d = layers.Reshape((8,8,256))(d)

```

```

91     for f in [128, 64, 32]:
92         d = layers.Conv2DTranspose(f, 4, 2, 'same', activation='relu')(d)
93     out = layers.Conv2D(3, 3, padding='same', activation='tanh')(d)
94
95     model = Model(inp, out, name="autoencoder")
96     return model
97
98 autoenc = build_autoencoder(latent_dim, (img_size,img_size,3))
99 autoenc.compile(optimizer=keras.optimizers.Adam(1e-3), loss='mae')
100
101 autoenc.summary()
102 autoenc.fit(X_all, X_all, epochs=epochs_autoencoder, batch_size=batch_size)
103 autoenc.save(os.path.join(out_dir, 'autoencoder_pretrained.keras'))
104
105 # =====
106 # EXTRAER ENCODER DEL AUTOENCODER
107 # =====
108
109 encoder_output = autoenc.get_layer("latent").output
110 encoder = Model(autoenc.input, encoder_output, name="encoder")
111
112 print(" Encoder extraido:", encoder)
113
114 # Calcular Z para todo el dataset
115 Z_all = encoder.predict(X_all, batch_size=64, verbose=1)
116 print("Latent space shape:", Z_all.shape)
117
118 # =====
119 # AJUSTAR GMM AL ESPACIO LATENTE
120 # =====
121
122 gmm = bestGMM(Z_all)
123 print(" GMM entrenado correctamente.")
124
125 # =====
126 # EXTRAER DECODER → GENERADOR PREENTRENADO
127 # =====
128
129 latent_input = keras.Input(shape=(latent_dim,))
130 x = latent_input
131 decoder_layers = autoenc.layers[-:]      # Decoder completo
132 for layer in decoder_layers:
133     x = layer(x)
134
135 generator = Model(latent_input, x, name="generator_pretrained")
136 print(" Generador preentrenado extraido:", len(decoder_layers), "capas")
137
138 # =====
139 # 2 CARGAR DISCRIMINADOR PREENTRENADO (PD)
140 # =====
141
142 disc_path = os.path.join(out_dir, "models", "discriminator_resnet_trained.h5")
143 discriminator = keras.models.load_model(disc_path)
144 discriminator.trainable = False
145
146 print(" Discriminador preentrenado cargado.")
147
148 # =====
149 # 3 DEFINIR GAN
150 # =====

```

```

152 z_input = keras.Input(shape=(latent_dim,))
153 fake_img = generator(z_input)
154 validity = discriminator(fake_img)
155
156 gan = Model(z_input, validity)
157 gan.compile(optimizer=keras.optimizers.Adam(2e-4, 0.5), loss="binary_crossentropy")
158
159 # =====
160 # UTILIDAD: GUARDAR IMÁGENES GENERADAS
161 # =====
162
163 def save_samples(epoch):
164     noise = gmm.sample(16)[0]           # GMM sampling
165     gen_imgs = generator.predict(noise, verbose=0)
166     gen_imgs = (gen_imgs+1)/2
167
168     plt.figure(figsize=(6,6))
169     for i in range(16):
170         plt.subplot(4,4,i+1)
171         plt.imshow(gen_imgs[i])
172         plt.axis('off')
173     plt.tight_layout()
174     plt.savefig(os.path.join(out_dir, 'images', f"epoch_{epoch:03d}.png"))
175     plt.close()
176
177 # =====
178 # 4 ENTRENAMIENTO GAN
179 # =====
180
181 half_batch = batch_size // 2
182
183 for epoch in range(1, epochs_gan+1):
184
185     # === Entrenar D ===
186     idx = np.random.randint(0, X_all.shape[0], half_batch)
187     real_imgs = X_all[idx]
188     y_real = np.ones((half_batch,1)) - np.random.uniform(0,0.1,(half_batch,1))
189
190     noise = gmm.sample(half_batch)[0]
191     fake_imgs = generator.predict(noise, verbose=0)
192     y_fake = np.random.uniform(0,0.1,(half_batch,1))
193
194     d_loss_real = discriminator.train_on_batch(real_imgs, y_real)
195     d_loss_fake = discriminator.train_on_batch(fake_imgs, y_fake)
196     d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)
197
198     # === Entrenar G ===
199     noise = gmm.sample(batch_size)[0]
200     y_gen = np.ones((batch_size,1))
201     g_loss = gan.train_on_batch(noise, y_gen)
202
203     print(f"Epoch {epoch}/{epochs_gan} - d_loss={d_loss[0]:.4f} acc={d_loss[1]:.3f}\n"
204          f"g_loss={g_loss:.4f}")
205
206     if epoch % save_every == 0:
207         save_samples(epoch)
208
209 print(" Entrenamiento GAN finalizado.")

```

Código A.1: Script de implementación del cGAN con muestreo GMM

ANEXO B

CATÁLOGO DE CARACTERÍSTICAS, ARQUITECTURAS E HIPERPARÁMETROS

Este anexo consolida la información técnica relativa a los **14 atributos seguros** de las URLs utilizados, así como las configuraciones detalladas de los hiperparámetros y la estructura de las redes neuronales empleadas para asegurar la replicabilidad de los experimentos.

B.1. Catálogo de Características Tabulares de URLs

Las **14 características** que conforman el conjunto final de `features_safe` (Sección 3.3.1) fueron diseñadas para asegurar la integridad y evitar el sesgo de las *cheating features* en los modelos de clasificación tabular (RF y ANN).

Tabla B.1: Catálogo de 14 Características Tabulares Seguras utilizadas en la Clasificación

ID Característica	Tipo de Dato	Descripción	Referencia en Código
url_length	Numérico (Int)	Longitud total de la URL en caracteres.	extract_url_features
num_dots	Numérico (Int)	Conteo total de puntos (‘.’) en la URL.	extract_url_features
num_digits	Numérico (Int)	Número total de dígitos numéricos en la URL.	extract_url_features
has_ip	Booleano (0/1)	Indicador de la presencia de una dirección Internet Protocol (IP) numérica en lugar de dominio.	extract_url_features
has_at_symbol	Booleano (0/1)	Indicador de la presencia del símbolo arroba (@).	extract_url_features
has_https	Booleano (0/1)	Indicador del uso del protocolo HTTPS.	extract_url_features
num_special_chars	Numérico (Int)	Conteo de caracteres especiales no alfanuméricos.	extract_url_features
has_suspicious_words	Booleano (0/1)	Presencia de keywords de fraude (e.g., <i>login</i> , <i>verify</i> , <i>bank</i>) en la URL.	extract_url_features
domain_length	Numérico (Int)	Longitud en caracteres del dominio base.	extract_domain_features
num_subdomains	Numérico (Int)	Número de subdominios presentes en la URL.	extract_domain_features
tld_length	Numérico (Int)	Longitud del TLD (e.g., 3 para .com).	extract_tld_features
title_length	Numérico (Int)	Longitud en caracteres del Título HyperText Markup Language (HTML).	extract_title_features
has_login_word_in_title	Booleano (0/1)	Presencia de la palabra 'login' en el Título HTML.	extract_title_features
has_secure_word_in_title	Booleano (0/1)	Presencia de la palabra 'secure' en el Título HTML.	extract_title_features

B.2. Arquitecturas e Hiperparámetros de Modelos

B.2.1. Hiperparámetros Comunes y de RF

Tabla B.2: Hiperparámetros Comunes y Modelo Random Forest (RF)

Parámetro	Valor	Justificación / Modelo(s) Aplicable(s)
Tamaño de Lote (Batch Size)	32	Estabilidad / ANN, ResNet, GAN
Optimizador (NN)	Adam	Eficiencia y adaptación automática de η .
Tasa de Aprendizaje (LR) (Clasificación)	1×10^{-3}	ANN, ResNet (<i>fine-tuning</i> más lento)
Tasa de Aprendizaje (LR) (GAN)	2×10^{-4}	Estabilidad del entrenamiento adversario.
Épocas Máximas (Clasificación)	50	Con <i>Early Stopping</i> (<i>patience</i> =10).
Épocas Máximas (GAN)	200	Mayor complejidad, más lento de converger.
Hiperparámetros Específicos de Random Forest (RF)		
<i>n_estimators</i>	100	Número de árboles en el bosque (Sección 4.1).
<i>random_state</i>	42	Semilla para reproducibilidad de resultados.
<i>cv</i> (RF Validación)	5	Folds para la Validación Cruzada Estratificada.

B.2.2. Arquitectura de Red Neuronal Artificial (ANN)

El Perceptrón Multicapa (MLP) utilizado para el *baseline* tabular se compone de tres capas densas, como se describe en la Sección 4.1.

Tabla B.3: Arquitectura del Perceptrón Multicapa (ANN)

Capa	Salida	Activación	Regularización
Input/Dense 1	64	ReLU	-
Dropout	64	-	$p = 0,2$
Dense 2	32	ReLU	-
Output Layer	1	Sigmoid	Clasificación binaria

B.2.3. Arquitectura del Generador (G) en la Estrategia PD/PG

El Generador se obtiene del Decodificador de un Autoencoder preentrenado. Utiliza capas *Conv2DTranspose* para realizar el *upsampling* de la imagen, con la arquitectura base de 4 bloques (3 de transposición y la capa final de salida Tanh).

Tabla B.4: Estructura del Generador (G) Preentrenado (PG) de la cGAN

Bloque	Capa	Filtros	Stride	Salida (Tamaño)
Input	Espacio Latente (Dense)	-	-	$8 \times 8 \times 256$
Decoder 1	Conv2DTranspose	128	2	$16 \times 16 \times 128$
Decoder 2	Conv2DTranspose	64	2	$32 \times 32 \times 64$
Decoder 3	Conv2DTranspose	32	2	$64 \times 64 \times 32$
Output	Conv2D (Tanh)	3	1	$64 \times 64 \times 3$

B.2.4. Arquitectura del Discriminador (D) en Estrategia PD

El Discriminador utilizado en las estrategias **PD** (*Preentrenado*) es una ResNet-50 con *Transfer Learning*, configurada de la siguiente manera (Sección 4.3.4):

- **Modelo Base:** ResNet-50, cargado con pesos de **ImageNet**.
- **Estrategia:** Las capas convolucionales del modelo base están **congeladas** (`trainable=False`).
- **Cabeza de Clasificación:** Se añade una capa *Dense* (512 unidades) con *ReLU* y una capa de salida *Dense* (1 unidad) con *Sigmoid* para la clasificación binaria final (*Phishing* vs. Legítimo).

ANEXO C

TABLAS DE RESULTADOS EXTENDIDAS Y GRÁFICAS DE ENTRENAMIENTO

Este anexo contiene los resultados de evaluación completos y las curvas de entrenamiento, complementando los datos resumidos en el Capítulo 5.

C.1. Resultados de Clasificación Detallados y Matriz de Confusión

Tabla C.1: Métricas de Evaluación Detalladas (Clasificadores Tabulares y Visuales)

Modelo	Accuracy	Precision	Recall	F1-Score	AUC-ROC
RF (Tabular)	0.966	0.963	0.970	0.967	0.992
ANN (Tabular)	0.958	0.950	0.965	0.957	0.989
CNN (Grayscale)	0.962	0.951	0.960	0.955	0.990
ResNet-50 (Visual)	0.984	0.978	0.989	0.983	0.997

Tabla C.2: Matriz de Confusión del Modelo ResNet-50 (Mejor Rendimiento)

Clasificación	Pred. Legítimo (1)	Pred. Phishing (0)	Total Real
Real: Legítimo (1)	1574 (Verdaderos Positivos)	21 (Falsos Negativos)	1595
Real: Phishing (0)	28 (Falsos Positivos)	1567 (Verdaderos Negativos)	1595

C.2. Gráficos de Convergencia del Entrenamiento

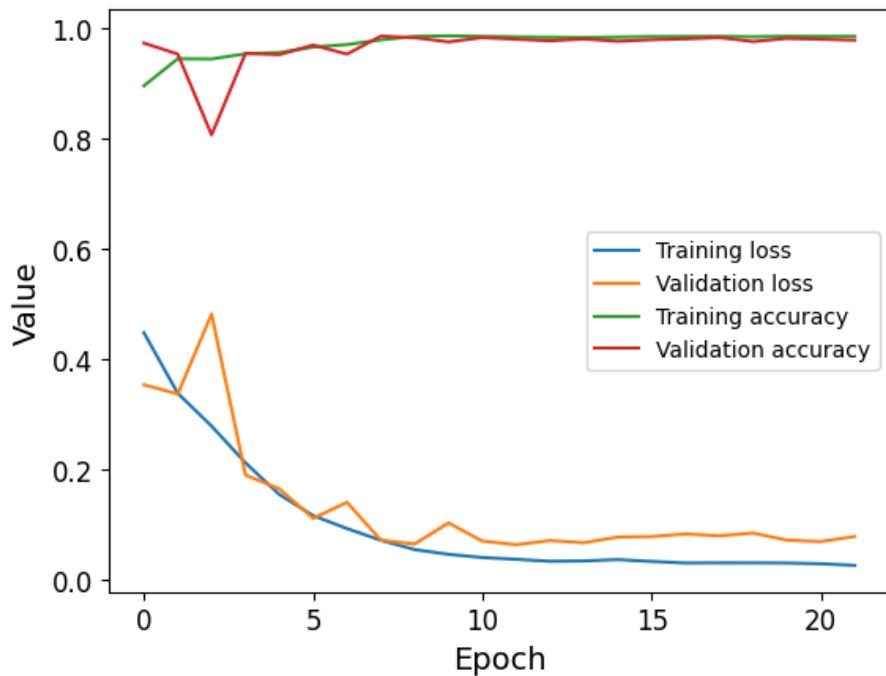


Figura C.1: Curvas de Pérdida (Loss) y Precisión (Accuracy) del modelo ANN durante el entrenamiento, mostrando la convergencia y el punto de *Early Stopping* (Fuente: Elaboración propia).

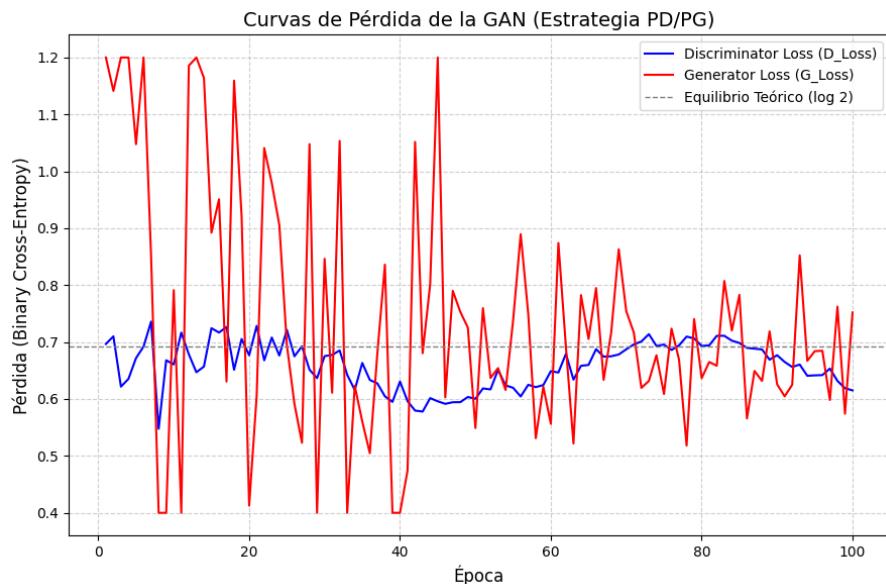


Figura C.2: Curvas de Pérdida del Generador (g_loss) y Discriminador (d_loss) de la cGAN (PD/PG) durante el entrenamiento, mostrando el equilibrio adversario (Fuente: Elaboración propia).

C.2.1. Análisis Detallado de la Convergencia Adversaria (cGAN PD/PG)

La Figura C.2 representa la evolución de las funciones de pérdida del Generador (G_Loss) y del Discriminador (D_Loss) para la configuración óptima PD/PG (Discriminador y Generador Preentrenados), a lo largo de 100 épocas. El análisis de esta curva es crucial para validar la estabilidad de la estrategia de *Transfer Learning* propuesta.

C.2.1.1. Estabilidad y Equilibrio de Nash

El factor más importante en la gráfica es el comportamiento de la pérdida del Discriminador (D_Loss, azul):

- **Estabilización Temprana:** La D_Loss no cae a cero y se estabiliza consistentemente alrededor del **Equilibrio Teórico (0,69)** desde la Época 10.
- **Conclusión:** Esta estabilidad demuestra que el Discriminador (ResNet-50 preentrenado) actúa como un **crítico robusto pero no dominante**. El valor $\sim 0,69$ implica que el Discriminador está **constantemente confundido**, acertando solo cerca del 50 % de las veces. Esto valida la eficacia de la estrategia PD (Discriminador Preentrenado) para evitar la pérdida de gradientes y asegurar el **equilibrio de Nash**.

C.2.1.2. Comportamiento del Generador y Exploración del Espacio Latente

La pérdida del Generador (G_Loss, rojo) muestra una alta oscilación:

- **Volatilidad:** La G_Loss presenta picos altos (hasta $\sim 1,2$) y valles bajos (hasta $\sim 0,4$) alrededor del D_Loss promedio.
- **Interpretación:** Esta volatilidad es intrínseca a la fase de entrenamiento adversario y es un indicador de que el Generador está **explorando agresivamente** el espacio de soluciones para engañar al Discriminador. La oscilación confirma que el Generador está en un ciclo de **aprendizaje activo** y no ha caído en una solución de bajo esfuerzo.
- **Ausencia de Colapso:** El patrón de oscilación constante, sin caer a valores mínimos extremos ($G_Loss \approx 0$) o sin que D_Loss caiga a 0, confirma que se evitó el **colapso de modo** (Generación de muestras repetitivas) y el **colapso de entrenamiento** (dominio total del Discriminador).

C.3. Análisis Detallado de Varianza (PCA)

El análisis de componentes principales (PCA), discutido en la Sección 5.1, valida que las 14 características tabulares son estructuralmente robustas.

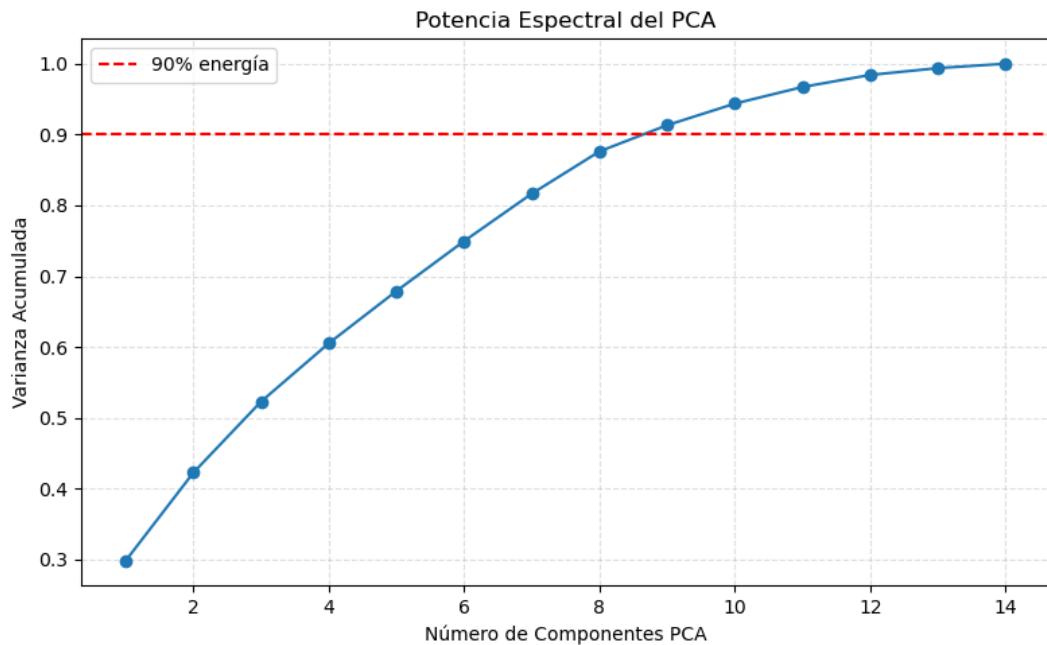


Figura C.3: Varianza acumulada de las componentes principales. Se retiene el 90 % de la varianza con 9 componentes, confirmando la dimensionalidad intrínseca de los datos (Fuente: Elaboración propia).

ANEXO D

MUESTRAS VISUALES DE FAVICONS REALES Y SINTÉTICOS

Este anexo proporciona una inspección visual de las imágenes utilizadas para el entrenamiento y las muestras sintéticas generadas por las arquitecturas GAN, apoyando la discusión cualitativa del Capítulo 5.

D.1. Muestras del Dataset Balanceado

La Figura D.1 presenta una muestra aleatoria del dataset balanceado (1595 legítimos / 1595 *phishing*), utilizado para el entrenamiento de los modelos visuales (ResNet-50 y GAN).

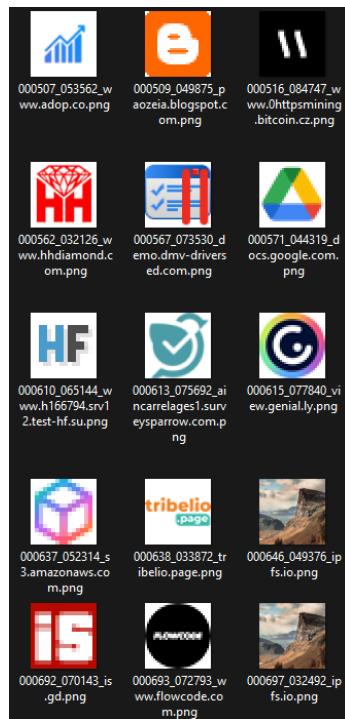


Figura D.1: Cuadrícula de Favicons (**64 × 64**) del dataset balanceado, mostrando la diversidad de logotipos y la baja calidad de algunas muestras (Fuente: Elaboración propia).

D.2. Comparación Visual de la Generación Adversarial

Se presentan las muestras de las estrategias GAN clave, ilustrando la conclusión de que **PD / PG** ofrece la mejor fidelidad perceptiva, mientras que **PD / MG** muestra un colapso a la media.

D.2.1. Mejor Estrategia Perceptual: PD/PG (FID: 75.3)

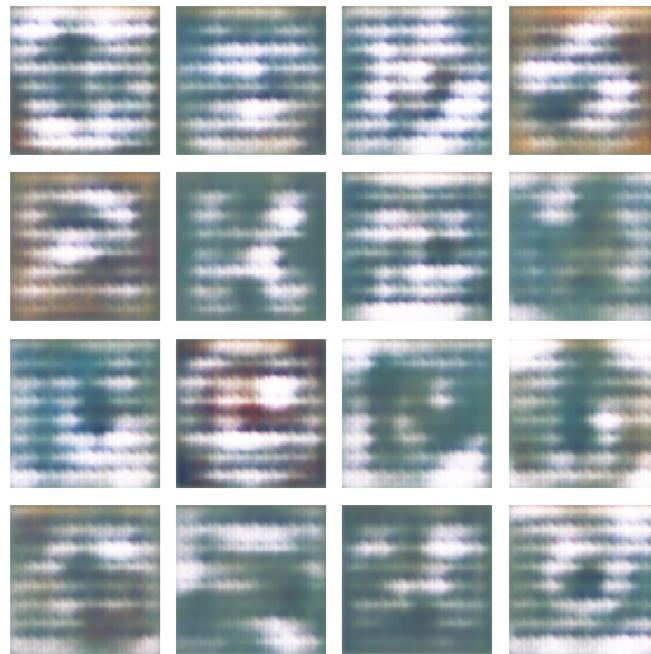


Figura D.2: Muestras generadas por la cGAN con inicialización PD/PG. Exhibe la mejor fidelidad perceptiva y diversidad visual (Fuente: Elaboración propia).

D.2.2. Mejor Estrategia Estadística: PD/MG (FID: 68.5)

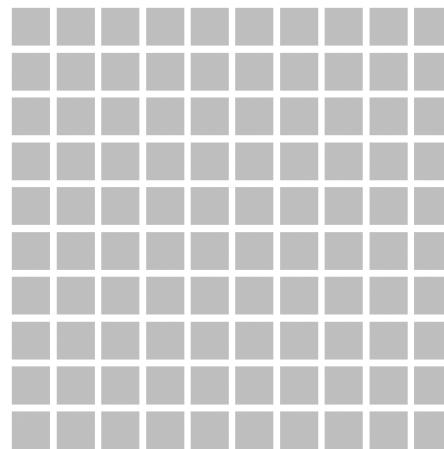


Figura D.3: Muestras generadas por la cGAN con inicialización PD/MG. El mejor FID a costa del colapso visual a la media (gris uniforme) (Fuente: Elaboración propia).

D.2.3. Prueba de Concepto del Muestreo GMM



Figura D.4: Favicon sintéticos generados utilizando muestreo GMM en el espacio latente. Muestra la reproducción de clusters dominantes (Google Drive, is.gd), confirmando la capacidad del GMM para identificar la distribución (Fuente: Elaboración propia).