Universitat Oberta de Catalunya

uoc.edu

Deep Learning y ciberseguridad.

Malicious URL detection mediante técnicas de Deep Learning

Alumno: Óscar Francés Luesma

Plan de Estudios: Máster Universitario en Ciberseguridad y Privacidad

Trabajo de fin de máster. M1.888 - Análisis de datos

Nombre Consultor/a. Enric Hernández Jiménez

Nombre Profesor/a responsable de la asignatura. Joan Caparrós Ramírez

Fecha de entrega: 28/12/2021

Esta obra está sujeta a una licencia de Reconocimiento-NoComercial-SinObraDerivada 3.0 España de Creative Commons

FICHA DEL TRABAJO FINAL

Título del trabajo:	*Deep Learning y ciberseguridad.* *Malicious URL detection mediante técnicas de Deep Learning*
Nombre del autor:	*Óscar Francés Luesma*
Nombre del consultor/a:	*Enric Hernández Jiménez*
Nombre del PRA:	*Joan Caparrós Ramírez*
Fecha de entrega (mm/aaaa):	02/10/2021
Titulación:	*Máster Universitario en Ciberseguridad y Privacidad*
Área del Trabajo Final:	*Deep Learning y ciberseguridad*
Idioma del trabajo:	*Castellano*
Palabras clave	*Deep Learning, Malicious URL*

Resumen del Trabajo (máximo 250 palabras): *Con la finalidad, contexto de aplicación, metodología, resultados i conclusiones del trabajo.*

En este trabajo se muestra la utilidad de las técnicas de Deep Learning para la detección de *URL* maliciosas que es una de las técnicas más utilizadas en los llamados ataques de ingeniería social.

Los ataques de ingeniería social es la práctica de obtener información confidencial a través de la manipulación de usuarios legítimos. Consiste en enviar mediante un medio de comunicación legítimo (por ejemplo, correo electrónico) un enlace a una *URL* de una página web que dispone de código malicioso. El objetivo es que el usuario pulse sobre el enlace de la *URL* para provocar el acceso a dicha página web maliciosa. En ese momento, el código malicioso se ejecuta con los permisos del usuario y puede realizar un ataque sobre el sistema de información en base a los permisos de los que dispone el usuario. Cuantos más privilegios tenga el usuario mayor efecto devastador tendrá el ataque sobre el sistema de información.

Las redes neuronales (Deep Learning) que se proponen en este proyecto detectarán con diversa precisión las *URL* potencialmente maliciosas. Para este propósito, se experimenta con distintos tipos de redes neuronales y se muestra cuáles de ellas son más eficientes para este tipo de ataque.

Para la consecución de estos objetivos es requisito imprescendible obtener un conjunto de datos de alta calidad y la aplicación previa de técnicas de Machine Learning para preparar dichos datos. También se requerirá de técnicas de entrenamiento y validación del modelo.

Abstract (in English, 250 words or less):

This paper shows the usefulness of Deep Learning techniques for the detection of URL malicious, which is one of the most used techniques in so-called social engineering attacks.

Social engineering attacks are the practice of obtaining sensitive information through the manipulation of legitimate users. It consists of sending through a legitimate means of communication (for example, email) a link to a URL of a web page that has malicious code. The objective is for the user to click on the URL link to cause access to the malicious web page. At that time, the malicious code is executed with the user's permissions and can carry out an attack on the information system based on the permissions available to the user. The more privileges the user has, the greater the devastating effect the attack will have on the information system.

The neural networks (Deep Learning) proposed in this project will detect potentially malicious URL with greater or lesser precision. To do this, we experiment with different types of neural networks and show which of them are more efficient for this type of attack.

To achieve these objectives, it is an essential requirement to obtain a high-quality data set and the prior application of Machine Learning techniques to prepare such data. Training techniques and model validation will also be required

Índice

Contenido

1 INTRODUCCIÓN **1**

1.1 Contexto y justificación del Trabajo 1

1.2 Objetivos del Trabajo 1

1.3 Enfoque y método seguido 3

1.4 Planificación del Trabajo 3

1.5 Planificación temporal detallada de tareas y sus dependencias 5

1.6 Análisis de riesgos 6

1.7 Breve sumario de productos obtenidos 7

1.8 Recursos necesarios y presupuesto del proyecto 8

1.9 Breve descripción de los otros capítulos de la memoria 9

2 FASE DE INVESTIGACIÓN Y PRIMERA *DNN* **10**

2.1 Estado del arte 10

2.1.1 Representación de características. *12*

2.1.2 Algoritmos de aprendizaje. *13*

2.1.3 Detección de URL maliciosas como un servicio. *17*

2.1.4 Problemas abiertos. *19*

2.1.5 Áreas relacionadas. *19*

2.2 Decisiones de diseño TFM. 20

2.3 Selección del conjunto de datos. 20

2.3.1 Datos desbalanceados. *21*

2.3.2 Selección y transformación de las características. *23*

2.3.3 Selección del algoritmo de aprendizaje. *25*

2.4 Primera red neuronal (*DNN*). 25

2.4.1 Creación del modelo. *25*

2.4.2 Detalles de diseño de la arquitectura de la DNN. *26*

2.4.3 Detalles de diseño de la compilación de la DNN. 27

2.4.4 Pasos de entrenamiento de la DNN mediante el método del Gradiente Descendente Estocástico. 27

2.4.5 Entrenamiento de la red. 28

2.4.6 Evaluar el modelo y calcular predicciones. 28

2.4.7 k-Fold Cross Validation. 29

2.4.8 Hiperparámetros. 29

2.4.9 Análisis de la función de pérdida y sobreentrenamiento ('loss function' y 'overfitting'). 30

3 FASE DE DESARROLLO DE *RNN* Y *TRANSFORMER* **34**

3.1 REDES LONG SHORT-TERM MEMORY (LSTM) 34

3.1.1 Arquitectura LSTM 35

3.1.2 Flujo de información y formulación 36

3.1.3 LSTM en nuestro escenario 38

3.1.4 Limitaciones de RNN, LSTM y GRU 48

3.2 REDES NEURONALES TRANSFORMER. *SELF-ATTENTION*. 50

3.2.1 Arquitectura Transformer 50

3.2.2 Flujo de información y formulación 52

3.2.3 Atención en nuestro escenario. Clasificación 54

4 MUESTRA DE RESULTADOS **63**

5 CONCLUSIONES **68**

6 GLOSARIO **70**

7 BIBLIOGRAFÍA **73**

8 ANEXOS **77**

Lista de figuras

Fig. 1. Desglose de tareas según planificación [29]. 4

Fig. 2. Diagrama de Gantt del proyecto. 5

Fig. 3. Ejemplo de una URL - “Uniform Resource Locator". 10

Fig. 4. Marco de proceso general para la detección de URL maliciosas mediante el aprendizaje automático. 12

Fig. 5. En rojo las páginas web maliciosas y en verde las benignas (2,27% vs 97,73%). 21

Fig. 6. De más oscuro a más claro el número de peticiones por host. 21

Fig. 7. Longitud del código JavaScript en páginas maliciosas (rojo) vs páginas benignas (verde). 22

Fig. 8. Respuesta completa e incompleta a las URL maliciosas(rojo) vs las URL benignas (verde) [30]. 23

Fig. 9. Visualización de los primeros registros del conjunto de datos después del paso 2. 24

Fig. 10. Correlación de Pearson en variables (después del paso 2). 24

Fig. 11. Visualización de los primeros registros del conjunto de datos después del paso 3. 24

Fig. 12. Matriz de confusión y precisión sobre los datos de test. 28

Fig. 13. Sobreentrenamiento (overfitting). 30

Fig. 14. ‘sentiment_polarity’ en el conjunto de datos URL maliciosas. 32

Fig. 15. ’profanity-check’ en el conjunto de datos URL maliciosas. 32

Fig. 16. RNN estándar en una sola capa [63]. 35

Fig. 17. Arquitectura simplificada en una red LSTM [63]. 35

Fig. 18. Cell State, flujo de información a través de la red LSTM [63]. 36

Fig. 19. Arquitectura de la compuerta forget gate [63]. 36

Fig. 20. Arquitectura de la compuerta input gate [63]. 37

Fig. 21. Actualización del estado [63]. 37

Fig. 22. Output gate basado en el estado de la celda [63]. 38

Fig. 23. Arquitectura simplificada Gated Recurrent Unit (GRU). 41

Fig. 24. Función tanh y softmax. 46

Fig. 25. Función de pérdida. 47

Fig. 26. Secuencialidad de las redes neuronales recurrentes (RNN) [21]. 48

Fig. 27. Codificador de atención neuronal jerárquico [21]. 49

Fig. 28. Ejemplo de Transformer como caja negra. Ejemplo de traducción [6]. 50

Fig. 29. Capas de los codificadores, decodificadores, incluyendo las capas embedded y a la entrada y capa de salida [6]. 50

Fig. 30. Estructura interna del encoder y decoder [6]. 51

Fig. 31. Ejemplo de secuencia mostrando que el mecanismo de atención permite que dos frases casi iguales sepan por el contexto que el término “it” hace referencia a “cat” o “milk”. 51

Fig. 32. Flujo de entrada y salida en el proceso de entrenamiento (izquierda) respecto al proceso de predicción (derecha). 52

Fig. 33. Revisión de la arquitectura Transformer para un problema de clasificación como URL Malicious. 54

Fig. 34. Pre-entrenamiento y ajuste fino en un problema de clasificación [20]. 55

Fig. 35. Modelos de BERT. L (capas), H (dimensiones capa oculta), A (head multi-detection) [9]. 56

Fig. 36. La capa de embedding de BERT facilita un vector de 768 dimensiones por cada palabra de un diccionario total de 30.000 palabras [24]. 57

Fig. 37. Diferencia entre pre-training y Fine tuning [8]. 57

Fig. 38. Modelo teacher-student para transferencia de aprendizaje entre dos redes neuronales [22]. 58

Fig. 39. BERT y DistilBERT [4]. 59

Fig. 40. Red neuronal antes y después de un proceso de poda (pruning) [47]. 59

Fig. 41. Factorización de matrices por vocabulario / parámetros [47]. 60

Fig. 42. Modelo teacher-student para transferencia de aprendizaje en BERT [4]. 60

Fig. 43. Pesos compartidos entre capas [47]. 61

Fig. 44. Ejemplo de cuantización de float a int8 [3]. 61

Fig. 45. Procesos completo en BERT para diferentes tareas. 62

Lista de tablas

Tabla 1. Lista de hitos del proyecto. 6

Tabla 2. Lista de recursos requeridos en el proyecto. 8

Tabla 3. Propiedades de diferentes representaciones de características para la detección de URL maliciosas. 12

1 Introducción

1.1 Contexto y justificación del Trabajo

La ingeniería social se basa en un principio muy básico: "el usuario es el eslabón más débil". A partir de esta idea, busca explotarlo apelando a sus motivaciones más personales, con el objetivo de conseguir que el usuario revele cierta información o "permita" ceder el control de su equipo. Nuestra mejor defensa es no dejarnos engañar y conocer cómo funcionan este tipo de fraudes.

Según el *INCIBE* 93% de las brechas de seguridad son debido a un ataque de ingeniería social *[38]*. Por ello, mediante la Oficina de Seguridad del Internauta (*OSI*) el instituto proporciona recomendaciones a los usuarios para evitar ser víctimas de engaños y, como desconfiar de los chantajes o extorsiones.

Adicionalmente, el aumento de técnicas de *Deep Learning [40]* en los últimos años, sobre todo en temas relacionados con las redes neuronales recurrentes (*RNN*) y más recientemente los *Transformer [59]* aplicados con éxito al proceso natural del lenguaje (*NLP*) nos plantea la pregunta de si dichas técnicas pudieran ser aplicables a la ciberseguridad y más concretamente a la detección de ataques de ingeniería social.

Actualmente, los ataques de ingeniería social son tratados mediante planes de comunicación hacia la ciudadanía o empresas. En estos planes de comunicación, por ejemplo, *INCIBE*, publica boletines informativos *[38]* indicando las acciones que deben realizar los potenciales atacados para evitar las brechas de seguridad. Las acciones recomendadas son:

- Identificación de la ingeniería social. Se dan una serie de pautas para que el usuario detecte que está siendo objeto de un ataque social.
- Recomendaciones sobre remitentes desconocidos, desconfianza sobre chantajes o extorsiones, suplantación de un técnico de un servicio, desconfianza ante petición de información bancaria, navegación siempre por páginas seguras (https) y no compartir información personal mediante correo electrónico, redes sociales o servicios de mensajería instantánea.

Este trabajo pretende ofrecer un mecanismo mediante una red neuronal con un modelo ya entrenado que al pasarle un enlace *URL* pueda saber con un grado de predicción elevado si se trata de un potencial enlace *URL* malicioso o no.

1.2 Objetivos del Trabajo

Los principales objetivos de este trabajo de fin de máster son los siguientes:

- Objetivos de investigación:

- Investigación sobre los posibles conjuntos de datos de entrada existentes en el mundo sobre la temática de *URL Malicious* y la selección de uno de ellos de referencia para tomarlo como entrada del proyecto.

- Investigación sobre las diferentes técnicas de *Deep Learning [40]* profundizando en *Artificial Neural Network (ANN), Recurrent Neural Network (RNN)* y *Transformer [59]*.

- Investigación sobre la plataforma *Keras* y *TensorFlow* 2.0.

- Investigación sobre Google BERT que dispone del motor basado en *Transformer* y es el sistema basado en inteligencia artificial (IA) que usa Google para entender mejor a los usuarios a la hora de realizar búsquedas.

- Objetivos de implantación:

  - Instalación del entorno que permita disponer de *Python*, *Keras* y *TensorFlow*.

  - Disponer de un Python notebook que muestre las bondades del conjunto de datos seleccionado *[64]*.

  - Disponer de un Python notebook con el desarrollo del modelo realizado sobre la *Artificial/Deep Neural Network (ANN, DNN)*.

  - Disponer de un Python notebook con la implementación del modelo sobre la *Artificial/Deep Neural Network (ANN,DNN).*

  - Disponer de un Python notebook con la implementación del modelo sobre la *Recurrent Neural Network (RNN)*.

  - Disponer de un Python notebook con la implementación del modelo sobre Google BERT (*Transformer*).

  - Realizar una tabla final en el que se pueda ver de una forma simple las diferencias entre los modelos empleados, así como de las conclusiones obtenidas.

Objetivos de entrega:

- Desarrollar las entregas parciales y enviarlas en tiempo y forma.

- Desarrollar la memoria final del trabajo.

- Preparar un vídeo presentación.

1.3 Enfoque y método seguido

El enfoque de este proyecto consistirá en investigar las técnicas de *Deep Learning [40]* necesarias, así como la realización de Python Notebooks, que irán mostrando los *KPIs* obtenidos sobre el rendimiento de cada modelo usando exactamente el mismo conjunto de datos de entrada *[58]* lo que nos permitirá finalmente compararlos y poder recomendar las técnicas óptimas para la detección de *URL Malicious*.

El proyecto se divide en dos partes:

- Primera parte, en la que vamos a montar el entorno necesario para investigar sobre las técnicas planteadas (*Python, Keras* y *TensorFlow* 2.0), también realizaremos un notebook sobre el *dataset* seleccionado *[58]* con el fin de mostrar la información que nos proporciona y finalmente realizaremos un notebook sobre la primera red neuronal *Deep Neural Network* (*DNN*) mostrando los primeros datos de rendimiento (PEC 2).

- Segunda parte, en la que nos centraremos en dos notebooks para cada una de las redes neuronales restantes: *Recurrent Neural Network* (*RNN*) y *Google BERT [59]* (*Transformer*) mostrando los datos de rendimiento (PEC 2).

Debido a la limitación temporal establecida en la planificación, la preparación teórica sobre algunos conceptos novedosos (*Transformer*) deben ser realizados durante la primera parte, para que al llegar a la segunda parte puedan acometerse los objetivos previstos sin impedimento por falta de conocimiento por dichas técnicas.

En el hito de la presentación de la PEC2, una vez realizado el montaje, analizado los datos y con la primera red neuronal montada (*DNN*) se valorará si la consecución de todos los objetivos es posible, ajustando la planificación, de forma que, si bien algún objetivo se considera inviable ponerlo en práctica, siempre queden los objetivos de aprendizaje e investigación.

1.4 Planificación del Trabajo

A continuación, se describen las tareas a realizar durante la elaboración del proyecto, así como la estimación de cada tarea a realizar *[64]*.

URL Malicious Plan

Tarea

Nombre	Fecha de inicio	Fecha de fin	Duración
Planificación (PEC 1)	15/9/21	30/9/21	12
Establecer problema a resolver y contexto	15/9/21	17/9/21	3
Definición de objetivos	20/9/21	21/9/21	2
Definir propuesta metodología	22/9/21	24/9/21	3
Elaborar cronograma de trabajo	27/9/21	27/9/21	0
Definición de tareas	27/9/21	29/9/21	3
Calcular tiempos para entregas	27/9/21	28/9/21	2
Identificar riesgos	29/9/21	30/9/21	2
Definir costes y recursos necesarios	29/9/21	30/9/21	2
Entrega del plan de trabajo	1/10/21	1/10/21	0
Entorno, selección Dataset y primera ANN (PEC 2)	4/10/21	26/10/21	17
Montaje del entorno	4/10/21	5/10/21	2
Selección del dataset	6/10/21	6/10/21	0
Exploración del dataset	6/10/21	6/10/21	1
Elaboración Notebook dataset	7/10/21	8/10/21	2
Elaboración primera ANN (Artificial Neuronal Network)	11/10/21	19/10/21	7
Elaboración Notebook ANN	20/10/21	21/10/21	2
Exploración hiperparámetros ANN	22/10/21	26/10/21	3
Selección de mejores resultados ANN	27/10/21	27/10/21	0
Implementación RNN y Transformer (PEC 3)	27/10/21	23/11/21	20
Elaboración RNN (Recurrent Neuronal Network)	27/10/21	29/10/21	3
Elaboración Notebook RNN	1/11/21	2/11/21	2
Exploración hiperparámetros RNN	3/11/21	4/11/21	2
Selección de mejores resultados RNN	5/11/21	5/11/21	0
Elaboración Transformer (Google BERT)	5/11/21	15/11/21	7
Elaboración Notebook Transformer	16/11/21	17/11/21	2
Exploración hiperparámetros Transformer	18/11/21	22/11/21	3
Selección de mejores resultados Transformer	23/11/21	23/11/21	0
Mostrar comparativa de KPIs entre diferentes métodos	23/11/21	23/11/21	1
Presentación	24/11/21	28/12/21	25
Presentación y defensa del TFM	24/11/21	24/11/21	0
Conclusiones sobre el trabajo realizado	24/11/21	29/11/21	4
Elaboración de la memoria final	30/11/21	16/12/21	13
Elaboración del vídeo presentación de la memoria	17/12/21	28/12/21	8

Fig. 1. Desglose de tareas según planificación [29].

1.5 Planificación temporal detallada de tareas y sus dependencias

A continuación, se indica una planificación temporal sobre las tareas y las dependencias entre ellas mediante un diagrama de Gantt *[64]*.

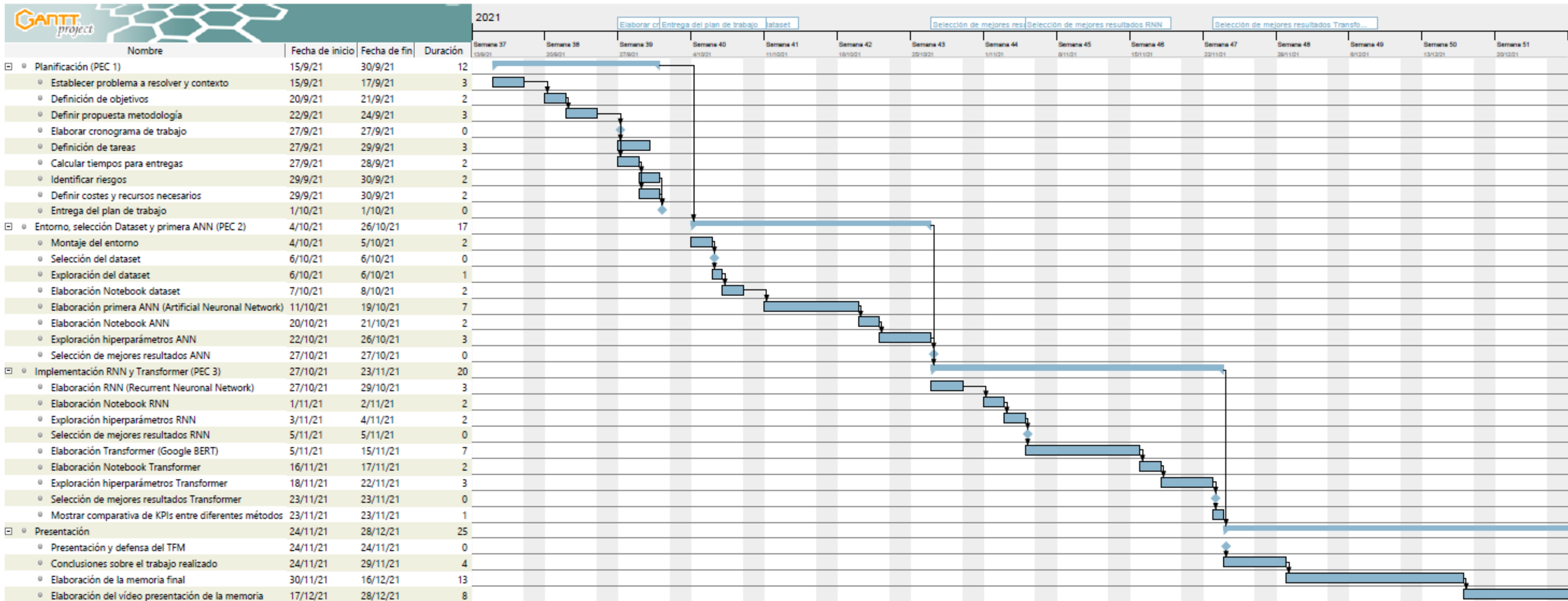


Fig. 2. Diagrama de Gantt del proyecto.

A continuación, se indica la lista de hitos que también pueden ser visualizados en el Gantt *[64]*:

Tabla 1. Lista de hitos del proyecto.

Hito	**Periodo**	**Fecha de entrega**
Elaborar cronograma de trabajo	Planificación (PEC 1)	27 de septiembre de 2021
Entrega del plan de trabajo	Planificación (PEC 1)	1 de octubre de 2021
Selección del dataset	Entorno, selección *Dataset* y primera *DNN* (PEC 2)	6 de octubre de 2021
Selección de mejores resultados *DNN*	Entorno, selección *Dataset* y primera DNN (PEC 2)	27 de octubre de 2021
Selección de mejores resultados RNN	Implementación RNN y *Transformer* (PEC 3)	5 de noviembre de 2021
Selección de mejores resultados Transformer	Implementación RNN y *Transformer* (PEC 3)	23 de noviembre de 2021
Presentación y defensa del TFM	Presentación (PEC 4)	29 de diciembre de 2021

Los recursos principales son las horas de elaboración de las tareas que las realizará un alumno, y se contará con un portátil de última generación en el que se instalarán las aplicaciones requeridas para la elaboración del análisis.

1.6 Análisis de riesgos

A continuación, se enumeran una serie de riesgos que pueden hacer que el proyecto fracase o que la planificación no pueda ajustarse a los tiempos marcados.

- *R1. Objetivo demasiado ambicioso.*

  Definición del riesgo:

  Existe el riesgo de no finalizar el trabajo en el tiempo planificado por tratarse de un objetivo de proyecto ambicioso para el periodo de elaboración del trabajo de fin

de máster sobre todo por la falta de conocimiento de la tecnología más novedosa como son los *Transformer* y por ende, *Google BERT [59]*.

Mitigación del riesgo:

Se ha comprado bibliografía sobre el tema, en concreto *[59]* adelantando su lectura en fases previas a la que está previsto su utilización.

- *R2. Problemas con las herramientas*.

  Definición del riesgo:

  Las herramientas a integrar pueden ser incompatibles entre versiones.

  Mitigación del riesgo:

  Es la primera tarea por realizar en la planificación para poder reaccionar ante cualquier problema con algo más de tiempo.

- *R3. Problemas con la capacidad del hardware*.

  Definición del riesgo:

  Algunas de las herramientas a utilizar requieren de gran capacidad de cómputo, así como el uso de *GPUs* o *FPUs* dedicados.

  Mitigación del riesgo:

  De momento se piensa en realizar los cálculos en local, pero si se encuentra problemas no se descarta el alquiler por un par de semanas de algún servicio de Google Cloud que permita diferir los cálculos a servidores externos con *GPUs* y *FPUs* dedicadas y que puedan acelerar el proceso de la construcción de modelos complejos.

1.7 Breve sumario de productos obtenidos

Este apartado se ampliará durante el proceso de investigación del proyecto se parte de algunos análisis realizados previamente y que se exponen a continuación.

Uno de los temas claves es la búsqueda de un conjunto de datos que sea suficientemente rico y preciso sobre la temática a tratar (*URL malicious*) y aunque sí que se han encontrado bastantes conjuntos de datos sobre la temática unos pocos cumplían los requisitos que buscábamos, tales como: que fueran públicos y usados por otros trabajos, que fueran suficientemente actualizados. Con estos criterios se encontró un conjunto de datos *[58]* proporcionado por el *National Center for Biotechnology Information* que dispone de amplia información sobre las *URL* detectadas y con una actualización bastante reciente (agosto de 2020).

El proyecto consistirá en 4 notebooks en Python que requieren de una instalación de Anaconda con *JupyterLab* donde ejecutaremos dichos Notebooks.

El primer notebook es una exploración de los datos entregados por el conjunto de datos *[58]* aplicando técnicas de *Machine Learning* y la librerías panda y *Scikit Learn* para lenguaje *Python*.

El segundo notebook es el desarrollo de un modelo *DNN* (*Deep Neural Network*) sobre los datos del conjunto de datos ya refinados en el paso anterior. Para este paso ya es necesario del uso de *Keras* y *TensorFlow* 2.0 para la construcción de la red neuronal.

El tercer notebook es el desarrollo de un modelo RNN (Recurrent Neural Network) sobre los datos del conjunto de datos. Para este paso es necesario el uso de *Keras* y *TensorFlow* 2.0 para la construcción de la red neuronal.

El cuarto notebook es el desarrollo de un modelo *Transformer [59]* sobre los datos del conjunto de datos. Para este paso es necesario del uso de *Keras* y *TensorFlow* 2.0 para la construcción de la red neuronal. Así mismo se requiere disponer de acceso a la librería Google Bert disponible online por Google.

Finalmente, se muestra información detallada acerca de los resultados de KPIs obtenidos con las diferentes aproximaciones recomendando un modelo y publicando la información relevante para que se pueda reproducir dicha red para ser aplicados en otros entornos.

1.8 Recursos necesarios y presupuesto del proyecto

Para llevar a cabo el proyecto, se requiere algunos recursos materiales con un coste asociado:

Tabla 2. Lista de recursos requeridos en el proyecto.

Recursos	**Uso**	**Coste**
Laptop + pantallas + periféricos	Ordenador donde redactar la memoria. Instalar herramientas en laboratorio virtual (Anaconda).	1.000€
Conexión a Internet	Salida a internet necesaria para acceder a Google BERT	25€/mes
Contratación servicio Google Cloud Platform	Opción para ejecutar los notebooks cuando el laptop se quede corto de recursos (GPUs-FPUs)	20€/mes

Herramientas Opensource necesarias: Anaconda, Python, *Sklearn*, *Keras*, *Tensor Flow* 2.0 y *Google Bert*	Las herramientas anteriores no tienen coste en las versiones requeridas. Para un uso comercial sí que tienen coste.	0€

1.9 Breve descripción de los otros capítulos de la memoria

En el *capítulo 2* se indican los conceptos básicos de una red neuronal, y como dicha red puede ayudar a detectar potenciales *URL* maliciosas, simplemente porque existe un modelo entrenado con un conjunto de datos *[58]* con información de otras *URL* maliciosas, ya que dicho modelo ha aprendido a prever qué nuevas *URL* pueden ser potencialmente maliciosas.

Para ello, se explicará el proceso de selección del conjunto de datos y su contenido. También se creará una red neuronal profunda (*DNN*) básica que permitirá la predicción de nuevas *URL*.

Con todo ello se expondrán los resultados obtenidos haciendo hincapié en los hiperparámetros, buscando aquellos que mejor optimicen el rendimiento de la red sin caer en sobreajustes del modelo.

En el *capítulo 2.3* se avanzará creando una red neuronal recurrente (RNN) que permitirá la predicción de nuevas *URL*. También buscaremos mediante los hiperparámetros de la red los mejores resultados de la construcción de dicha red neuronal aplicado al mismo conjunto de datos que en el capítulo 2.

A continuación, en el *capítulo 3* se creará una red neuronal mediante un *Transformer [59]* que permitirá la predicción de nuevas *URL*. Aquí también buscaremos mediante hiperparámetros los mejores resultado con el mismo conjunto de datos empleado durante todo el trabajo.

Finalmente, en el *capítulo 0* mostraremos los resultados expuestos en detalle durante los capítulos anteriores de una forma resumida y clara para exponer cuales son las mejores tipos de redes neuronales encontradas para la detección de *URL Malicious*, cuales ofrecen el mejor rendimiento sin ofrecer sobreajustes sobre el modelo.

En el *capítulo 5* se ofrecen las conclusiones del trabajo fin de máster.

2 Fase de investigación y primera *DNN*

En este apartado se analizan los diferentes trabajos que se están realizando a nivel mundial para solucionar la detección de *URL* Maliciosas y se detallan las decisiones de diseño que se han tomado para la realización de este trabajo.

Finalmente, se explicará la primera implementación de referencia sobre el problema con una primera red neuronal profunda (*DNN*) explicando todo el proceso realizado para su implementación y mostrando los primeros resultados.

2.1 Estado del arte

URL es la abreviatura de *Uniform Resource Locator [26]*, que es la dirección global de documentos y otros recursos en la *World Wide Web*. Una *URL* tiene dos componentes principales:

- Identificador de protocolo (indica que protocolo usar: *http*, *https* etc.).
- Nombre de recurso (especifica la dirección IP o el nombre de dominio donde se encuentra el recurso). El identificador de protocolo y el nombre del recurso están separados por dos puntos y dos barras diagonales, por ejemplo, la Figura 1.

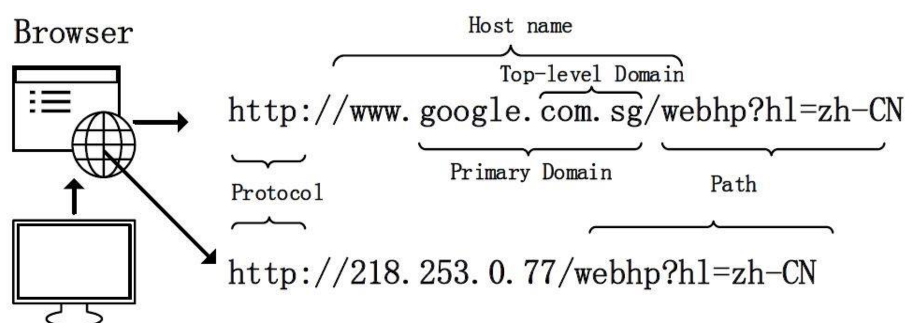


Fig. 3. Ejemplo de una URL - "Uniform Resource Locator".

Las *URL* comprometidas que se utilizan para ataques cibernéticos se denominan *URL* maliciosas. Una *URL* maliciosa o un sitio web malicioso aloja una variedad de contenido no solicitado en forma de *Spam*, *Phishing* o descarga automática para lanzar ataques.

Los usuarios desprevenidos visitan los sitios web y se convierten en víctimas de varios tipos de estafas, incluida la pérdida monetaria, el robo de información privada y la instalación de malware. Los tipos de ataques que se utilizan las *URL malicious* incluyen: *drive-by-download*, *Phishing* e Inteligencia Social y *Spam*.

Estos ataques son masivos y han causado daños por valor de miles de millones de dólares. Los sistemas efectivos para detectar tales *URL* maliciosas de manera oportuna pueden ayudar en gran medida a contrarrestar una gran cantidad y una variedad de amenazas de seguridad cibernética. En consecuencia, los

investigadores y profesionales han trabajado para diseñar soluciones efectivas para la detección de *URL* maliciosas.

El método más común para detectar *URL* maliciosas implementadas por muchos grupos antivirus es el método de lista negra. Las listas negras son esencialmente una base de datos de *URL* que se ha confirmado que son maliciosas en el pasado. Sin embargo, es casi imposible mantener una lista exhaustiva de *URL* maliciosas, especialmente porque todos los días se generan nuevas *URL*. Recientemente, con los servicios de acortamiento de *URL*, se ha convertido en una generalizada técnica de ofuscación (ocultar la *URL* maliciosa detrás de una *URL* corta). Los atacantes utilizan muchas otras técnicas para evadir las listas negras. Por lo tanto, los métodos de listas negras tienen limitaciones severas, y son inútiles para hacer predicciones en nuevas *URL*. Un primer intento para solucionar este problema surge de los enfoques heurísticos en los que se crean una "lista negra de firmas" identificando los atacantes comunes y asignándole una firma a este tipo de ataque. A continuación, los sistemas de detección de intrusos escanean las páginas web en busca de tales firmas y levantar una bandera si encuentra algún comportamiento sospechoso. El inconveniente es que este método sólo se aplica a un número limitado de amenazas comunes y no puede generalizarse a todos los tipos de ataques. Adicionalmente son poco tolerantes a las técnicas de ofuscación.

Ante esta problemática, en la última década, los investigadores han aplicado técnicas de aprendizaje automático para la detección de *URL* maliciosas. Para aplicar dichas técnicas es necesario disponer de datos de entrenamiento, y en función de las propiedades estadísticas, aprenden una función de predicción para clasificar una *URL*. Es conveniente extraer características informativas que describan adecuadamente una *URL* y, al mismo tiempo puedan ser interpretadas matemáticamente mediante modelos de aprendizaje automático. A continuación, el siguiente paso es la construcción del modelo de predicción mediante el entrenamiento real del modelo.

Hay dos enfoques en el aprendizaje automático para la detección de *URL* maliciosas:

- <u>Análisis estático</u>: las características extraídas incluyen características léxicas de la cadena de *URL*, información sobre el host y, a veces, incluso contenido *HTML* y *JavaScript*. Al no requerir ninguna ejecución este método es más seguro.

- <u>Análisis dinámico</u>: incluye la monitorización de secuencias de llamadas del sistema para detectar comportamientos anormales. Estas técnicas tienen riesgos inherentes siendo difíciles de implementar y generalizar.

A continuación, se muestra el proceso de aprendizaje para la detección de *URL* maliciosas usando técnicas de *Machine Learning*.

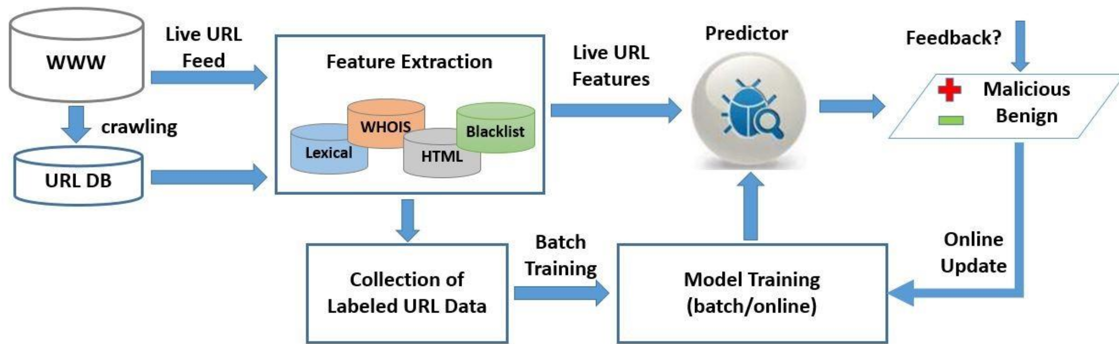


Fig. 4. Marco de proceso general para la detección de URL maliciosas mediante el aprendizaje automático.

2.1.1 Representación de características.

El éxito de un modelo de aprendizaje automático depende críticamente de la calidad de los datos de entrenamiento *[26]*, que a su vez depende de la calidad de la representación de características.

- *Colección*: esta fase está orientada a la ingeniería, que tiene como objetivo recopilar información relevante sobre la *URL*. Esto incluye información como la presencia de las *URL* en una lista negra, características obtenidas de la cadena de *URL*, información sobre el host, el contenido del sitio web como *HTML* y *JavaScript*, información de popularidad, etc.

- *Preprocesamiento*: en esta fase, la información no estructurada sobre la *URL* (por ejemplo, la descripción textual) se formatea adecuadamente y se convierte en un vector numérico para que pueda introducirse en algoritmos de aprendizaje automático. Por ejemplo, la información numérica se puede usar tal cual, y la bolsa de palabras se usa a menudo para representar contenido textual o léxico.

Tabla 3. Propiedades de diferentes representaciones de características para la detección de URL maliciosas.

Feature	**Category**	**Criteria**					
		Collection Difficulty	**Risk**	**External Dependency**	**Collection Time**	**Processing Time**	**Feature Size**
Blacklist	Blacklist	Moderate	Low	Yes	Moderate	Low	Low
Lexical	Traditional Advanced	Easy	Low	No	Low	Low	Very High Low

		Easy	Low	No	Low	High	
Host	Unstructured	Easy	Low	No	High	Low	Very High
	Structured	Easy	Low	No	High	Low	Low
Content	HTML	Easy	High	No	Depends	Low	High
	JavaScript	Easy	High	No	Depends	Low	Moderate
	Visual	Easy	High	No	Depends	High	High
	Other	Easy	High	No	Depends	Low	Low
Others	Context	Difficult	Low	Yes	High	Low	Low
	Popularity	Difficult	Low	Yes	High	Low	Low

2.1.2 Algoritmos de aprendizaje.

Existe una rica familia de algoritmos de aprendizaje automático en la literatura, que se pueden aplicar para resolver la detección de *URL* maliciosas. Después de convertir las *URL* en vectores de características, muchos de estos algoritmos de aprendizaje se pueden aplicar generalmente para entrenar un modelo predictivo de una manera sencilla *[26]*. Sin embargo, para resolver eficazmente el problema, también se han explorado algunos esfuerzos para diseñar algoritmos de aprendizaje específicos, en los cuales se explotan las propiedades exhibidas por los datos de entrenamiento de las *URL* maliciosas, o aborden algunos desafíos específicos que a los que se enfrenta su aplicación.

2.1.2.1 Aprendizaje por lotes

1. Máquina de soporte (*SVM*) es uno de los métodos de aprendizaje supervisado más utilizado, explota el principio de minimización de riesgos estructurales utilizando un enfoque de aprendizaje de margen máximo. Además, *SVM* puede aprender clasificadores no lineales usando *kernels*. Los *SVM* son probablemente uno de los clasificadores más utilizados para la detección de *URL* maliciosas en la literatura *[69][60][66][49][71][36][54][42][55][56][70][35][28][23][48]*.

2. Regresión logística es uno de los métodos de aprendizaje supervisado utilizado para predecir el resultado de una variable categórica (una variable que puede adoptar un número limitado de categorías) en función de las variables independientes o predictoras. Es útil para modelar la probabilidad de un evento ocurriendo en función de otros factores. Las probabilidades que describen el posible resultado de un

único ensayo se modelan como una función de variables explicativas, utilizando una función logística.

3. *Naive Bayes* es uno de los métodos de aprendizaje supervisado fundamentado en el teorema de *Bayes* y algunas hipótesis simplificadoras adicionales. Es a causa de estas simplificaciones, que se suelen resumir en la hipótesis de independencia entre las variables predictoras, que recibe el apelativo de "*naive*", es decir, ingenuo.

4. Árboles de decisión es uno de los métodos más populares para la inferencia inductiva y tiene una gran ventaja de sus modelos de clasificación de árbol de decisión altamente interpretables, que también se pueden convertir en un conjunto de reglas para la legibilidad humana.

5. Otros algoritmos y conjuntos:

    - *Extreme Learning Machines* (*ELM*) para clasificar los sitios web de *Phishing*.

    - Clasificación esférica que permite que los modelos de aprendizaje por lotes sean adecuados para muchas instancias también se utilizó para esta tarea.

    - *Clasificación de múltiples etiquetas es un m*étodo de dos pasos: primero usar *SVM* para clasificar una *URL* como maliciosa o benigna; y segundo, realizar la clasificación de múltiples etiquetas en las *URL* maliciosas utilizando algunos métodos populares de aprendizaje de múltiples etiquetas (por ejemplo, *RAkEL* y *ML-kNN*).

    - *Adaboost* para detectar sitios web de *Phishing* utilizando un enfoque basado en contenido junto con *Latent Dirichlet Allocation* (*LDA*) para el modelado de temas.

    - Conjunto de clasificadores múltiples para hacer una predicción ponderada. Entrenan de forma independiente árboles de decisión, bosques aleatorios, clasificadores bayesianos, máquinas de vectores de apoyo y regresión logística, y diseñan un esquema de votación por mayoría ponderada por confianza para hacer la predicción final.

    - Análisis multi-vista donde se entrena un modelo de regresión logística en diferentes porciones de las características léxicas de la *URL*, y se aprende su combinación óptima.

    - Método de optimización evolutiva para buscar la mejor combinación de características y modelos para obtener el conjunto final.

2.1.2.2 Aprendizaje profundo en línea (*Online Machine Learning*).

El aprendizaje en línea representa una familia de algoritmos *[50][43][44][46][67]* de aprendizaje eficientes y escalables que aprenden de los datos secuencialmente.

1. Aprendizaje en línea de primer orden.

    - *Perceptrón* según el principio del descenso de gradiente (Estocástico).

    - *Aprendizaje pasivo-agresivo* (PA) compensa dos preocupaciones, la pasividad, es decir, evitar que el nuevo modelo se desvíe demasiado del existente y la agresividad, actualizar el modelo corrigiendo el error de predicción tanto como sea posible.

2. Aprendizaje en línea de segundo orden. Tiene como objetivo aumentar la eficacia del aprendizaje mediante la explotación de información de segundo orden, por ejemplo, las estadísticas de segundo orden de las distribuciones subyacentes

    - *Aprendizaje ponderado por confianza* (*CW*) es similar a los algoritmos de aprendizaje de PA en términos de compensación de pasividad y agresividad, excepto que CW explota la información de segundo orden. En particular, el aprendizaje (*cw*) mantiene una medida de confianza diferente para cada característica individual, de modo que los pesos de menor confianza se actualizarán de manera más agresiva que los de mayor confianza.

3. Aprendizaje en línea sensible a los costos. A diferencia de una tarea de clasificación binaria normal, la detección de *URL* maliciosa a menudo se enfrenta a los desafíos de la distribución de etiquetas desequilibrada (es decir, un número diferente de *URL* maliciosas y benignas), y también a un costo diferencial de clasificación errónea (por ejemplo, la instalación de malware es mucho más grave que un simple spam). En consecuencia, los algoritmos de aprendizaje diseñados deben tener en cuenta esta tasa de clasificación errónea diferencial en el problema de optimización.

4. Aprendizaje activo en línea. Se basa en no asumir que el aprendizaje va a tener coste y desarrolla un algoritmo que entrena un modelo que consulta la etiqueta de una instancia *URL* entrante solo ante una necesidad, por ejemplo, con alguna medida de incertidumbre.

2.1.2.3 Aprendizaje de representación

1. Aprendizaje profundo para la detección de *URL* maliciosas. el aprendizaje profundo tiene como objetivo aprender características apropiadas directamente de datos (a menudo no estructurados) y realizar la clasificación utilizando estas características. Para la

detección de *URL* maliciosas, esto puede ayudarnos a reducir la ingeniería de características dolorosa y crear modelos sin ninguna experiencia en el dominio.

En consecuencia, se aplicaron *Redes Neuronales Convolucionales [40]* (*CNN*) para esta tarea. *eXpose [41]* aplicó redes convolucionales a nivel de caracteres *[68]* a la cadena *URL*. Este modelo no utiliza la codificación para las palabras y aun así el modelo ofreció un rendimiento superior en comparación con la aplicación de *SVM* en las características de *Bag-of-Words*.

Debido al éxito de los modelos *Bag-of-Words* en *Malicious URL Detection*, una extensión natural para aplicar modelos de aprendizaje profundo basados en *NLP*, debido a su capacidad para inferir características a partir de datos textuales no estructurados se implementaron modelos de redes neuronales recurrentes (*RNN*, *LSTM*) con esta misma aproximación.

URLNet [37] desarrolló un sistema de extremo a extremo donde se aplicaron circunvoluciones tanto para caracteres como para palabras en la *URL*, y demostró empíricamente que el uso de información a nivel de palabra junto con el nivel de carácter podría aumentar el rendimiento del modelo.

2. Métodos de selección de características y regularización de escasez. Hay un gran número y variedad de características utilizadas para la detección de *URL* maliciosas, y se han realizado estudios para determinar qué características pueden ser las más adecuadas. Los modelos con tantas características son computacionalmente caros y son ruidos y, por lo tanto, al optimizarlos tendemos a tener un sobreajuste.

   - Método de filtro. Usan una medida estadística para evaluar la idoneidad de una característica. Algunos enfoques populares incluyen la prueba de Chi cuadrado y los puntajes de ganancia de información.

   - Método contenedores. Utiliza algoritmos genéticos para realizar la selección de características y divide las características en críticas y no críticas. Las no críticas se utiliza para proporcionar información complementaria, en lugar de descartarse.

   - Regularización de escasez L1. Consiste en mejorar la exactitud de la predicciones e interpretabilidad de los modelo estadísticos de regresión al alterar el proceso de construcción del modelo y seleccionar solamente un subconjunto de las variables provistas para usar en el modelo final.

2.1.2.4 Otros métodos de aprendizaje

- Aprendizaje no supervisado. El aprendizaje no supervisado es el escenario en el que la verdadera etiqueta de los datos no está disponible durante la fase de entrenamiento. Estos enfoques se basan en técnicas de detección de anomalías donde la anomalía se define como un comportamiento anormal. Desafortunadamente, debido al conjunto extremadamente diverso de *URL*, es difícil determinar qué es un comportamiento "normal" y qué es una anomalía. Como resultado, tales técnicas no se utilizan asiduamente para la detección de *URL* maliciosas.

- Aprendizaje de similitud. El aprendizaje de similitud tiene como objetivo aprender qué tan similares son dos *URL* ayudándonos a identificar qué *URL* legítimas específicas están siendo imitadas por los atacantes. En este caso, hay un conjunto de *URL* protegidas y un conjunto de *URL* sospechosas que potencialmente intentan imitar las *URL* protegidas. El objetivo es medir la similitud de las *URL* de la sospecha con las *URL* protegidas, y si la similitud está por encima de un umbral específico, podemos detectar una *URL* maliciosa.

- Coincidencia de patrones de cadena. Las características léxicas a menudo se obtienen en forma de representación de un conjunto de palabras de dimensionalidad extremadamente alta. Sin embargo, aquí se propone un enfoque dinámico de minería de patrones de cadenas que tomando prestadas las ideas de búsqueda eficiente y la coincidencia de subcadenas.

2.1.3 Detección de *URL* maliciosas como un servicio.

En la práctica no es fácil la implementación de un servicio que pueda funcionar o ser efectivo. Debe cumplir los siguiente principios *[26]*:

2.1.3.1 Principios de diseño.

Factores más importantes:

- Precisión. Este es a menudo uno de los objetivos más importantes que se deben lograr para cualquier detección de *URL* maliciosa. Idealmente, queremos maximizar la detección de todas las amenazas de *URL* maliciosas.

- Velocidad de detección: La velocidad de detección es una preocupación importante para un sistema práctico de detección de *URL* maliciosas, particularmente para sistemas en línea o aplicaciones de ciberseguridad.

- Escalabilidad: Capaz de poder entrenar con millones de datos de entrenamiento. Para ello se plantea construir sistemas de aprendizaje

escalables en entornos de computación distribuida (*Apache Hadoop o Spark*).

- Adaptación: la distribución de *URL* maliciosas cambia con el tiempo o de manera adversaria para eludir el sistema de detección.

- Flexibilidad: El servicio debe ser flexible a mejoras y extensiones incorporando la actualización rápida de los modelos predictivos con respecto a los nuevos datos de entrenamiento.

2.1.3.2 *Frameworks* de diseño.

Ya existen casos de servicios que han desarrollado su propia implementación. Aquí vamos a nombrarlos brevemente:

- ***Monarch***: diseñó su propio *framework* y planteó la idea de proporcionar detección de *URL* maliciosas como servicio. *Monarch* rastrearía los servicios web en tiempo real y determinaría si la *URL* sería maliciosa o benigna. La implementación de este sistema comprende un agregador de *URL* que recopila *URL* de algunos flujos de datos. A partir de estas *URL*, se recopilan características, que luego se procesan y se convierten en características dispersas en el extractor de características. Finalmente, se entrena a un clasificador en los datos procesados para detectar *URL* maliciosas. Las características recopiladas incluyen tanto características basadas en *URL* como características basadas en contenido. Para el entrenamiento rápido de clasificación desde la perspectiva de la eficiencia en la memoria y las actualizaciones algorítmicas, un clasificador lineal basado en la regresión logística con el regularizador L1 (para inducir modelos dispersos) se entrena en una arquitectura de computación distribuida. El sistema es capaz de procesar una *URL* completa en 5.5 segundos. La predicción es relativamente eficiente.

- ***Prophiler***: clasificación de *URL* se realice en un proceso de dos etapas. La primera etapa sería analizar las características de *URL* livianos y filtrar rápidamente las *URL* para las cuales el clasificador tiene una alta confianza. Para las predicciones de baja confianza, se puede realizar un análisis más intensivo basado en el contenido.

- ***WarningBird***: el objetivo principal es detectar *URL* sospechosas en una secuencia, aunque utiliza heurística para obtener nuevas características, y el clasificador utilizado fue un modelo *SVM* que usa LIBLINEAR y no entrenado, y desplegado en una arquitectura distribuida.

- ***BINSPECT***: se desarrolló para aprovechar la clasificación de conjuntos, donde la predicción final se realizó sobre la base del voto mayoritario ponderado por confianza.

2.1.4 Problemas abiertos.

A pesar de muchos avances en la última década para la detección de *URL* maliciosas utilizando técnicas de aprendizaje automático, todavía hay muchos problemas y desafíos abiertos que son críticos e imperativos, incluidos, entre otros, los siguientes:

- *Alto volumen y alta* velocidad: los datos *URL* del mundo real son una forma de Big Data con alto volumen y alta velocidad.

- *Dificultad para adquirir etiquetas: la mayoría de los* enfoques de detección de *URL* maliciosas existentes mediante el aprendizaje automático se basan en técnicas de aprendizaje supervisado, que requieren datos de entrenamiento etiquetados, incluidos datos de *URL* benignos y maliciosos. Desafortunadamente, la escala de estos datos etiquetados es pequeña en comparación con el tamaño de todas las *URL* disponibles en la web.

- *Dificultad para recopilar* características: algunas características son costosas (en términos de tiempo) para recopilar, por ejemplo, características basadas en host. Algunas características pueden faltar, o ser ruidosas, o no se pueden obtener debido a una variedad de razones (por ejemplo, las direcciones *IP / DNS* de una *URL* pueden variar de vez en cuando).

- *Representación* de características: Además del alto volumen y la alta velocidad de los datos de *URL*, otro desafío clave son las características de muy alta dimensión (a menudo en escala de millones o incluso miles de millones). Esto plantea un desafío en la práctica para entrenar un modelo de clasificación.

- *Desafíos* emergentes: los hackers y delincuentes maliciosos siempre encontrarán formas de eludir los sistemas de seguridad cibernética.

- *Interpretabilidad de los* modelos: Una dirección de investigación importante es comprender qué hace que una *URL* sea maliciosa y validar qué patrones en las *URL* pueden ayudar a determinar la naturaleza maliciosa o benigna de una *URL*. Esto es particularmente difícil cuando se utilizan modelos de aprendizaje profundo, que a menudo se comportan como cajas negras.

- *Ataques* adversarios: los modelos de aprendizaje automático pueden simular *URL* para evitar los detectores de *URL* maliciosos basados en aprendizaje automático.

2.1.5 Áreas relacionadas.

La detección de *URL* maliciosas está estrechamente relacionadas con otras áreas de investigación, como:

- *Detección de Spam*: algunas técnicas son Content Spam, Link Spam, *Cloaking* and *Redirection*, y *Click Spam*, así como las técnicas utilizadas para contrarrestarlos.

- *Clasificación de páginas web*: utilizan en común características de contenido (etiquetas de texto y *HTML* en la página) y características de vecinos (clasificación basada en la etiqueta de clase de páginas web similares).

Resumiendo, la detección de *spam*, la clasificación de páginas web y la detección de *URL* maliciosas utilizan algunos tipos similares de características y técnicas para resolver su respectivo problema. En la práctica, estos métodos y características pueden complementarse entre sí para mejorar el rendimiento de los modelos de aprendizaje automático.

2.2 Decisiones de diseño TFM.

A continuación, se van a mostrar las decisiones de diseño por el que se ha implementado una serie de *Jupyter Notebooks* (ver *Anexos*).

2.3 Selección del conjunto de datos.

Tal y como se indica en el apartado anterior, referido al estado del arte en la detección de *URL* maliciosas, es muy importante la selección de un buen conjunto de datos *[58]*.

Uno de los temas claves es la búsqueda de un conjunto de datos *[58]* que sea suficientemente rico y preciso sobre la temática a tratar (*URL malicious*) y aunque sí que se han encontrado bastantes conjuntos de datos sobre esta temática, sólo unos pocos cumplían los requisitos que buscábamos que principalmente eran, que fueran públicos, usados por otros trabajos y, finalmente, que estuvieran suficientemente actualizados.

Con estos criterios se encontró un conjunto de datos proporcionado por el *National Center for Biotechnology Information [58]* que dispone de amplia información sobre las *URL* maliciosas detectadas y bastante actualizado (2020).

Entrando al contenido del conjunto de datos hemos visto algunos aspectos que han debido ser tratado convenientemente en la posterior fase de modelado.

2.3.1 Datos desbalanceados.

El balance entre páginas web maliciosas respecto a las benignas está muy desequilibrado en cantidad *[58]*.

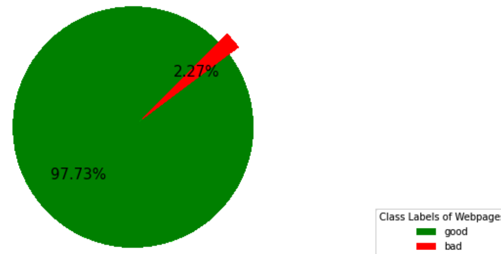


Fig. 5. En rojo las páginas web maliciosas y en verde las benignas (2,27% vs 97,73%).

Esta característica del conjunto de datos debe ser tenida en cuenta a la hora de trabajar con algún tipo de red neuronal que sea sensible a esta característica (ver *3.1.3.1*).

2.3.1.1 Importancia de la distribución geográfica.

El conjunto de datos *[58]* debe tener en cuenta el origen de la ubicación física del host en el que está desplegada la página web.

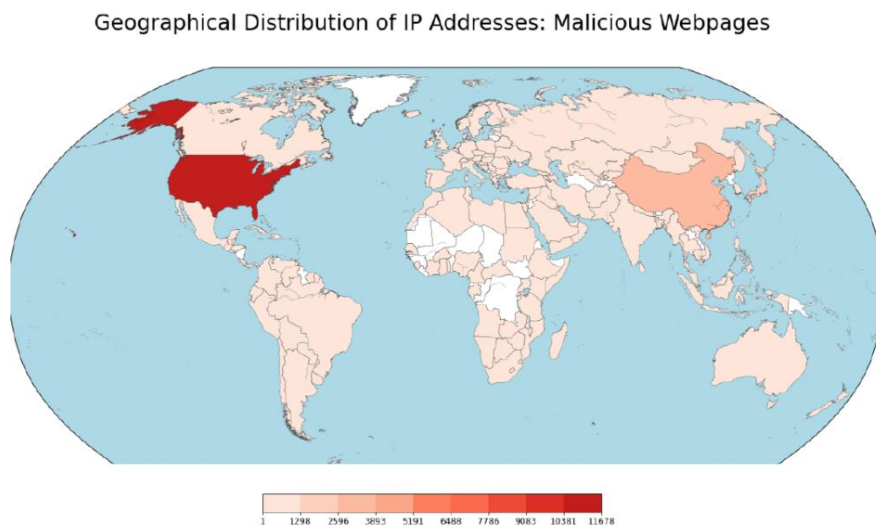


Fig. 6. De más oscuro a más claro el número de peticiones por host.

2.3.1.2 Importancia de la longitud del código JavaScript embebido en la *URL*.

Se aprecia que en las páginas web que tiene mucho código *JavaScript* son más susceptibles de ser etiquetadas como *URL* maliciosas *[26]*.

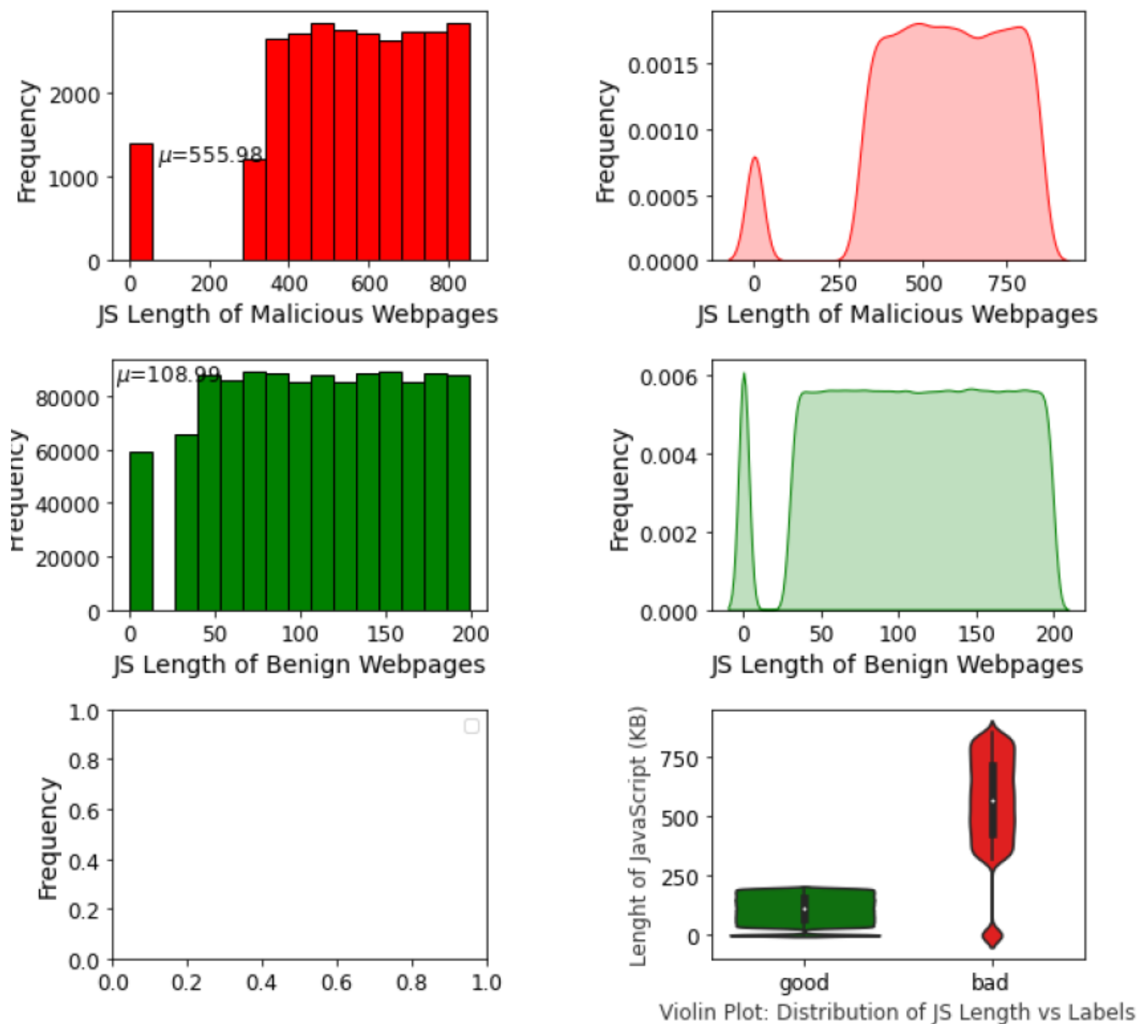


Fig. 7. Longitud del código JavaScript en páginas maliciosas (rojo) vs páginas benignas (verde).

Este mismo patrón todavía es más extremo en el caso de las páginas que disponen de código JavaScript ofuscado.

2.3.1.3 Importancia de obtener respuesta al comando '*who_is*'.

En la figura inferior se aprecia que en las páginas web benignas se devuelven casi todos los parámetros de información relativas a la petición del comando '*who_is*' respecto a las páginas malignas en las que, en algunos casos, estas páginas no muestran toda esta información manteniéndola oculta.

Fig. 8. Respuesta completa e incompleta a las URL maliciosas(rojo) vs las URL benignas (verde) [30].

2.3.2 Selección y transformación de las características.

Las características han sido transformadas siguiendo los siguientes pasos:

1. *Características numéricas*: '*url_len*', '*js_len*' o '*js_obf_len*' se mantienen como están y, la única transformación a realizar es la estandarización de los datos sobre estos campos.

2. *Generación de nuevas características categóricas*.

De '*ip_add*' a '*net_type'*. Se ha convertido la dirección IP en la clase de red (clase A, B o C) pasando a ser una variable categórica.

De '*content'* a '*special_char*'. Se cuentan los caracteres 'especiales', es decir no alfanuméricos en el contenido de la *URL*.

De 'content' a '*content_len'*. Se cuenta el total de caracteres en el contenido de la *URL*.

3. *Tratamiento de características categóricas*. Después de los pasos anteriores tenemos variables categóricas como son *'geo_loc'*, *'tld'*, *'who_is'*, 'https', *'net_part'* y 'net_type'. A todas ellas se les aplican las técnicas *Label Enconder* y *One-Hot Encoder*.

El resultado hasta el paso 2 supone 10 dimensiones en el conjunto de datos *[31]* (la clase '*label'* no cuenta ya que la extraemos por separado).

	url_len	geo_loc	tld	who_is	https	js_len	js_obf_len	label	net_type	special_char	content_len
0	0.296446	179	92	0	1	-0.679601	-0.134936	0	0	-0.775701	-0.453198
1	-0.265653	192	92	0	1	-0.740941	-0.134936	0	0	-1.028241	-0.964011
2	-0.827752	8	92	0	1	-0.172149	-0.134936	0	0	-0.391401	-0.298263
3	-1.038540	192	130	1	0	6.703547	8.887442	1	1	6.295408	6.862516
4	-0.054866	192	92	0	1	-0.807858	-0.134936	0	2	-0.951381	-0.859783

Fig. 9. Visualización de los primeros registros del conjunto de datos después del paso 2.

En este momento (después del paso 2) queremos realizar una correlación de variables para detectar si se produce algún tipo de correlación entre las mismas y apreciamos que la categoría '*js_len*', '*js_obf_len*' y '*label'* están directamente correlacionadas.

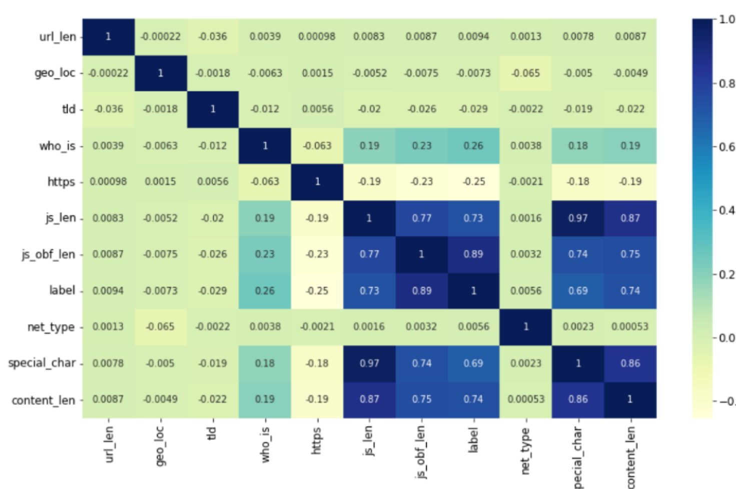


Fig. 10. Correlación de Pearson en variables (después del paso 2).

Después del paso 3 (Tratamiento de características categóricas), vemos que las variables pasan de ser 10 a ser 718 (excluimos la variable objetivo '*label*').

	url_len	who_is	https	js_len	js_obf_len	label	special_char	content_len	geo_loc_0	geo
0	0.296446	0	1	-0.679601	-0.134936	0	-0.775701	-0.453198	0.0	
1	-0.265653	0	1	-0.740941	-0.134936	0	-1.028241	-0.964011	0.0	
2	-0.827752	0	1	-0.172149	-0.134936	0	-0.391401	-0.298263	0.0	
3	-1.038540	1	0	6.703547	8.887442	1	6.295408	6.862516	0.0	
4	-0.054866	0	1	-0.807858	-0.134936	0	-0.951381	-0.859783	0.0	

5 rows × 719 columns

Fig. 11. Visualización de los primeros registros del conjunto de datos después del paso 3.

Por lo tanto, nuestro algoritmo tendrá que tratar con 718 dimensiones.

Una vez realizada esta labor dividimos los datos en tres conjuntos de *URL* maliciosas, 80.000 para entrenamiento, 20.000 para test y otros 20.000 para validación.

2.3.3 Selección del algoritmo de aprendizaje.

Visto el estado del arte en la detección de *URL* maliciosas nos hemos planteado acometer tres redes neuronales que, a nivel de algoritmos de aprendizaje, se englobarían en Aprendizaje de Representación, más concretamente en Aprendizaje profundo para la detección de *URL* maliciosas (ver *15*).

A continuación, se indican los tres tipos de algoritmos seleccionados:

- *DNN*: esta es una primer implementación de un algoritmo de Aprendizaje básico y que nos ha servido para calibrar la calidad del conjunto de datos *[58]*.

- *RNN*: se ha optado por una implementación de una algoritmo *LSTM* cuyas virtudes están altamente probadas en el ámbito del *NLP* (*Natural Language Processing*).

- *Transformer*: se desea experimentar con alguna red *Transformer [59]* que ha revolucionado el resultado en el ámbito del *NLP* en los últimos tres años. Para ello, se experimentará con soluciones *BERT* con el objetivo de obtener un óptimo resultado aplicado al problema de detección de *URL* maliciosas.

2.4 Primera red neuronal (*DNN*).

Una red neuronal profunda (*DNN*) es una red neuronal artificial (ANN) con múltiples capas ocultas entre las capas de entrada y salida. Existen diferentes tipos de redes neuronales, pero siempre constan de los mismos componentes: neuronas, sinapsis, pesos, sesgos y funciones.

Estos componentes funcionan de manera similar a los cerebros humanos y se pueden entrenar como cualquier otro algoritmo de *Machine Learning*.

2.4.1 Creación del modelo.

Para la creación del modelo usaremos la librería *Keras*. *Keras* es una biblioteca de Redes Neuronales de código abierto escrita en Python que es capaz de ejecutarse sobre *TensorFlow*, *Microsoft Cognitive Toolkit* o *Theano*.

Está especialmente diseñada para posibilitar la experimentación en más o menos poco tiempo con redes de Aprendizaje Profundo *[40]* (*Deep Learning*). Su fortaleza se centra en ser amigable para el usuario , modular y extensible.

2.4.2 Detalles de diseño de la arquitectura de la *DNN*.

A continuación, se define la primera red neuronal profunda *DNN*. Observaremos varios puntos a tener en cuenta:

- *Dimensiones*:
  - La dimensión de la capa de entrada debe ser igual al total de las características de entrada del entrenamiento (sin incluir las de salida). Es decir, 718 dimensiones *[31]*.
  - La capa de entrada ha sido dimensionada como la suma de las características de entrada (718) y salida (1) dividido por dos *[31]*.

    Como disponemos de 718 características de entrada y una de salida el número debería ser unas 359 (719/2). Se ha redondeado a 350.
- *Función de activación*:
  - Se decide que en las capas intermedias la función de activación será *RELU [31]*. Esta suele ser una decisión bastante usual en los problemas de clasificación, lo que permite que las neuronas de las capas intermedias se especialicen.
  - Se decide que en la capas de salida la función de activación será *Sigmoide [31]*. Esto es debido a que para las decisiones probabilísticas esta función está demostrado que funciona mejor (similar a la regresión logística).
- *Función de entrada*:
  - 350 *units* (número de neuronas) porque es la media aproximada entre los nodos de entrada (711) y los nodos de salida (1).
  - *kernel_initializer*="uniform". Pesos pequeños, pero no 0 *[31]*.
  - Función de activación RELU *[31]*.
  - Dimensión de los datos de entrada: *input_dim* = 718 *[31]*.
- *Función de salida*:
  - No es necesaria función *Softmax* al haber sólo dos opciones. La probabilidad siempre sumará uno. Si tuviéramos más de tres opciones de salida, deberíamos tener tres características y es probable que no sumaran 1. Para ese caso sería muy interesante la aplicación de la función *Softmax [31]*.

2.4.3 Detalles de diseño de la compilación de la DNN.

Una vez establecida la arquitectura es necesario establecer los detalles de diseño para compilar la red:

- Selección del optimizador:
  - se decide que en las utilizar el optimizador de *Adam [31]*.
- Selección de la función de pérdida:
  - se decide que en las utilizar la función de pérdida de *binary_crossentropy [31]*.
- Selección de la métrica de la precisión:
  - se decide que en las utilizar la métrica de precisión (*'accuracy'*) *[31]*.

2.4.4 Pasos de entrenamiento de la DNN mediante el método del Gradiente Descendente Estocástico.

Se han de seguir los siguientes pasos para inicializar el entrenamiento de nuestra red neuronal (*DNN*) *[31]*:

- Paso 1: Inicializar los pesos aleatoriamente con valores cercanos a 0 (pero no 0).
- Paso 2: Introducir la primera observación del conjunto de datos en la capa de entrada, cada característica es un nodo de entrada.
- Paso 3: Propagación hacia adelante: de izquierda a derecha, las neuronas se activan de modo que la activación de cada una se limita por los pesos. Propaga las activaciones hasta obtener la predicción *y*.
- Paso 4: Se compara la predicción con el resultado real. Se mide entonces el error generado.
- Paso 5: Propagación hacia atrás de derecha a izquierda, propagando el error hacia atrás. Se actualizan los pesos según la responsabilidad del error de los pesos. El ratio de aprendizaje gobierna cuánto deben actualizarse los pesos.
- Paso 6: Dos opciones:
  - Se repiten los pasos 1 a 5 y se actualizan los pesos después de cada observación (*Reinforcement Learning*).
  - Se repiten los pasos 1 a 5 pero se actualizan los pesos después de un conjunto de observaciones (*Batch Learning*).
- Paso 7: Cuando todo el conjunto de entrenamiento ha pasado por la DNN se completa un *epoch*. Hacer N *epochs*.

2.4.5 Entrenamiento de la red.

Se han tenido que realizar una serie de decisiones de diseño mientras se estaba decidiendo cómo entrenar la *DNN*:

- '*batch_size*': se decide que en el proceso de *batch* procese de 10 en 10 registros y luego calibre los pesos.

- '*epochs*': 25 epochs.

Resultado:

- El resultado sobre los datos de entrenamiento da una precisión del *99.98%* (en *25 epochs*) con el que sabremos si una página Web es maliciosa o no.
- Es una precisión bastante alta teniendo en cuenta que esta es la primera red neuronal generada sin haber aplicado mecanismos de optimización sobre los hiperparámetros.

2.4.6 Evaluar el modelo y calcular predicciones.

A continuación, se muestra la matriz de confusión en la que se ve que la mayoría de las muestras de test, 19.977 tienen una predicción correcta (suma primera diagonal) y únicamente 23 muestras (suma resto de celdas) han errado en la predicción.

```
Confusion matrix:
 [[  450    23]
 [    0 19527]]

Accuracy test:  99.885 %
```

Fig. 12. Matriz de confusión y precisión sobre los datos de test.

Vemos que se consiguen valores muy altos para los datos test (99.885) pero no llega al nivel de los datos de entrenamiento (99.98%).

Sospechamos que puede estar pasando que la división entre datos de entrenamiento y test no sea suficientemente óptima o que tengamos un ligero sobreentrenamiento (*overfitting*) sobre el modelo.

2.4.7 *k-Fold Cross Validation*.

Mediante la selección de dos conjuntos de entrenamiento y de test se ha procurado compensar el sesgo (sesgo = 1 - precisión) y la varianza para estabilizar el modelo procurando un bajo sesgo y una baja varianza *[31]*.

Para ello, un paso más allá es, utilizar la técnica de *k-Fold Cross validation [31]* que divide el conjunto de entrenamiento *[58]* en pliegues (*folds*). Esto permite que en cada iteración se utilice un conjunto de pliegues (n-1) para entrenar y un pliegue (*fold*) para validar. En cada iteración se modifica cual es el pliegue (*fold*) de validación evitando el sesgo y no tener que separar desde el principio entre subconjunto de datos de entrenamiento y prueba.

De esta forma se consigue mayor estabilidad en el modelo acotando el sesgo y la varianza *[31]* entre la selección de diferentes datos para la construcción de este. El objetivo será tener un sesgo lo más pequeño posible manteniendo una varianza lo más baja posible.

2.4.8 Hiperparámetros.

La computación en el entrenamiento de la red neuronal es costosa y el algoritmo debe probar todas las combinaciones de los parámetros presentados. Se ha optado por tres hiperparámetros y 18 combinaciones.

De esta forma, cuando llamemos a esta función además de los hiperparámetros básicos como '*batch_size*' o '*nb_epoch*' podremos también cambiar el algoritmo del optimizador (parámetro '*optimizer*').

Para probar este mecanismo de forma automática se ha usado la clase '*GridSearchCV*' *[31]* de la librería '*scikit-learn*'.

Para este mecanismo hemos realizado dos pasadas, una con datos básicos cercanos a los ofrecidos en la primera red neuronal con los siguientes valores:

- Primera pasada:
    - Parámetros:

      `'batch_size': [16,32], 'nb_epoch': [10,20], 'optimizer': [ 'adam', 'rmsprop'].`

    - Mejor resultado:

      `{'batch_size': 16, 'nb_epoch': 10, 'optimizer': 'adam'}.`

        - Precisión: 99.91%.
- Segunda pasada:
    - Parámetros:

```
'batch_size': [8,16,24], 'nb_epoch': [5,10,15], 'optimizer': [
'adam', 'rmsprop'].
```

- Mejor resultado:

```
{'batch_size': 16, 'nb_epoch': 5, 'optimizer': 'adam'}.
```

  - Precisión: 99.93%.

2.4.9 Análisis de la función de pérdida y sobreentrenamiento ('loss function' y '*overfitting*').

Hemos monitorizado el efecto de la función de pérdida y de la precisión en dos gráficas en un entrenamiento de 25 épocas sobre la misma red neuronal. En ella vemos que a partir de la época 10, la función de pérdida (*loss*) se hace cada vez más pequeña sobre los datos de entrenamiento y más grande sobre los datos de test. Esto nos indica que a partir de aproximadamente la décima época el modelo se ve afectado por sobreentrenamiento. Para verificarlo podemos comprobar en la gráfica que monitoriza la precisión (*accuracy*) como también a partir de la décima época sube la precisión en el entrenamiento mientras se mantiene estable o incluso baja levemente en los datos de test.

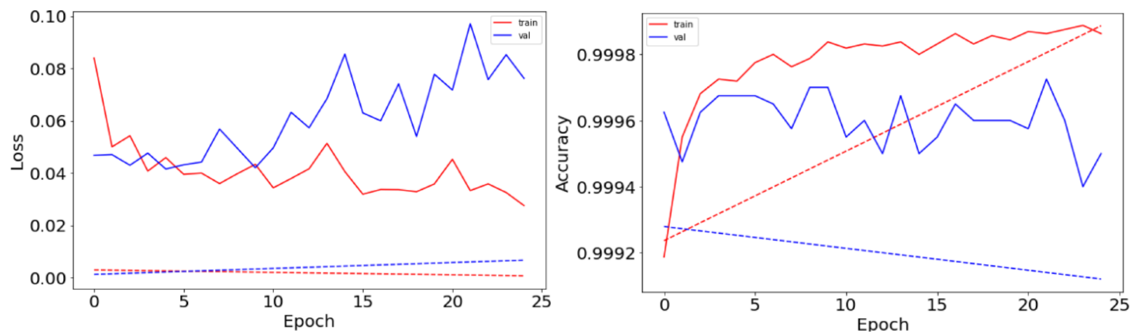


Fig. 13. Sobreentrenamiento (overfitting).

Como obtenemos un buen ratio con épocas inferiores a 10 el sobreentrenamiento no va a afectar para conseguir un buen modelo con alta precisión (superior al 99.9%).

Si hubiéramos tenido problemas con el sobreentrenamiento, sin llegar al resultado precisión objetivo, hubiéramos tenido que aplicar alguno de los siguientes refinamientos:

- Implementar una *'callback'* sobre la función de pérdida: para evitar el *overfitting* se suele implementar un *callback* que pare cuando vea que la función de pérdida tiene tendencia creciente ('*EarlyStoppingByLossVal*').

- Aplicar un ratio de aprendizaje personalizado: otro método para suavizar el comportamiento de la función de pérdida es utilizar el objeto *'LearningRateScheduler'* para modular el comportamiento de la función de pérdida suavizándola y haciéndola más predecible.

2.4.9.1 *Data Augmentation* y *Transfer Learning*.

Nos planteamos si estas dos técnicas podrían ayudar a mejorar el modelo que tenemos actualmente

2.4.9.2 Data Augmentation

Sobre la técnica de *Data Augmentation* es una técnica que mejora aquellos entornos en los que se dispone de pocas muestras o bien, que aquellas muestras no indiquen toda la información que es precisa para el modelo.

Dado los resultados de precisión (99.895%) obtenidos con los hiperparámetros adecuados en nuestra primera *DNN*, siendo que sólo hemos utilizado 80.000 muestras de los 1.2M de las muestras disponibles creemos que esta técnica no nos va a ayudar a mejorar los resultados obtenidos.

2.4.9.3 Transfer Learning

Transfer Learning [11] es un método de aprendizaje automático en el que un modelo desarrollado para una tarea se reutiliza como punto de partida para un modelo en una segunda tarea.

El aprendizaje por transferencia está relacionado con problemas como el aprendizaje multitarea y la deriva de conceptos y no es exclusivamente un área de estudio para el aprendizaje profundo.

Las mayores aplicaciones de esta técnica suelen utilizarse en aprendizaje con datos de imagen *[40] [11]* (*CNN*) y con datos lingüísticos *[11]* (*RNN*) dado que en estas áreas existen buenos modelos de referencia que pueden ser aprovechados. Cuando acometamos las redes *RNN [40]* deberemos tener en cuenta partir de modelos ya existentes más generalistas como son '*word2vect*' de *Google* o *GloVe* (*Global Vectors for Word Representation*) de Stanford *[11]*.

Lo que sí se puede hacer es utilizar otros modelos para enriquecer nuestro conjunto de datos *[58]*, principalmente partiendo del campo '*content*' y realizando un proceso previo. Hemos experimentado con algunos atributos:

- <u>*Análisis de sentimiento*</u>: se refiere al uso de procesamiento de lenguaje natural, análisis de texto y lingüística computacional para identificar y extraer información subjetiva. El análisis de sentimientos es una tarea de clasificación masiva de textos de manera automática, en función de la connotación positiva o negativa del lenguaje ocupado en el texto. Estos tratamientos generalmente se basan en relaciones estadísticas y de asociación, no en análisis lingüístico".

  Se ha usado la función '*sentiment_polarity*' (librería '*textblob*').

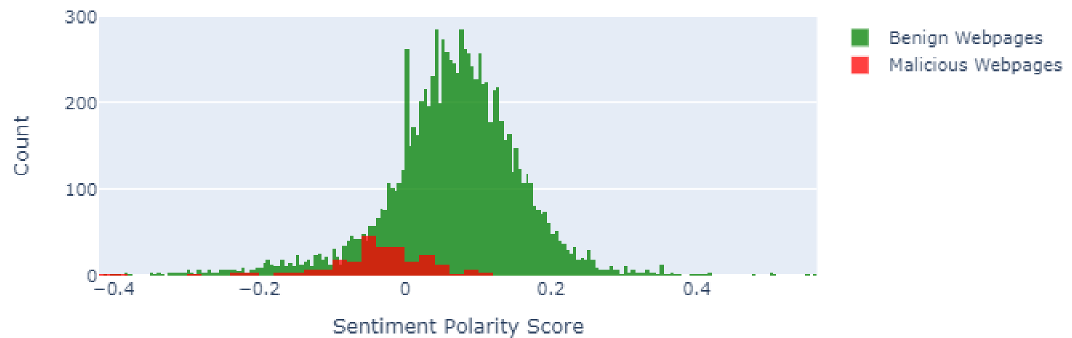


Fig. 14. ‘sentiment_polarity’ en el conjunto de datos URL maliciosas.

Aplicado a nuestro conjunto de datos *[58]* vemos que las páginas maliciosas están derivadas hacia una puntuación negativa.

- *Análisis de palabras malsonantes* (en inglés ‘*profanity*’): durante el proceso de entrenamiento, el modelo aprende qué palabras son "malas" y qué tan "malas" son porque esas palabras aparecerán con más frecuencia en los textos ofensivos. Este método es mucho más eficiente que la utilización de un método de lista negra.

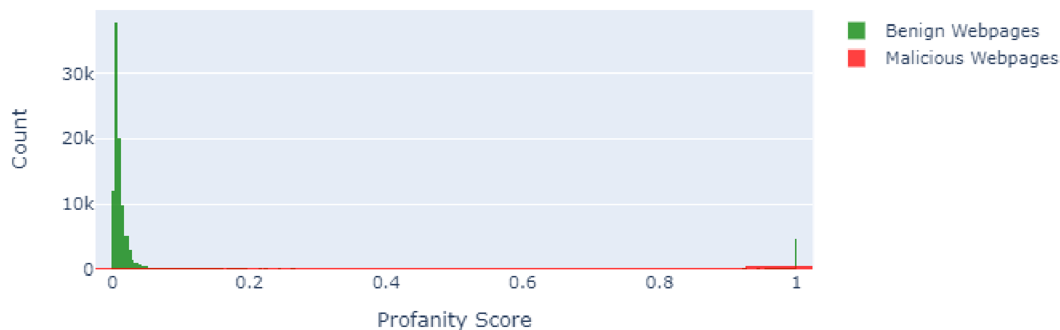


Fig. 15. ’profanity-check’ en el conjunto de datos URL maliciosas.

Aplicado a nuestro conjunto de datos *[58]* vemos que las páginas maliciosas acumulan bastante nivel de palabras malsonantes mientras que las páginas benignas están cercanas a 0. Se ha utilizado la función ‘*predict_prob*’ de la librería ‘*profanity-check*’. Utiliza un modelo SVM lineal entrenado con 200k muestras etiquetadas por humanos.

- *Análisis de errores ortográficos* (*GECwBERT*): una idea es analizar los errores ortográficos en las páginas. Hemos encontrado un modelo (GECwBERT) que es capaz de detectar los errores gramaticales en un texto (GED). Esta idea se desarrollará cuando expongamos nuestra red

neuronal con *Transformer [59]* (ver *Fase de desarrollo de RNN y Transformer*).

- *Análisis de publicidad*: una idea es analizar si las páginas web tienen más o menos publicidad asignándoles un índice numérico que va desde 0 sin publicidad a 1 lleno de publicidad. Hay algunos modelos experimentales que están diseñados para detectar y ser capaces de extraer este factor *[5]*.

3 Fase de desarrollo de *RNN* y *Transformer*

En este apartado se trata el mismo problema de *URL* maliciosas desde el punto de vista de las redes neuronales recurrentes (*RNN*) presentando diferentes opciones, en esencia se plantean las redes neuronales *Long short-term memory (LSTM)* planteando diferentes variantes aplicadas sobre la misma *URL* o el contenido de las páginas web. También veremos la variante *Gated Recurrent Units* (*GRU*) presentándola como una evolución de las redes *LSTM*.

Finalmente, la segunda parte de este apartado trata las redes Transformer y un nuevo paradigma que ha revolucionado en los últimos años el Proceso Natural del Lenguaje (*NLP* por sus siglas en inglés) e incluso se ha extendido a otras áreas de conocimiento. Se explica el modelo de atención tal y como lo presenta el artículo que lo presentó *All you need is attention [7]*.

3.1 Redes Long short-term memory (LSTM)

Las redes *Long short-term memory* (*LSTM*), son un tipo de red neuronal recurrente (*RNN)* creadas para dar solución al problema *vanishing gradient* encontrado en las RNN clásicas. Fueron introducidas por primera vez en el artículo [57], y desde entonces han sido mejoradas y popularizadas. Hoy en día son utilizadas ampliamente en una gran variedad de problemas, especialmente para el modelado de secuencias temporales.

Esta arquitectura ampliamente utilizada hoy en día debido a su superioridad rendimiento en la modelización precisa de las dependencias de datos a corto y largo plazo, se encuentra implementada en distintas librerías de programación. En nuestro caso usaremos la librería *Keras*.

Las redes *LSTM* surgen con el objetivo de resolver el problema de la desvanecimiento del gradiente [34] (*vanishing gradient*) sin imponer ningún sesgo sobre observaciones recientes y manteniendo la transferencia del error a través del tiempo. El problema consiste en que, el gradiente se irá desvaneciendo a valores muy pequeños, impidiendo al peso cambiar su valor eficazmente. En el peor caso, esto puede impedir que la red neuronal continúe su entrenamiento en la búsqueda de un mínimo absoluto en la función de coste.

3.1.1 Arquitectura LSTM

Una red neuronal recurrente (*RNN*) es la forma de una cadena de módulos repetitivos que se "desenvuelven" simulando una red estándar *[63]*. En las RNN tradicionales, estos módulos disponen de una estructura simple, con una sola capa de activación, por ejemplo: la función *tanh*.

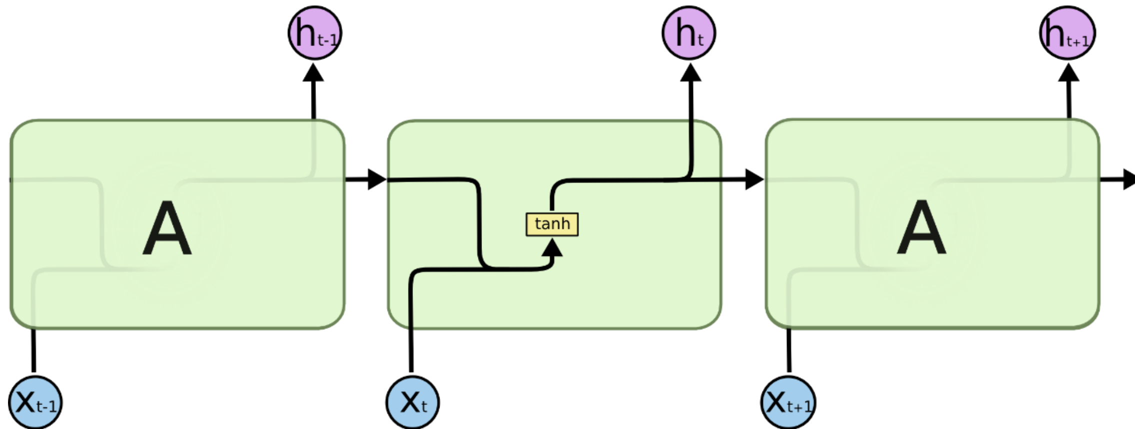


Fig. 16. RNN estándar en una sola capa [63].

Las redes *Long short-term memory* (*LSTM*) tienen esta estructura en cadena *[63]*, en que la unidad de cálculo se llama célula de memoria, y están compuestas por pesos y compuertas. Es decir, en lugar de tener una sola función de activación, hay cuatro, interactuando de distintas formas y cumpliendo funciones de almacenamiento y descarte de información.

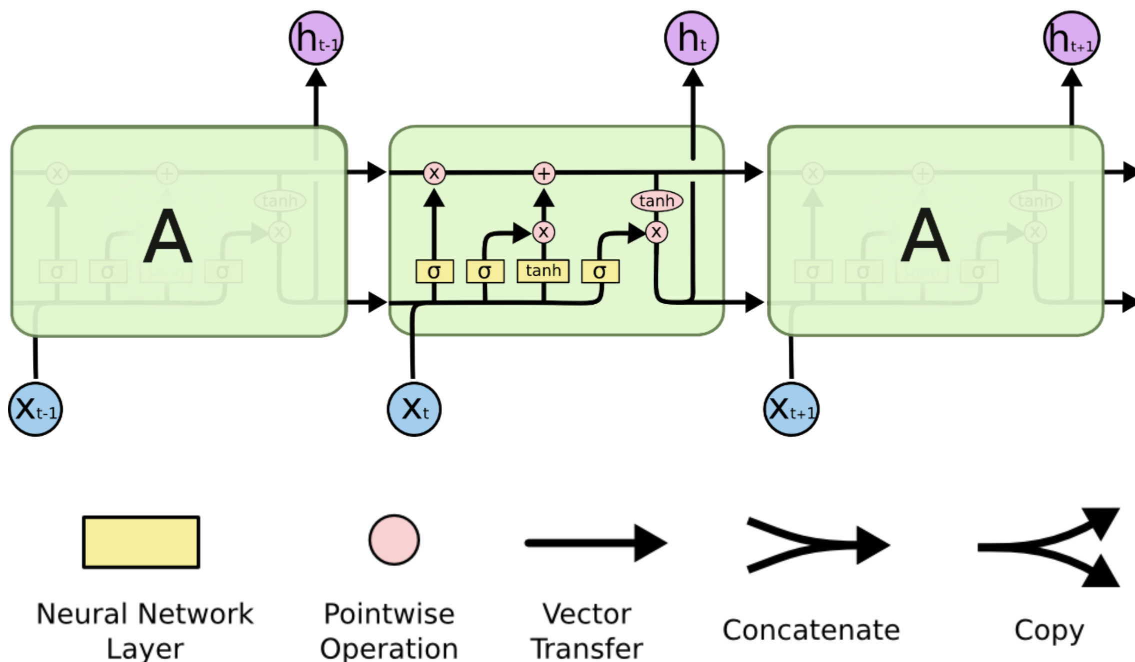


Fig. 17. Arquitectura simplificada en una red LSTM [63].

Una capa de una red LSTM consiste en un conjunto de bloques conectados recurrentemente. La red se basa en el estado de las celdas (*cell state*), en este

caso, en la línea horizontal que atraviesa la parte superior del diagrama (ver *Fig. 18*).

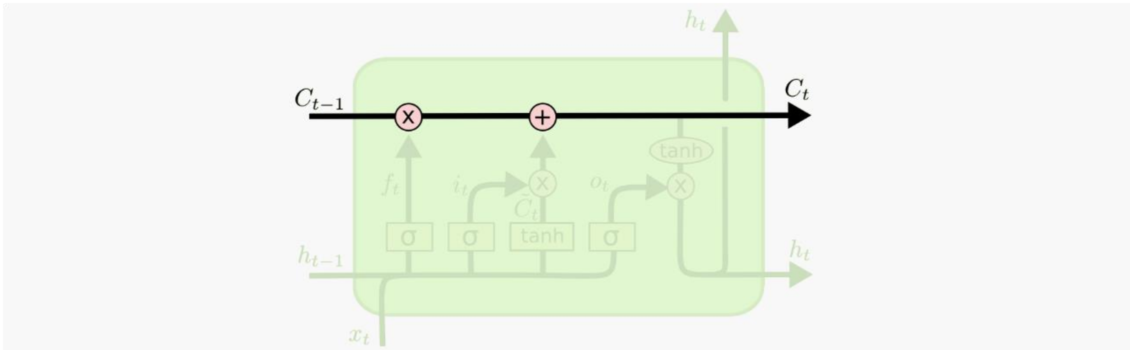


Fig. 18. Cell State, flujo de información a través de la red LSTM [63].

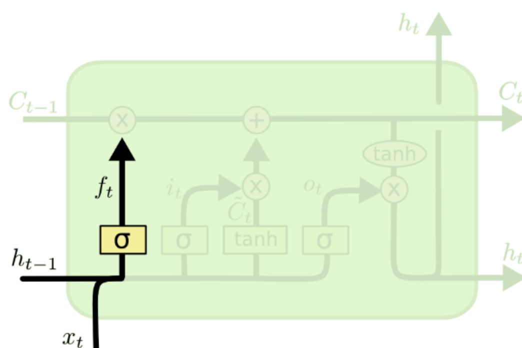
Este estado actúa como una especie de cinta transportadora fluyendo la información a lo largo de toda la cadena, con sólo algunas interacciones lineales menores.

La red tiene la capacidad de eliminar o añadir información al estado de la celda, siendo esto regulado por las estructuras llamadas compuertas y que permiten pasar la información opcionalmente y están formadas por:

- **Forget Gate** (compuerta de olvido): decide qué información se descarta.
- **Input Gate** (compuerta de entrada): decide qué valor se actualiza.
- **Output Gate** (compuerta de salida): decide los valores de salida basados en la información que se mantiene y que se introdujo de entrada.

3.1.2 Flujo de información y formulación

Inicialmente, las redes *Long short-term memory* (*LSTM*) debe decidir qué información van a descartar. Esta decisión la toma una función de activación sigmoide denominada *forget gate*.

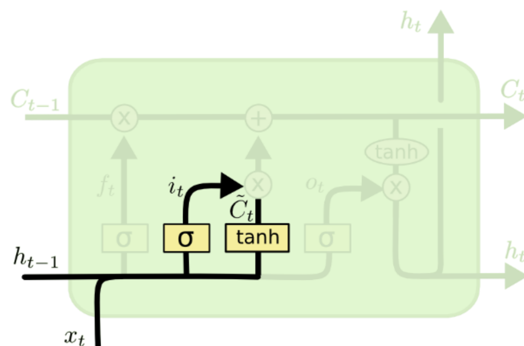


$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \ + \ b_f\right)$$

Fig. 19. Arquitectura de la compuerta forget gate [63].

Esta puerta está conectada con el intervalo anterior (recibe h_{t-1}) e información de entrada en el tiempo t (x_t) produciendo un número entre 0 y 1 para cada registro del estado anterior (C_{t-1}). Un valor 1 representa "mantener toda la información" mientras que un 0 representa "olvidar toda la información".

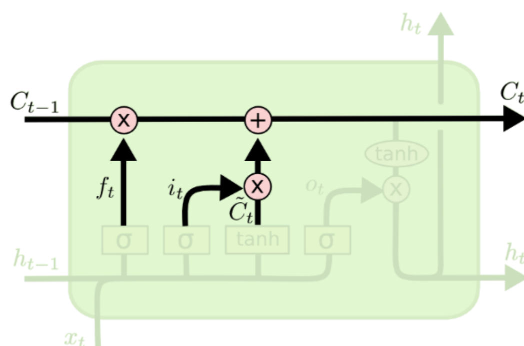
El siguiente paso, consiste en decidir la nueva información que se almacenará. Para ello, primero, una función sigmoide llamada "*input gate*" determina qué valores se actualizarán. Posteriormente, una función *tanh* crea un vector de candidatos C_t que podrían ser añadidos al nuevo estado.



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \ + \ b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \ + \ b_C)$$

Fig. 20. Arquitectura de la compuerta input gate [63].

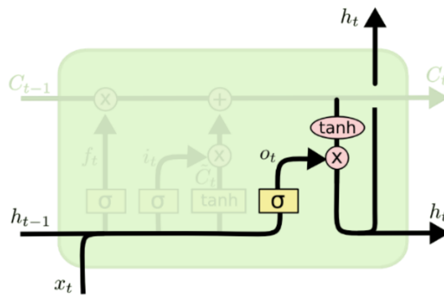
A continuación, se procede a combinar estos dos resultados multiplicando escalarmente el estado anterior C_{t-1} por la "*forget gate* (f_t) añadiendo los valores candidatos, escalados según la "*input gate* (i_t)".



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Fig. 21. Actualización del estado [63].

Finalmente, se definen los valores de salida (*output gate*). Estos estarán basados en el nuevo estado de la celda calculado anteriormente, pero serán filtrados por una función de activación *tanh* y una sigmoide que multiplicará a los estados ocultos (*hidden*) del momento anterior (t-1)



$$o_t = \sigma \left(W_o \ [h_{t-1}, x_t] \ + \ b_o \right)$$
$$h_t = o_t * \tanh \left(C_t \right)$$

Fig. 22. Output gate basado en el estado de la celda [63].

3.1.3 LSTM en nuestro escenario

La aplicación de *Long short-term memory* (*LSTM*) viene dado porque el objetivo es centrar la detección en sólo dos características de la página web, '*url*' y '*content*' que corresponden respectivamente con la *URL* de la página y a su contenido.

Es decir, la red no va a procesar previamente la información (al contrario que en *2.3.2*), sino que va a tratar con los datos brutos del contenido o de la *URL* tratando la secuencia de caracteres como una secuencia temporal con el objeto de aplicar todas las bondades esperadas en las redes neuronales recurrentes, más concretamente en las *LSTM*.

3.1.3.1 Desbalanceo del conjunto de datos.

Si bien ya detectamos en el capítulo anterior que los datos estaban desbalanceados (ver *2.3.1*), esto no derivó en un problema para representar la red neuronal profunda (*DNN*, ver *2.4*) que presentamos en el capítulo anterior con una precisión muy alta (99.91%). Sin embargo, las redes neuronales LSTM son muy sensibles a esta característica y ni siquiera llegan a entrenarse si no somos capaces de sistematizar un método que nos permita llegar a un nivel superior. Hay varios métodos para realizar el balanceo de los datos *[16]* y el que mejor se ajusta a nuestro conjunto de datos ha sido el método presentado como *undersampling [16]*.

Este método *undersampling [16],* permite seleccionar todas las *URL* maliciosas existentes en el conjunto de datos, en este caso 27.253 *URL* en un conjunto de 1.2M de *URL* y, por otro lado, se recuperan otras tantas *URL* benignas (obviamente necesitamos muchas menos muestras para obtenerlas). De esta forma, hemos igualado el porcentaje de benignas y malignas para poder construir nuestro modelo. Nos podemos permitir este método debido a que disponemos de 1.2M de *URL* lo cual es bastante respecto a las muestras necesarias para entrenar nuestra red neuronal. En caso de disponer de pocas muestras iríamos a otro método llamado *oversampling [16],* que consiste en generar *URL* malignas para poder equilibrar el conjunto de datos. En cualquier caso, para este caso hemos comprobado que con 54.506 *URL* la red neuronal es capaz de entrenar sin problemas por lo que hemos procedido a aplicar el método de *undersampling* con éxito.

3.1.3.2 Búsqueda sobre '*url'* y '*content'*.

Se ha intentado detectar si una *URL* era maliciosa o no entrenando la red LSTM sólo con el conjunto de datos de las *URL* ignorando la información de la característica '*content'*. Los resultados han presentado unas predicciones con precisiones demasiado bajas obteniéndose porcentajes entre el 60% y 70%, dependiendo del algoritmo empleado y, optimizando sus hiperparámetros. No se ha profundizado más en esta búsqueda debido a que los resultados parecen no ser comparables con los resultados obtenidos con una red neuronal artificial clásica tras un tratamiento de datos adecuado (ver *2.4*).

Llegado a este momento, para este análisis, se ha procedido a profundizar en la característica '*content'* que incluye el contenido de la página web (excluidos los caracteres extraños) y tratarla como si fuera un diccionario de palabras al estilo de un problema *NLP* clásico.

Para este tratamiento lo que se implementa es un vocabulario con las distintas palabras que encontramos en todas las páginas web con la frecuencia en la que aparecen y con ello, sustituimos cada palabra con un código que hace referencia a dicha palabra. También aplicamos una técnica de *padding* (consiste en que todas las páginas tengan la misma longitud rellenando con un código 0 las páginas cortas y truncando más allá del número máximo de palabras establecidas en las páginas más largas).

Por otro lado, se procede a utilizar métodos ya vistos en el capítulo anterior como *k-fold Cross Validation* (ver *2.4.7*), aplicando técnicas *GridSearch* para la selección de los hiperparámetros óptimos (ver *2.4.8*) y fijándonos en el posible sobreentrenamiento del modelo (ver *2.4.9*). No merece la pena profundizar en estos métodos (tratados en el apartado anterior) y revisaremos la optimización del modelo centrándonos en las características específicas de las redes neuronales recurrentes (*RNN*).

3.1.3.3 Optimizaciones del modelo *RNN*

El modelo presentado nos plantea las siguientes posibles optimizaciones:

- **Relacionadas con la arquitectura de la red** *[29] [19]*:
    - Aplicar una capa de *masking* previa. El enmascaramiento (*Masking [62]*) es una forma de contar capas de secuencia de procesamiento (tokens) indicando qué debe ser saltado al procesar los datos. El relleno (*padding [62]*) es una forma especial de enmascaramiento, donde los pasos son enmascarados al principio o al final de una secuencia. El relleno proviene de la necesidad de codificar datos de secuencia en lotes contiguos (*batchs*) para que todas las secuencias de un lote (*batch*) se ajusten a una longitud estándar determinada, para ello, es necesario rellenar o truncar algunas secuencias.

Se muestra el detalle del resultado de este modelo en *8*. La precisión de las previsiones nos da un resultado del 73.22%.

- Aplicar una capa 'embed' previa. La incrustación (Embedding *[62]*) consiste en convertir números enteros positivos (índices) en vectores densos de tamaño fijo.

  Esta técnica es la que se ha utilizado para conseguir los mejores resultados conseguidos con redes neuronales recurrentes en la parte práctica (ver *8*). Para ello, hay dos parámetros clave, uno es el número de palabras del diccionario (N_DICTIONARY) y, el más importante, el número de palabras por cada *URL* (N_STEPS).

  Se muestra el detalle del resultado de este modelo en *8*. La precisión de las previsiones nos da un resultado del 95.08%.

- Aplicar una capa convolucional *1D adicional a la de embedding*. Las redes convolucionales son muy efectivas en el ámbito de clasificación de imágenes, aunque también son adecuadas en las redes neuronales recurrentes para pronosticar series de tiempo en convoluciones dilatadas *[33]* o la capacidad de usar filtros para calcular las dilataciones entre cada celda. Es decir, el tamaño del espacio entre cada celda, que a su vez permite que la red neuronal comprenda mejor las relaciones entre las diferentes observaciones en la serie temporal. Este modelo se basa en una capa previa de *embedding* por lo que la capa convolucional 1D es una capa intermedia entre dicha capa y la capa LSTM.

  Se muestra el detalle del resultado de este modelo en *8*. La precisión de las previsiones nos da un resultado del 95.08%. En esta red neuronal hemos podido ampliar el tamaño de *embedding* a 1024 consiguiendo resultados del 97.39% que son los resultados más altos conseguidos con redes neuronales recurrentes, LSTM o GRU durante las pruebas realizadas.

- Aplicar una segunda capa oculta LSTM. Se pueden encadenar las capas LSTM que sean necesarias. En la práctica sólo se usan una o dos capas ocultas LSTM siendo en la mayoría de los casos suficiente una única capa y sólo en algunos problemas muy complejos es necesario una segunda capa oculta.

  En este trabajo, introducir una segunda capa LSTM no ha mejorado la precisión del modelo. Por ello, hemos prescindido de esta mejora en la implementación final (ver *8*).

- Aplicar una capa Gated Recurrent Unit (GRU *[52]*). Las redes neuronales *GRU* son una generación más moderna de redes neuronales recurrentes (*RNN*) y es bastante similar a una red

LSTM. Una red *GRU [62]* no dispone del estado de la celda necesaria en *LSTM* y utiliza el estado oculto para transferir la información. También tiene solo dos puertas, una puerta de reinicio (*reset gate*) y una puerta de actualización (*update gate*) simplificando el modelo respecto a *LSTM*.

Se muestra el detalle del resultado de este modelo en *8*. La precisión de las previsiones nos da un resultado del 95.08%.

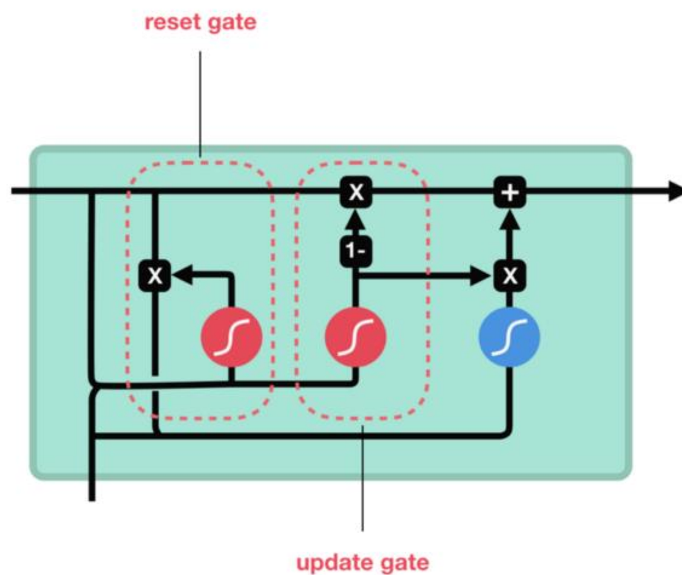


Fig. 23. Arquitectura simplificada Gated Recurrent Unit (GRU).

La puerta de actualización (*update gate*) actúa de manera similar a la puerta de entrada y olvido de una red *LSTM*. Decidiendo qué información desechar y qué información nueva agregar.

La puerta de reinicio (*reset gate*) es el otro tipo de puerta y se usa para decidir cuánta información pasada debe ser olvidada.

Las redes neuronales *GRU* disponen de menos operaciones de tensores; por lo tanto, son redes algo más rápidas durante el proceso de entrenamiento respecto a las redes *LSTM*.

En nuestro caso, la precisión de las predicciones con las redes GRU es comparable a la de las redes *LSTM*, aunque sí se percibe que el tiempo de entrenamiento es algo menor (ver *8*).

- Regularización L1 y L2. Con la regularización se intenta evitar que se aprenda información compleja, por lo que queremos suavizar el modelo para que no aprenda a memorizar los datos de entrenamiento tan rápidamente. Se combina normalmente con la regularización de abandono (*dropout*) y con la parada anticipada. Estos dos conceptos se verán más adelante.

La regularización *L1 [51]* hace referencia a *Lasso Regression* y L2 a *Ridge Regression [51]*. Mientras la regresión *L1* añade un valor lineal la regresión *L2* introduce un factor cuadrático.

$$\sum_{i=1}^{n}(Y_i - \sum_{j=1}^{p} X_{ij}\beta_j)^2 + \lambda \sum_{j=1}^{p} |\beta_j|$$

$$\sum_{i=1}^{n}(y_i - \sum_{j=1}^{p} x_{ij}\beta_j)^2 + \lambda \sum_{j=1}^{p} \beta_j^2$$

Ecuación 1. Regularización función de coste L1 (Lasso) superior y L2 (Ridge) inferior.

En nuestro caso, se ha decidido optar por los parámetros de regularización de *Dropout* y por la parada anticipada ya que este parámetro (*Dropout*) y método (parada anticipada) ha sido suficientes para controlar el sobreentrenamiento (*overfitting*) de la red.

- **Relacionadas con la configuración** *[29] [19]*:

  - Optimizador de aprendizaje adaptativo: para manejar mejor la compleja dinámica de entrenamiento de las redes neuronales recurrentes (que un descenso de gradiente simple puede no abordar), se recomiendan optimizadores adaptativos como *Adam*.

    En este trabajo se ha optado directamente por el optimizador *Adam*, en la literatura no hay duda de que es el óptimo para evitar los problemas asociados al desvanecimiento del gradiente (*vanishing gradient*). La selección del algoritmo *LSTM* o *GRU* respecto a cualquier red clásica *RNN* es el método más eficaz para solventar este inconveniente *[31]*.

  - *Gradient clipping*: los picos en el gradiente pueden alterar los parámetros durante el entrenamiento. Una solución común y relativamente fácil de aplicar al problema de los gradientes explosivos es cambiar la derivada del error antes de propagarlo hacia atrás a través de la red y usarlo para actualizar los pesos. Dos enfoques incluyen cambiar la escala de los gradientes dada una norma de vector elegida y recortar los valores de gradiente que exceden un rango preferido. Juntos, estos métodos se denominan *gradient clipping*.

    No se ha encontrado este problema durante el trabajo realizado por lo que no ha sido necesario la aplicación de esta técnica.

  - Normalización de la función de pérdida: agregar los términos de pérdida a lo largo de la secuencia y luego dividirlos por la

longitud máxima de la secuencia. Esto promediará la pérdida en el lote y, a su vez, facilitará la reutilización de los hiperparámetros entre experimentos.

Esto aplica a series más que al problema que nos aplica. En nuestro caso que aplicamos *embeddings* no tiene sentido la aplicación de este método.

- Retro-propagación truncada: cualquier forma de red recurrente puede tener dificultades para aprender secuencias largas debido a gradientes ruidosos y que se desvanecen *[14]*. La retro-propagación truncada a través del tiempo, o TBPTT, es una versión modificada del algoritmo de entrenamiento BPTT para redes neuronales recurrentes donde la secuencia se procesa un paso de tiempo a la vez y periódicamente (k1 pasos de tiempo), mientras que la actualización BPTT se realiza hacia atrás para un número fijo de pasos de tiempo (k2 pasos de tiempo). Este es algoritmo:

  - Presentar una secuencia de k1 pasos de tiempo de pares de entrada y salida a la red.
  - Desenrollar la red y luego calcular y acumular errores en k2 pasos de tiempo.
  - Reunir la red y actualizar los pesos.
  - Repetir.

  El algoritmo TBPTT requiere la consideración de dos parámetros:

  - k1: el número de pasos de tiempo de paso hacia adelante entre actualizaciones. Esto influye en lo lento o rápido que es el entrenamiento, dada la frecuencia con la que se realizan las actualizaciones de peso.
  - k2: el número de pasos en los que se aplicará BPTT. Generalmente, debe ser lo suficientemente grande para capturar la estructura temporal en el problema para que la red aprenda. Un valor demasiado grande da como resultado la desaparición de los degradados.

  No ha sido necesario el uso del algoritmo TBPTT en la ejecución de este trabajo.

- **Hiperparámetros relevantes** *[29] [19]*:

  - Número de nodos y capas ocultas: como regla general, una capa oculta en redes LSTM o GRU funcionará con la mayoría de los problemas simples y dos capas con los razonablemente complejos *[27]*. Además, aunque muchos nodos (con técnicas

de regularización) dentro de una capa pueden aumentar la precisión, una menor cantidad de nodos puede causar un ajuste inadecuado.

En nuestro caso únicamente ha sido necesaria la aplicación de una capa oculta ya que una segunda no mejoraba la precisión de las predicciones de la clasificación binaria.

- Número de unidades en la capa densa: esto ser refiere al número de neuronas en las capas ocultas. Es clave escoger un número por encima del mínimo que nos permita entrenar la red neuronal. Por debajo de este número de unidades mínimo la red tendremos *underfitting* y la red será incapaz de obtener resultados adecuados.

  Se han realizado pruebas variando entre 2 y 256 neuronas. Finalmente se ha optado por un valor intermedio de 32 aunque se ha detectado que, todos los valores intermedios (se han probado algunas potencias de 2, 4, 8, 16, 32, 64, 96,128,192 y 256), han ofrecido buenas prestaciones eligiendo el valor de 32 por la búsqueda combinada de precisión y rapidez de entrenamiento.

- *Dropout*: cada capa LSTM debe acompañarse por una capa de *dropout*. Esta capa evita el sobreentrenamiento al saltarse un porcentaje de neuronas de forma aleatoria durante la fase de entrenamiento reduciendo la sensibilidad a pesos específicos de una neurona concreta. No deben ser aplicadas sobre las capas de salida. Este hiperparámetro es considerado una forma de regularización bastante óptima.

  El mejor valor de dropoput encontrado ha sido 0.5 que significa que la mitad de las neuronas de la capa de entrada y oculta están deshabilitadas. El objetivo de esta regularización es evitar el sobreentrenamiento y conseguir unos resultados consistentes en la precisión en la predicción en el conjunto de datos de entrenamiento respecto al de validación.

- Inicialización de pesos: es mejor emplear diferentes esquemas heurísticos de inicialización de ponderaciones según la función de activación que se utilice *[17]*. Sin embargo, normalmente se usa una distribución uniforme al elegir los valores de peso iniciales. No es posible establecer todos los pesos en 0.0 ya que el algoritmo de optimización resalta la asimetría en el gradiente de error; para comenzar a buscar de manera efectiva. Estas heurísticas más personalizadas pueden resultar en un entrenamiento más efectivo *[17]* de modelos de redes neuronales utilizando el algoritmo de optimización de descenso de gradiente estocástico.

Diferentes conjuntos de pesos dan como resultado diferentes puntos de partida del proceso de optimización, lo que potencialmente conduce a diferentes conjuntos finales con diferentes características de rendimiento. Finalmente, los pesos deben inicializarse aleatoriamente a números pequeños (una expectativa del algoritmo de optimización estocástica, también conocido como descenso de gradiente estocástico) para aprovechar la aleatoriedad en el proceso de búsqueda.

En nuestro algoritmo hemos presentado un sistema de inicialización que compensara el desbalanceo de la información mediante el uso de la variable *bias*. Finalmente, y dado el correcto funcionamiento del algoritmo no ha sido necesario la utilización de esta técnica.

- <u>Tasa de decaimiento</u>: la disminución de peso se puede agregar en la regla de actualización de peso que hace que los pesos disminuyan a cero exponencialmente, si no se programa ninguna otra actualización de peso *[15]*. Después de cada actualización, los pesos se multiplican por un factor ligeramente inferior a 1, lo que evita que crezcan demasiado. Esto especifica la regularización en la red.

  No ha sido necesario modificar el ratio de decaimiento (*decay rate*) de la red LSTM ni GRU presentada en este trabajo.

- <u>*Learning rate*</u>: la tasa de aprendizaje es un hiperparámetro que controla cuánto cambiar el modelo en respuesta al error estimado cada vez que se actualizan los pesos del modelo. Elegir la tasa de aprendizaje es un desafío, ya que un valor demasiado pequeño puede resultar en un proceso de entrenamiento largo que podría atascarse, mientras que un valor demasiado grande puede resultar en aprender un conjunto de pesos subóptimo demasiado rápido o un proceso de entrenamiento inestable *[15]*.

  Este parámetro está directamente relacionado con la tasa de decaimiento (*decay rate*) *[15]*.

  Ecuación 2. Relación entre learning rate (lrate) y decay rate (decay) [15].

```
lrate = initial_lrate * (1 / (1 + decay * iteration))
```

  Conjuntamente definen la rapidez con la que la red actualiza sus parámetros. Hay que encontrar un equilibrio para que la tasa sea rápida al inicio pero que converja. Para ello se suele implementar una tasa de aprendizaje decreciente durante la fase de entrenamiento.

  Como se ha comentado no ha sido necesario tener que optimizar este parámetro respecto a los valores existentes por defecto en la librería *Keras*.

- <u>Función de activación</u>: se ha usado *softmax* para la salida debido a que el problema es de clasificación binaria y *tanh* en la capa *LSTM*.

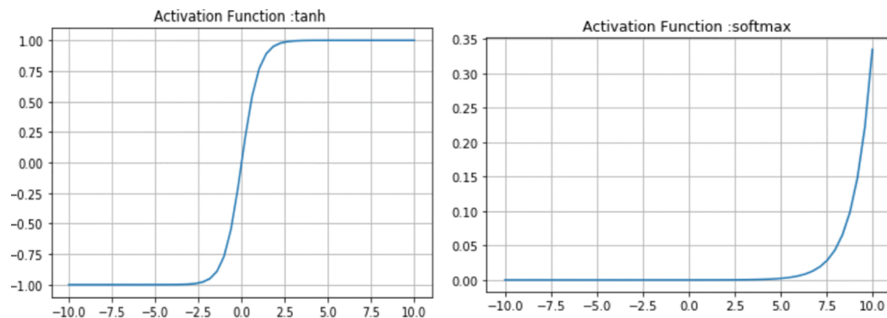


Fig. 24. Función tanh y softmax.

La función de activación tangente hiperbólica (*tanh*) es la función por defecto y la establecida de facto como la óptima para la red *LSTM*.

La función de activación *softmax* es necesaria debido a la naturaleza del problema de clasificación haciendo que la suma de las probabilidades de salida sume el valor uno. En este caso, habría una alternativas tratando el problema como una regresión y usando en este caso como función de activación de la última capa la función sigmoide y una sola neurona en la última capa (tratamiento como regresión). Se ha preferido mantener este esquema más adecuado para un problema de clasificación con dos neuronas en la capa de salida y la función de activación *softmax*.

- <u>*Momentum*</u>: el impulso (*momentum*) puede suavizar la progresión del algoritmo de aprendizaje que, a su vez, puede acelerar el proceso de entrenamiento. La adición de impulso acelera el entrenamiento del modelo. Específicamente, los valores de impulso entre 0,9 y 0,99 logran una precisión razonable de entrenamiento y prueba.

En el caso del trabajo realizado no se considera que sea necesario añadir el impulso (*momentum*) que por defecto va desactivado en las librerías de *Keras* ya que, las redes *LSTM* o *GRU* tienen un comportamiento voraz llegando en pocas épocas a conseguir el objetivo planteado.

- <u>Número de épocas</u>: número de iteraciones completas del conjunto de datos se ejecutarán.

Se ha implementado el concepto de parada anticipada para parar cuando no haya mejora en la función de pérdida de los datos de validación.

Adicionalmente, se ha visto que el número de épocas necesarias es muy bajo (entre 2 y 3) para conseguir un correcto equilibrio de la función de pérdida entre el conjunto de entrenamiento y de validación.

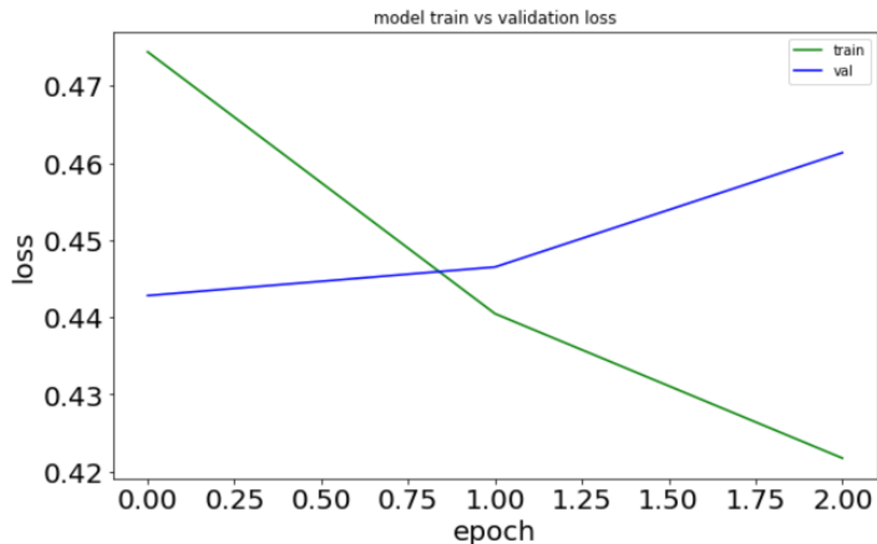


Fig. 25. Función de pérdida.

- *Batch size*: el número de muestras en las que trabajar antes de que se actualicen los parámetros internos del modelo en cada iteración.

  Este valor puede ser diferente para el entrenamiento y para la previsión *[12]*. En el entrenamiento se suele buscar un parámetro óptimo que de estabilidad y predictibilidad al proceso de entrenamiento. En la fase de predicción depende del número de datos a predecir. Si se va a usar con un conjunto de datos suficiente, por ejemplo, en la fase de test se puede mantener el mismo número que el establecido para entrenamiento.

  En nuestro caso no se encuentran grandes diferencias, pero se ha obtenido una ligera mejora en la precisión de la predicción en fase de validación con el valor 96 (respecto a 32, 64 o 128) empleado tanto para la fase de entrenamiento como de validación.

- Parada anticipada: consiste en implementar el método de detención anticipada (implementado como *callback* de *Keras EarlyStopping*) para especificar primero una gran cantidad de épocas de entrenamiento y detener el entrenamiento una vez que el rendimiento del modelo deja de mejorar por un umbral preestablecido en el conjunto de datos de validación.

  Esto debe ser utilizado junto a la regularización de *Dropout* y las de *L1* o/y *L2* ya que las redes *LSTM* o *GRU* tienen tendencia a

obtener resultados muy rápidos en pocas épocas y adicionalmente rápidamente tienden hacia el *overfitting*.

En este trabajo se han usado los tres métodos para conseguir suavizar la búsqueda de la solución en varias épocas (originalmente con dos épocas era suficiente) y poder parar el algoritmo en el momento en el que se consigue estabilizar la función de pérdida sobre los datos de validación.

3.1.4 Limitaciones de RNN, LSTM y GRU

Al analizar un texto en una red neuronal recurrente las palabras que son analizadas inicialmente tienen un peso menor que las que son analizadas hacia el final *[2]*. Este efecto implica que la red no dispone de memoria de largo plazo impidiendo relaciones entre palabras que están muy alejadas entre ellas debido a esta carencia.

Con la entrada del artículo *All you need is attention [7]*, la comunidad ha revisado la aproximación de las redes recurrentes hacia el nuevo concepto de Atención.

Las redes RNN, LST, GRU y derivados utilizan principalmente procesamiento secuencial a lo largo del tiempo. Esta flecha significa que la información a largo plazo tiene que viajar secuencialmente hasta llegar a la celda de procesamiento actual.

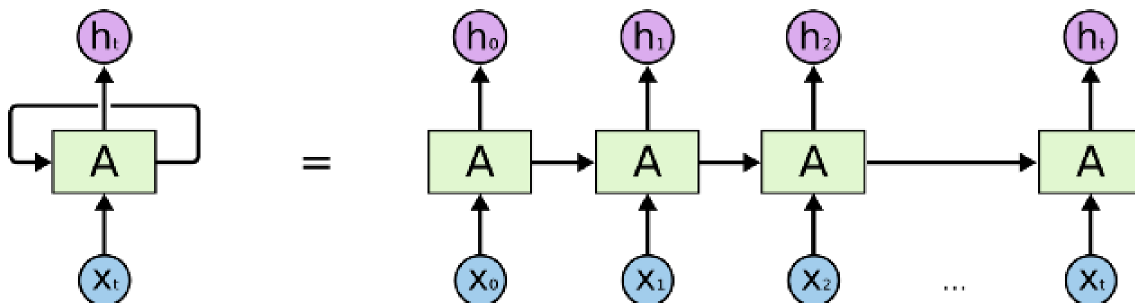


Fig. 26. Secuencialidad de las redes neuronales recurrentes (RNN) [21].

Para solucionar el problema de las *RNN* llegaron las redes *LSTM* y *GRU* que reducían el problema de evaporación (*vanishing*) del gradiente, pero añadía un esquema complejo de puertas pudiendo aprender bastante información, pero llegando a límites de cientos de secuencias no pudiendo llegar al orden de miles o superior.

Para evitar el error de dar mayor peso a las palabras cercanas respecto las lejanas se propone la utilización de módulos de atención *[21]* para resumir todos los vectores codificados en un vector de contexto C_t.

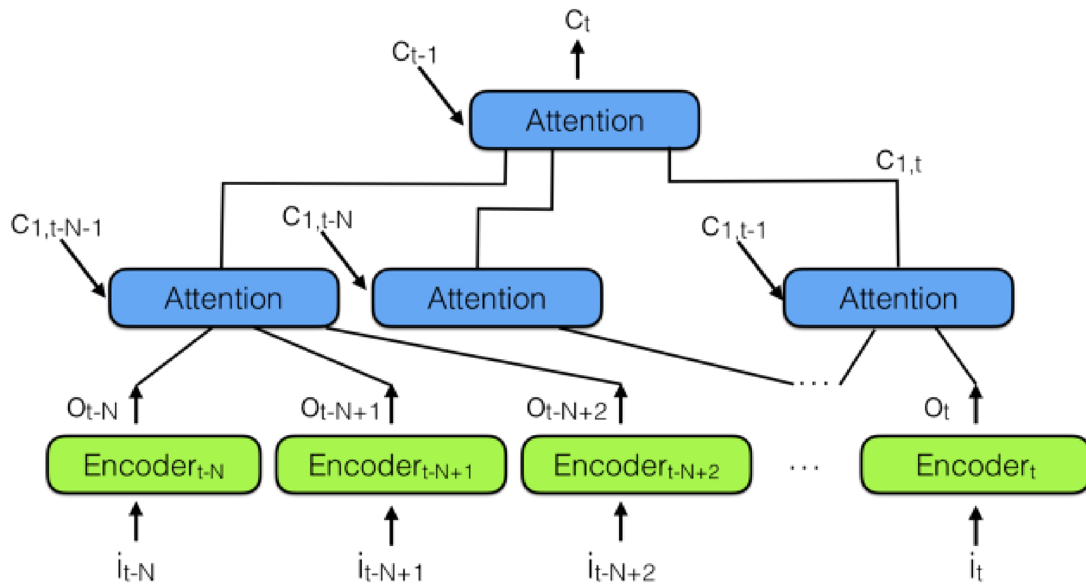


Fig. 27. Codificador de atención neuronal jerárquico [21].

Pero lo más importante es observar la longitud de la ruta necesaria para propagar un vector de representación a la salida de la red: en las redes jerárquicas es proporcional a un orden logarítmico *[21]* (log N) donde N es el número de capas jerárquicas. Esto contrasta con los pasos (T) que debe realizar un red neuronal recurrente (RNN), donde T es la longitud máxima de la secuencia a recordar y T >> N.

Acerca del entrenamiento de RNN / LSTM: RNN y LSTM son difíciles de entrenar porque requieren grandes capacidad de memoria y limita la aplicabilidad de las soluciones de redes neuronales. Durante el trabajo hemos lidiado con este problema limitando el tamaño de *embedding* a un número de palabras máximo que pudiera tratar adecuadamente cada red neuronal sin llegar a un desbordamiento de la memoria del ordenador.

Las redes RNN, LSTM o GRU procesan la secuencia de entrada secuencialmente palabra a palabra, lo que significa que no pueden realizar el cálculo para el paso de tiempo *t* hasta que se haya completado el cálculo para el paso de tiempo *t-1*. Esto ralentiza el entrenamiento y la inferencia.

Los experimentos de comportamiento sugieren que los seres humanos y algunos animales emplean esta estrategia de dividir las secuencias cognitivas o de comportamiento en fragmentos en una amplia variedad de tareas. En cualquier caso, parece que recordamos aquellas cosas sobre las que se aplicó un proceso de atención por lo que este mecanismo parece más cercano a la forma en la que se captura dicha información y como se selecciona para su almacenamiento.

3.2 Redes neuronales Transformer. *Self-attention.*

Las redes *Transformer* mediante el mecanismo de *self-attention* superan las limitaciones mostradas por las redes RNN (ver *3.1.4*), siendo capaces de procesar las palabras de la secuencia en paralelo acelerando su cálculo. Además, la distancia entre palabras en la secuencia del proceso de análisis ya no importa dado que la red es óptima encontrando relaciones entre palabras adyacentes y palabras separadas.

3.2.1 Arquitectura Transformer

La arquitectura *Transformer* es adecuada para el procesamiento de texto que son inherentemente secuenciales. Toman una secuencia de texto como entrada y producen otra secuencia de texto como salida.

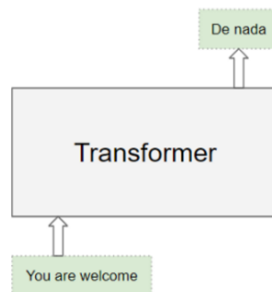


Fig. 28. Ejemplo de Transformer como caja negra. Ejemplo de traducción [6].

La arquitectura *Transformer* contiene una pila de codificadores (*encoder*) y otra pila de decodificadores (*decoder).* Ambas pilas tienen sus capas de incrustación (*embedding*) para sus respectivas entradas. También el sistema dispone de una capa de salida para generar la salida final.

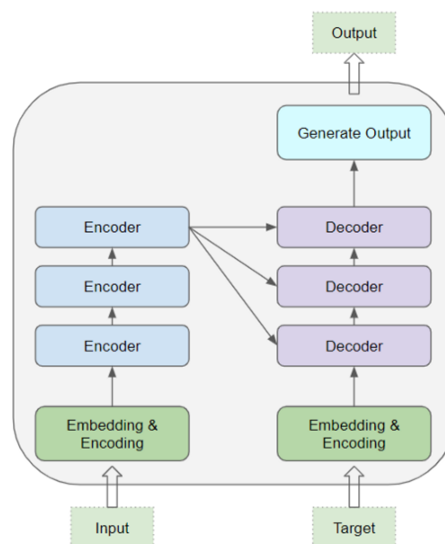


Fig. 29. Capas de los codificadores, decodificadores, incluyendo las capas embedded y a la entrada y capa de salida [6].

Todos los codificadores son idénticos entre sí. Del mismo modo, todos los decodificadores también son idénticos.

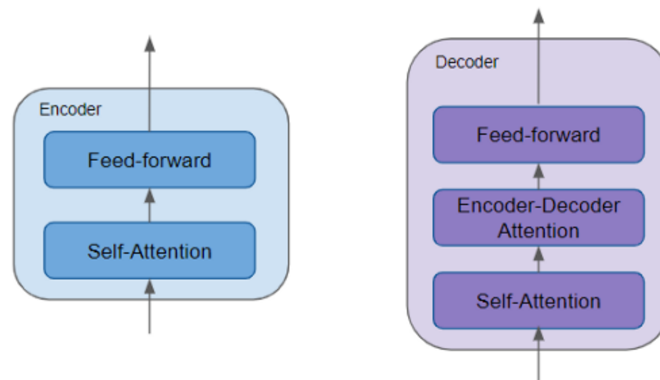


Fig. 30. Estructura interna del encoder y decoder [6].

El codificador (*encoder*) contiene la capa *self-attention* que calcula la relación entre diferentes palabras de la secuencia, así como una capa de retroalimentación. El mecanismo matemático que se utiliza para calcular la distancia entre las palabras es el producto escalar partiendo de que cada palabra viene representada por un vector que se origina en la capa de *embedding*.

El decodificador (*decoder*) contiene la capa *self-attention* y la capa de retroalimentación y en medio una segunda capa de atención compartida entre el codificador y el decodificador.

3.2.1.1 Capa de atención.

Al procesar una palabra, el sistema *Attention* permite que el modelo se centre en otras palabras de la entrada que están estrechamente relacionadas con esa palabra.

Fig. 31. Ejemplo de secuencia mostrando que el mecanismo de atención permite que dos frases casi iguales sepan por el contexto que el término "it" hace referencia a "cat" o "milk".

Además, estamos buscando de algún modo inteligencia en este método, por ejemplo, en las frases: “*the cat drank the milk because it was hungry*” o ”*the cat drank the milk because it was sweet*”, el objetivo es que el proceso de atención de la palabra “*it*” permita distinguir que en la primera frase hace referencia a la palabra “*cat*” y en la segunda a la palabra “*milk*”.

Es necesario notar que todo se basa en el primer proceso del codificador llamada “*Embedding* & *Encoding*” que asigna un vector a cada palabra. Esto vector n-dimensional establece una distancia entre las palabras agrupándolas por su sentido semántico entre otras propiedades. Esto es importante, para entender que la relación entre las palabras de una frase la establecerá la distancia entre las mismas y, por tanto, en parte, por su valor semántico.

3.2.2 Flujo de información y formulación

En este apartado vamos a ver la diferencia en el modelo mientras se están entrenado respecto a la fase de predicción. También se hará hincapié en cómo afecta este proceso, llamado *Teacher Forcing* en la mejora del aprendizaje y la posibilidad del entrenamiento en paralelo.

3.2.2.1 Fase de entrenamiento respecto a la fase de predicción.

Se establece un flujo de entrada y otro de salida. Para un ejemplo de traducción el flujo de entrada sería la secuencia de palabras a traducir y el fujo de salida la secuencia de palabras traducidas.

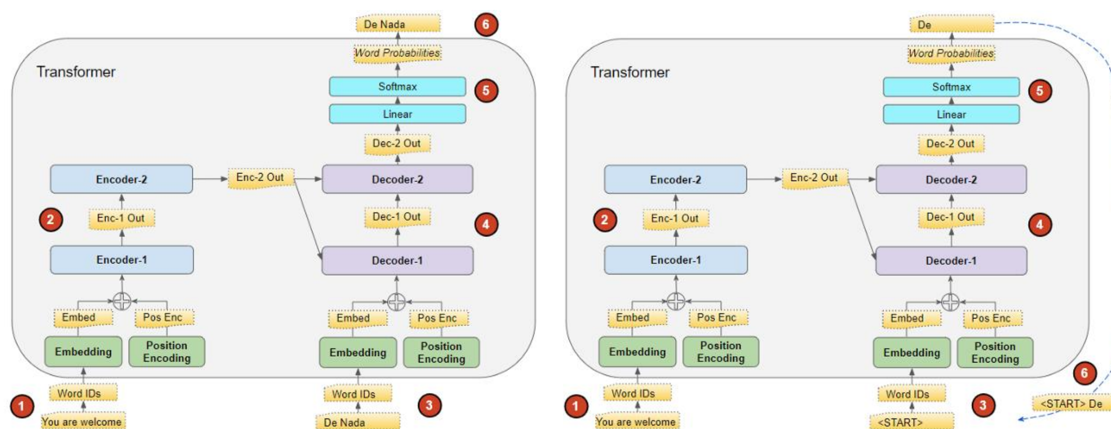


Fig. 32. Flujo de entrada y salida en el proceso de entrenamiento (izquierda) respecto al proceso de predicción (derecha).

El transformador procesa los datos de esta manera *[32]*:

1) La secuencia de entrada se convierte en *embeddings* (vector de *embedding* y codificación de la posición) y se envía hacia el codificador. En el proceso de entrenamiento esto puede ser realizado en paralelo por todas las palabras.

2) La pila de codificadores procesa los vectores de las palabras junto a la codificación de la posición (necesario para devolver su posición original

durante el entrenamiento cuando las palabras son procesadas en paralelo) y produce una representación codificada de la secuencia de entrada.

3) La secuencia de destino origina con un token de inicio de oración ([CLS]), se convierte en *embeddings* (vector de *embeddings* con codificación de posición) y se envía hacia el decodificador.

4) La pila de decodificadores procesa esto junto con la representación codificada de la pila del codificador para producir una representación codificada de la secuencia objetivo.

5) La capa de salida la convierte en probabilidades de palabras y la secuencia de salida final.

6) Según sea entrenamiento o predicción

   a) Entrenamiento: la función de pérdida del transformador compara esta secuencia de salida con la secuencia objetivo de los datos de entrenamiento. Esta función de pérdida se utiliza para generar gradientes con el objetivo de entrenar al *Transformer* durante la propagación hacia atrás (*backpropagation*). Se entrena hasta conseguir la precisión detectada y finaliza en este paso.

   b) Predicción: se toma la última palabra de la secuencia de salida como la palabra predicha. Esa palabra ahora se completa en la segunda posición de nuestra secuencia de entrada del decodificador, que ahora contiene un símbolo de inicio de oración y la primera palabra ya decodificada.

7) Sólo para la predicción. Seguir por el paso 3) introduciendo la nueva secuencia del decodificador en el modelo. Después, procesar la segunda palabra de la salida y añadirla a la secuencia del decodificador. Repetir hasta recibir un token de final de oración ([SEP]).

3.2.2.2 Teacher Forcing

El enfoque de alimentar la secuencia de destino al decodificador durante el entrenamiento se conoce como *Teacher Forcing*.

Durante el entrenamiento, se podría usar el mismo enfoque que se usa durante la inferencia. En otras palabras, ejecutar el transformador en un bucle, tomar la última palabra de la secuencia de salida, añadirla a la entrada del decodificador y enviarlo al decodificador para la siguiente iteración. Cuando se predijera el token de fin de oración, la función de pérdida compararía la secuencia de salida generada con la secuencia objetivo para entrenar la red.

Este bucle dificultaría el entrenamiento del modelo, además de ralentizarlo. El modelo tendría que predecir la segunda palabra basándose en una primera palabra predicha potencialmente errónea, y así sucesivamente.

En cambio, al alimentar la secuencia de destino al decodificador, le estamos dando una pista. Aunque predijo una primera palabra errónea, en su lugar puede

usar la primera palabra correcta para predecir la segunda palabra para que esos errores no sigan acumulándose.

Esto permite que la arquitectura *Transformer* pueda generar todas las palabras en paralelo durante el entrenamiento sin bucles, por lo que el proceso se acelera.

3.2.3 Atención en nuestro escenario. Clasificación

Estamos ante un problema de clasificación (*URL Malicious detection*), por ello, debemos adaptar las capas del modelo *Transformer* para añadir una o varias capas de clasificación a continuación del modelo *Transformer*. En este caso, deberá descargar el contenido de la página web (sin ejecutarla) y mediante una capa *Classification Head,* tomando la salida del transformador, es capaz de realizar predicciones y detectar si la *URL* es o no maliciosa.

Fig. 33. Revisión de la arquitectura Transformer para un problema de clasificación como URL Malicious.

Hay diferentes enfoques para utilizar una arquitectura *Transformer* para nuestro escenario:

- Uso del *tokenizer* de *BERT*. El *tokenizer* de Bert, se diferencia de un *tokenizer* realizado manualmente sólo con la información de entrenamiento en que, no pueden aprovecharse del conocimiento inherente en el modelo BERT. Las palabras codificadas desde BERT tendrán la posibilidad mediante capa de *Embedding* la posibilidad de tener un tensor (vector) asociado (*embedding*).

  Se muestra el detalle del resultado de este modelo en *8*. Se ha obtenido un valor de un 99.08% aplicando esta técnica, aunque con algo de sobreentrenamiento (overfitting).

- *Pre-training*: uso de la capa de *embedding* de *BERT [45]*. Una vez usado el *tokenizer* querremos que la primera capa de *Embedding* se nutra del conocimiento que proporciona por cada palabra el modelo *BERT [61]*. Hay que tener en cuenta que *BERT* ha sido entrenado con millones de palabras (con fuentes de entrada fiables como artículos de la Wikipedia y otros) que nos permiten garantizar que los vectores van a indicarnos un gran valor sobre la temática, género, número y otras características adquiridas por aprendizaje de cada palabra.

  Se muestra el detalle del resultado de este modelo en *8*. Se ha obtenido un valor de un 94.77% aplicando esta técnica sobre los datos de verificación.

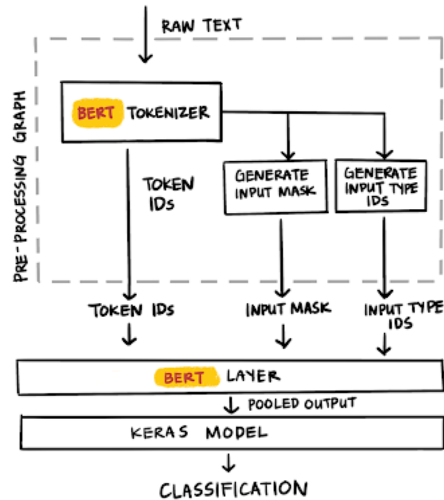


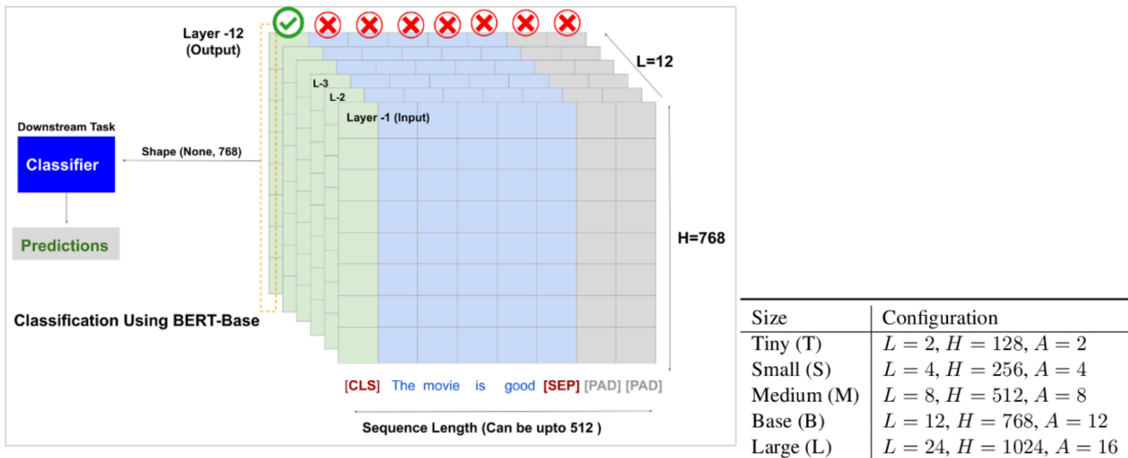
Fig. 34. Pre-entrenamiento y ajuste fino en un problema de clasificación [20].

- *Fine tuning [1]*: uso de la capa de *embedding* de *BERT* de forma que posteriormente pueda ser entrenable. En este caso, además de la capa de pre-training, se permite que sea el usuario el que afine algunos parámetros de la red en la fase de entrenamiento.
- *Knowledge Distillation [4]* en *BERT*: *Knowledge distillation* es el proceso de transferencia de conocimiento de un modelo grande a uno más pequeño (ver *3.2.3.4*). Se genera una transferencia de conocimiento desde el modelo de *BERT* a una red estudiante (la nuestra) que la generamos con el fin de luego ser entrenada en un proceso de clasificación.

3.2.3.1 Uso del *tokenizer* de BERT

El *tokenizer* de *BERT* (*FullTokenizer*), convierte el texto en *tokens* asociando información adicional como el tamaño del vocabulario o para permitir convertir las palabras a minúsculas *[45]*.

Para ello, se genera una capa de *BERT* que usa la librería *hub* construyendo una capa de Keras (KerasLayer). Usamos un modelo pre-entrenado en *BERT* que dispone de 12 *encoders (L-12)*, 768 dimensiones en capa oculta (*H-768*) y 12 cabezas de multidetección (*A-12*). Este modelo corresponde con el denominado como *Base* dentro de las diferentes configuraciones de *BERT*. Hágase notar (ver *Fig. 35*) que existe un modelo más potente denominado *Large*.



Size	Configuration
Tiny (T)	$L = 2, H = 128, A = 2$
Small (S)	$L = 4, H = 256, A = 4$
Medium (M)	$L = 8, H = 512, A = 8$
Base (B)	$L = 12, H = 768, A = 12$
Large (L)	$L = 24, H = 1024, A = 16$

Fig. 35. Modelos de BERT. L (capas), H (dimensiones capa oculta), A (head multi-detection) [9].

Esta capa usa el *tokenizer* de *BERT,* pero no la capa de *embedding*. Esto hará que no se saque todo el partido al modelo ya que lo más relevante del modelo es obtener un vector de *embedding* de cada palabra con el conocimiento pre-entrenado que nos proporcionará el modelo de *BERT* sobre la aplicación de cada una de nuestras palabras.

3.2.3.2 *Pre-training*: uso de la capa de *embedding* de *BERT*

Además de usar el *tokenizer* de BERT, para aprovechar el gran conocimiento inherente en el modelo, es necesario no sólo usar el *tokenizer* sino utilizar la capa de *embedding*. Esto es debido a que asociado a cada identificador de la palabra que nos proporciona el *tokenizer* también, el modelo, nos proporciona un vector. Este vector ha sido entrenado con millones de palabras y en diferentes contextos lo que le ha permitido a *BERT* proporcionarnos un vector de cada palabra que es genérico y nos permite ser aplicado a múltiples contextos.

Esto diferencia a este modelo del modelo anterior en el que hemos entrenado una capa de *embedding* sólo con el vocabulario presente en nuestro conjunto de datos.

Utilizando la capa de *embedding* de *BERT* se ha obtenido ausencia de sobreentrenamiento (*overfitting*) en nuestro modelo. De hecho, en bastantes casos se ha obtenido mejor resultado sobre los datos de test o validación que los obtenidos durante el entrenamiento. Sin embargo, en el punto anterior ocurría lo contrario, teniendo que parar el entrenamiento muy pronto para no caer en sobreentrenamiento (*overfitting*).

Word Embeddings

"Word Embedding" = Feature Vector representation of a word

e.g. 300

<0.4125, -1.6098, 0.6047, ..., -1.4257, -1.2321>

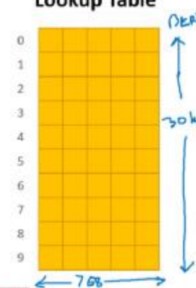


Fig. 36. La capa de embedding de BERT facilita un vector de 768 dimensiones por cada palabra de un diccionario total de 30.000 palabras [24].

3.2.3.3 *Fine tuning*: uso de la capa de *embedding* de *BERT* para ser entrenado.

BERT permite realizar un ajuste fino llamado *fine tuning* sobre algunos parámetros del modelo de la red. En este caso, se ha indicado en la creación de la capa *BERT* la precarga de los pesos de la red base comentada anteriormente, para que esta capa pueda ser entrenada. Sin embargo, en el apartado anterior se precargaba los pesos de la red entrenada pero no se permitía el ajuste fino de ninguno de sus parámetros.

Esto suele redundar en una mayor calidad del modelo, aunque el entrenamiento es más exigente, tardando más tiempo en ser entrenado y aumentando el riesgo de sobreajuste (*overfitting*) obligando incluso a aumentar el conjunto de datos.

El ajuste fino modifica los parámetros del modelo a entrenar, aunque no modifica las transformaciones, como la creación de tokens de entrada del texto y la asignación de tokens a sus entradas correspondientes en una matriz de *embedding*.

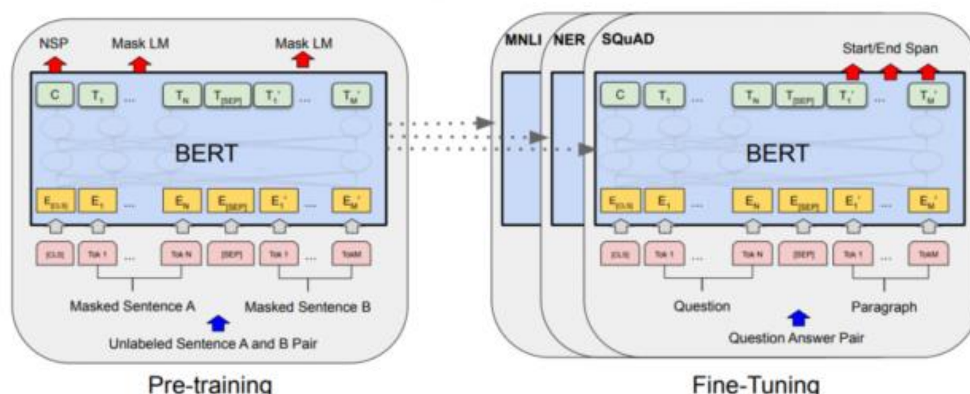


Fig. 37. Diferencia entre pre-training y Fine tuning [8].

Es muy habitual que una vez realizado el *fine tuning*, que es un proceso largo y costoso, se desee guardar el modelo ajustado para que no haya que entrenarlo cada vez. Los parámetros más habituales suelen ser los regularizadores (ver *3.1.3.3*) que permiten converger los resultados de entrenamiento y validación de una forma más eficiente en un número de épicas adecuado.

3.2.3.4 Modelo estudiante basado en *Knowledge Distillation* de *BERT* aplicado a un problema de clasificación

Knowledge distillation es el proceso de transferencia de conocimiento de un modelo grande a uno más pequeño *[4]*. Concretamente en nuestro caso se plantea una transferencia de conocimiento desde el modelo de *BERT* a una red estudiante que se genera con el fin de ser entrenada posteriormente en un proceso de clasificación.

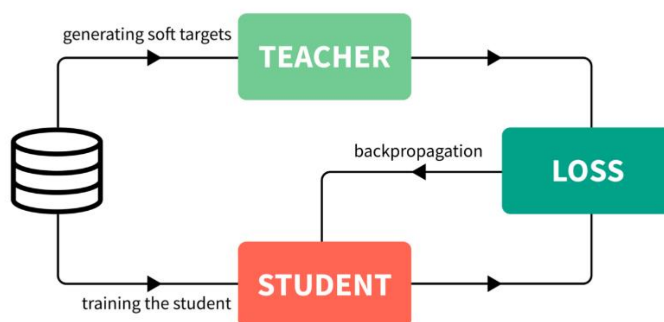


Fig. 38. Modelo teacher-student para transferencia de aprendizaje entre dos redes neuronales [22].

Knowledge Distillation es un método de *transfer learning* (ver *2.4.9.3*) que usa modelos pre-entrenados de BERT que facilita ser puesto en producción respecto a la alternativa de descargar el modelo de BERT completo previo a entrenar el modelo de clasificación.

Este es un modelo de transferencia de conocimientos de alumno a profesor. En esta técnica, se entrena un modelo o conjunto de modelos más grande y se crea un modelo más pequeño para imitar a ese modelo más grande. El proceso de *Distillation [4]* se refiere a copiar el conocimiento oscuro, el concepto es similar al suavizado de etiquetas durante la regularización *[53]*, evitando que el modelo esté demasiado seguro sobre su predicción.

La arquitectura de estudiante ofrecida por *DistilBERT* es la misma que *BERT*, excepto por la eliminación de *embedding* de tipo *token* y el agrupador (*pooler*), al tiempo que se reduce el número de capas en un factor de 2, lo que afecta en gran medida a la eficiencia del cálculo.

La inicialización de la red estudiante debe ser realizada en el momento adecuado para conseguir la convergencia durante el entrenamiento del modelo.

DistilBERT ofrece resultados extraordinarios en la tarea de clasificación que nos ocupa disminuyendo sólo un 0,6% en la precisión respecto a *BERT*, mientras que el modelo es un 40% más pequeño y un 60% más rápido.

Los pasos para aplicar *DistilBERT* son los siguientes:

- <u>Comprobación de la configuración</u>: donde se indica cual es el modelo estudiante y se referencia el modelo profesor.

- <u>*Tokenizer*</u>: es similar al *tokenizer* de BERT.

- <u>Modelo</u>: en este apartado se carga el modelo pre-entrenado con las entradas salidas incluidas.

- <u>Modelado para clasificación en *DistilBERT*</u>: se define la pérdida de la clasificación y la puntuación de clasificación antes de aplicar la función *Softmax*.

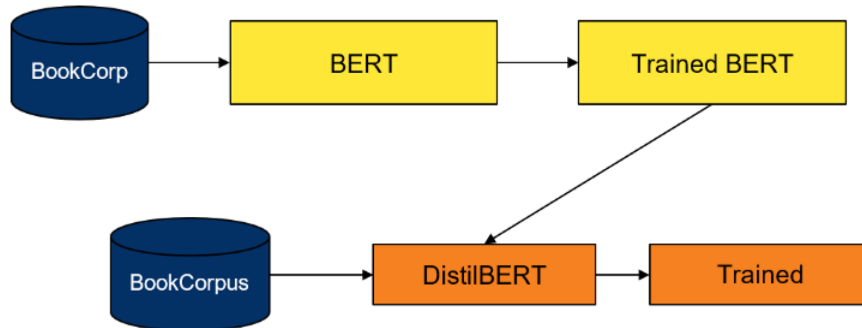


Fig. 39. BERT y DistilBERT [4].

3.2.3.5 Métodos para comprimir *BERT*

La compresión del modelo BERT reduce la redundancia en una red neuronal entrenada. Esto es útil, ya que *BERT* apenas ocupa una GPU (*BERT-Large* necesita al menos dos) y, por ejemplo, no puede ejecutarse en un *smartphone*. La mejora en la eficiencia de la memoria y velocidad de inferencia también ahorra costes. A continuación, se indican algunos método utilizados para dicha compresión:

1. <u>Poda</u> (*pruning*): se eliminan partes innecesarias de la red después del entrenamiento. Esto incluye la poda de matrices de pesos, poda de la cabeza de atención (A), eliminación de capas y otros. Algunos métodos también imponen la regularización durante el entrenamiento para aumentar la capacidad de poda (abandono de la capa).

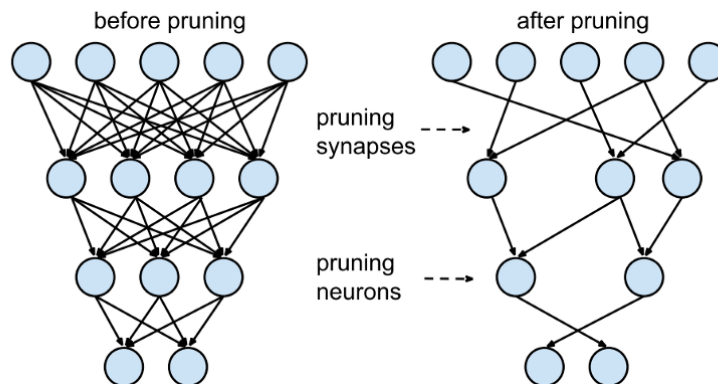


Fig. 40. Red neuronal antes y después de un proceso de poda (pruning) [47].

2. Factorización de peso: aproxima matrices de parámetros factorizándolas en una multiplicación de dos matrices más pequeñas. La factorización de peso se puede aplicar tanto a la capa de *embedding* de tokens (lo que ahorra mucha memoria) como a parámetros en capas de *feed-forward* o *self-attention* (para conseguir mejoras en la velocidad de cálculo).

$$\underset{m \times n}{A} = \underset{m \times m}{Q} \; \underset{m \times n}{\begin{bmatrix} R \\ 0 \end{bmatrix}}$$

Fig. 41. Factorización de matrices por vocabulario / parámetros [47].

3. *Knowledge Distillation*: aplica el método "Maestro-Estudiante" entrenando previamente un modelo de representación de lenguaje de propósito general más pequeño, *DistilBERT*, que luego será ajustado para nuestra tarea de clasificación. Se aplica la K*nowledge distillation* durante la fase de preentrenamiento y reduciendo el tamaño de un modelo *BERT* en un 40%, mientras se mantiene el 97% de efectividad además de ser un 60% más rápido (ver *3.2.3.4*).

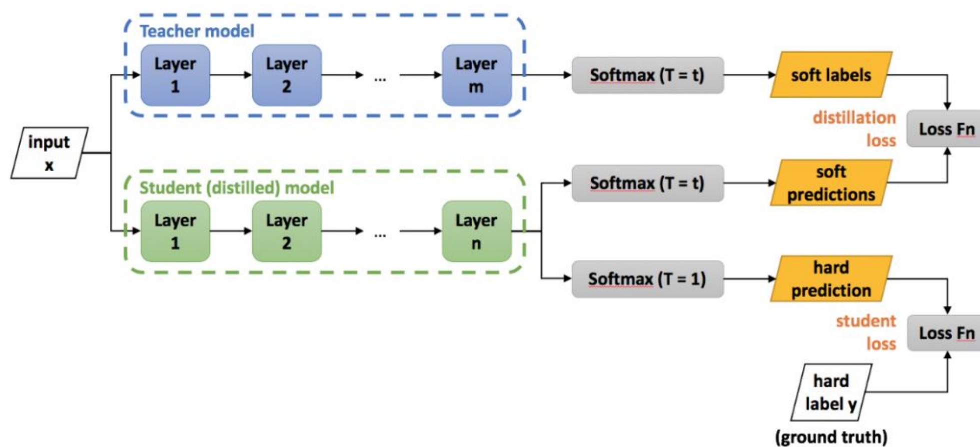


Fig. 42. Modelo teacher-student para transferencia de aprendizaje en BERT [4].

4. Parámetro compartido: los pesos del modelo comparten el mismo valor que otros parámetros del modelo entre capas. Por ejemplo, *ALBERT* (*A Lite BERT*) utiliza las mismas matrices de peso para cada capa de *self-attention* que en las capas de *BERT*.

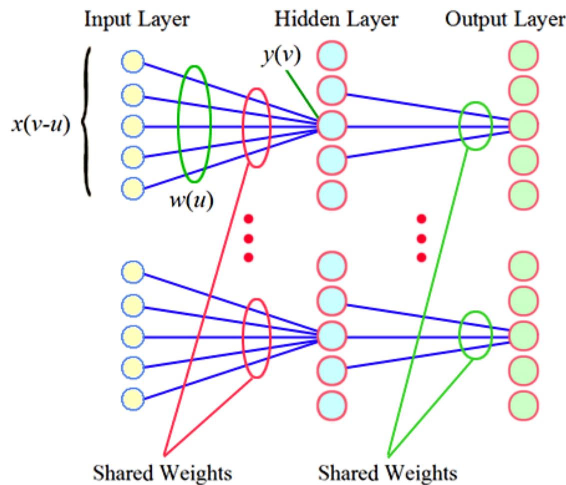


Fig. 43. Pesos compartidos entre capas [47].

5. <u>Cuantización</u>: ejecutar números de coma flotante para usar sólo unos pocos bits (típicamente int8, lo que causa un error de redondeo). Los valores de cuantización se pueden aprender durante o después del entrenamiento. Esta técnica sólo pierde un 1% en la precisión y suele mejorar cerca de un 25% en la eficiencia.

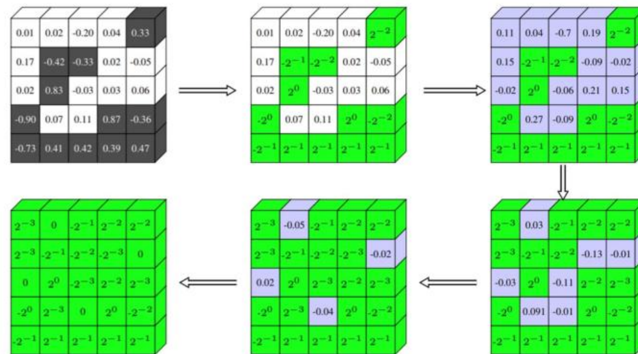


Fig. 44. Ejemplo de cuantización de float a int8 [3].

6. <u>*Only pre-train vs Downstream after pre-train*</u>: Primero el modelo es entrenado en las tareas de pre-entrenamiento. Una vez que se completa el entrenamiento previo, el mismo modelo se puede ajustar (*fine tuning*) para una variedad de tareas posteriores (*downstream*). Un modelo pre-entrenado de BERT está ajustado para una tarea descendente específica. Por lo tanto, los modelos individuales previamente entrenados pueden generar múltiples modelos específicos de tareas posteriores después del ajuste fino. Un ejemplo de tareas sería el de clasificación que aplica a este trabajo.

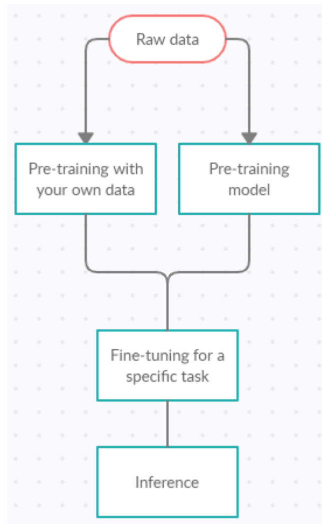


Fig. 45. Procesos completo en BERT para diferentes tareas.

4 Muestra de resultados

A continuación, se va a indicar los resultados obtenidos con las diferentes redes y con la selección de los mejores hiperparámetros para cada una de ella.

- <u>Redes neuronales profundas (*DNN*)</u>:
  - **DNN 2 capas ocultas** (ver *8,* TFM_PEC2_2021_ofrances_modelling_v1.html).
    - Tratamiento datos:
      - Desbalanceo datos: sí. No se tratan previamente.
      - Se tratan los metadatos para extraer características "inteligentes".
    - Modelo:

```
Layer (type)                 Output Shape              Param #
=================================================================
dense_12 (Dense)             (None, 350)               251650
_________________________________________________________________
dense_13 (Dense)             (None, 350)               122850
_________________________________________________________________
dense_14 (Dense)             (None, 1)                 351
=================================================================
Total params: 374,851
Trainable params: 374,851
Non-trainable params: 0
_________________________________________________________________
```

    - Mejores hiperparámetros:
      - `batch_size = 10, nb_epoch = 10`
      - `optimizer="adam",loss="binary_crossentropy"`
    - Resultados:
      - <u>Precisión sobre validación:</u> **99.895%**.
      - <u>*Overfitting*</u>: no.
- <u>Redes neuronales recurrentes (RNN)</u>:
  - **RNN Masking** (ver *8,* TFM_PEC3_2021_ofrances_RNN_Masking_LSTM_content__v1.html).
    - Tratamiento datos:
      - Desbalanceo datos: No. Técnica *undersampling*.
      - No se tratan los datos. Se analiza el 'content' sin procesar
    - Modelo:

```
Layer (type)                 Output Shape              Param #
=================================================================
masking (Masking)            (None, 64, 1)             0
_________________________________________________________________
spatial_dropout1d (SpatialDr (None, 64, 1)             0
_________________________________________________________________
lstm (LSTM)                  (None, 16)                1152
_________________________________________________________________
dense (Dense)                (None, 2)                 34
=================================================================
```

```
Total params: 1,186
Trainable params: 1,186
Non-trainable params: 0
_________________________________________________________________
```

- Mejores hiperparámetros:
  - `batch_size=96, nb_epoch=2, embed_size=128, hide_cells=16, dropout_rate=0.5`
  - `optimizer="adam",loss="categorical_crossentropy"`
- Resultados:
  - Precisión sobre validación: **73.22%**.
  - *Overfitting*: no.

○ ***RNN Embedding*** (ver *8,* TFM_PEC3_2021_ofrances_RNN_Embedding_LSTM__content__v1.html).

- Tratamiento datos:
  - Desbalanceo datos: No. Técnica *undersampling*.
  - No se tratan los datos. Se analiza el *'content'* sin procesar
- Modelo:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, 64, 128)           6400000
_________________________________________________________________
spatial_dropout1d (SpatialDr (None, 64, 128)           0
_________________________________________________________________
lstm (LSTM)                  (None, 16)                9280
_________________________________________________________________
dense (Dense)                (None, 2)                 34
=================================================================
Total params: 6,409,314
Trainable params: 6,409,314
Non-trainable params: 0
_________________________________________________________________
```

Mejores hiperparámetros:
  - `batch_size=96, nb_epoch=2, embed_size=128, hide_cells=16, dropout_rate=0.5`
  - `optimizer="adam",loss="categorical_crossentropy"`
- Resultados:
  - Precisión sobre validación: **95.08%**.
  - *Overfitting*: no.

○ ***RNN Embedding* + CNN** (ver *8,* TFM_PEC3_2021_ofrances_RNN_Embedding_CNN_LSTM_content_v1.html).

- Tratamiento datos:
  - Desbalanceo datos: No. Técnica *undersampling*.
  - No se tratan los datos. Se analiza el 'content' sin procesar
- Modelo:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, 64, 128)           6400000
_________________________________________________________________
```

```
spatial_dropout1d (SpatialDr (None, 64, 128)          0
_________________________________________________________________
conv1d (Conv1D)              (None, 64, 256)          164096
_________________________________________________________________
max_pooling1d (MaxPooling1D) (None, 16, 256)          0
_________________________________________________________________
lstm (LSTM)                  (None, 16)               17472
_________________________________________________________________
dense (Dense)                (None, 2)                34
=================================================================
Total params: 6,581,602
Trainable params: 6,581,602
Non-trainable params: 0
=================================================================
```

- Mejores hiperparámetros:
  - `batch_size=96, nb_epoch=2, embed_size=128, hide_cells=16, dropout_rate=0.5`
  - `optimizer="adam",loss="categorical_crossentropy"`
- Resultados:
  - Precisión sobre validación: **95.08%**. **Nota**: subiendo el hiperparámetro `embed_size=1024` se ha conseguido un **97,39%** (mejor resultado *RNN*).
  - *Overfitting*: no.

○ ***RNN Embedding*** **+ bi-LSTM** (ver *8,* TFM_PEC3_2021_ofrances_RNN_ _Embedding_CNN_LSTM_content_v1.html).

- Tratamiento datos:
  - Desbalanceo datos: No. Técnica *undersampling*.
  - No se tratan los datos. Se analiza el 'content' sin procesar
- Modelo:

```
_________________________________________________________________
Layer (type)                 Output Shape             Param #
=================================================================
embedding (Embedding)        (None, 64, 128)          6400000
_________________________________________________________________
spatial_dropout1d (SpatialDr (None, 64, 128)          0
_________________________________________________________________
bidirectional (Bidirectional (None, 32)               18560
_________________________________________________________________
dense (Dense)                (None, 2)                66
=================================================================
Total params: 6,418,626
Trainable params: 6,418,626
Non-trainable params: 0
=================================================================
```

- Mejores hiperparámetros:
  - `batch_size=96, nb_epoch=2, embed_size=128, hide_cells=16, dropout_rate=0.5`
  - `optimizer="adam",loss="categorical_crossentropy"`
- Resultados:
  - Precisión sobre validación: **95.08%**.
  - *Overfitting*: no.

- ***RNN Embedding* + GRU** (ver *8,* TFM_PEC3_2021_ofrances_RNN_Embedding_ GRU_content_v1.html).
  - Tratamiento datos:
    - Desbalanceo datos: No. Técnica *undersampling*.
    - No se tratan los datos. Se analiza el *'content'* sin procesar
  - Modelo:

Layer (type)	Output Shape	Param
embedding (Embedding)	(None, 64, 128)	6400000
spatial_dropout1d (SpatialDr	(None, 64, 128)	0
gru (GRU)	(None, 16)	7008
dense (Dense)	(None, 2)	34

```
Total params: 6,407,042
Trainable params: 6,407,042
Non-trainable params: 0
```

  - Mejores hiperparámetros:
    - `batch_size=96, nb_epoch=2, embed_size=128, hide_cells=16, dropout_rate=0.5`
    - `optimizer="adam",loss="categorical_crossentropy"`
  - Resultados:
    - Precisión sobre validación: **95.08%%**.
    - *Overfitting*: no.

- Redes neuronales basadas en BERT:
  - ***BERT Tokenizer* + *self-embedding*** (ver *8,* TFM_PEC3_2021_ofrances_BERT_content_v1.html).
    - Tratamiento datos:
      - Desbalanceo datos: No. Técnica *undersampling*.
      - No se tratan los datos. Se analiza el 'content' sin procesar
    - Modelo:

Layer (type)	Output Shape	Param #
keras_layer_1 (KerasLayer)	multiple	109482241
conv1d (Conv1D)	multiple	196736
conv1d_1 (Conv1D)	multiple	295040
conv1d_2 (Conv1D)	multiple	393344
global_max_pooling1d (Global	multiple	0
dense (Dense)	multiple	98560
dropout (Dropout)	multiple	0

```
dense_1 (Dense)              multiple                  257
=================================================================
Total params: 110,466,178
Trainable params: 983,937
Non-trainable params: 109,482,241
```

- Mejores hiperparámetros:
  - `batch_size=16, nb_epoch=2, embed_size=128, hide_cells=256, dropout_rate=0.4`
  - `optimizer="adam",loss="binary_crossentropy", learning_rate=2e-5`
- Resultados:
  - Precisión sobre validación: **94.77%**.
  - *Overfitting*: no.

○ ***BERT Tokenizer + BERT embedding*** (ver *8*, TFM_PEC3_2021_ofrances_BERT_content_v1.html).

- Tratamiento datos:
  - Desbalanceo datos: No. Técnica *undersampling*.
  - No se tratan los datos. Se analiza el 'content' sin procesar
- Modelo:

```
Layer (type)                 Output Shape              Param #
=================================================================
keras_layer_1 (KerasLayer)   multiple                109482241
_________________________________________________________________
conv1d (Conv1D)              multiple                  196736
_________________________________________________________________
conv1d_1 (Conv1D)            multiple                  295040
_________________________________________________________________
conv1d_2 (Conv1D)            multiple                  393344
_________________________________________________________________
global_max_pooling1d (Global multiple                  0
_________________________________________________________________
dense (Dense)                multiple                  98560
_________________________________________________________________
dropout (Dropout)            multiple                  0
_________________________________________________________________
dense_1 (Dense)              multiple                  257
=================================================================
Total params: 110,466,178
Trainable params: 983,937
Non-trainable params: 109,482,241
```

- Mejores hiperparámetros:
  - `batch_size=16, nb_epoch=2, embed_size=128, hide_cells=256, dropout_rate=0.4`
  - `optimizer="adam",loss="binary_crossentropy", learning_rate=2e-5`
- Resultados:
  - Precisión sobre validación: **94.77%**.
  - *Overfitting*: no.

5 Conclusiones

Durante la elaboración de este trabajo se han aplicado un número importante de técnicas para la construcción de redes neuronales como han sido *DNN*, *LSTM*, *GRU* y *BERT* (*Transformer*) para un problema de clasificación binaria, en nuestro caso, la detección de *URL* maliciosas sobre un conjunto de datos fiable publicado por el *National Center for Biotechnology Information [58]*.

Las mejores predicciones se han obtenido con la combinación de técnicas de *Machine Learning* junto a la preparación adecuada de los datos y la aplicación de redes neuronales profundas (*DNN*).

El resultado presentado por las redes neuronales profundas (*DNN*) en cuanto a predicción sobre los datos de validación ha sido excelente (99.895%) debido al tratamiento de datos previo que ha permitido dar con parámetros clave del conjunto de datos, como la geolocalización, la ofuscación del código JavaScript o el tamaño del código *JavaScript*.

Sin embargo, tanto las redes neuronales recurrentes (*RNN*) como las redes basadas en *Transformer* han tenido que lidiar directamente con el contenido en bruto de las páginas obteniendo resultados de predicción superiores al *95%* sobre los datos de validación. Sin embargo, para conseguir estos resultados ha sido necesario la aplicación de técnicas de balanceo de datos *undersampling* para conseguir, disponer de la información balanceada (ver *2.3.1*) entre páginas *URL* maliciosas (2.27%) y no maliciosas (97.73%) debido al desbalanceo existente en el conjunto de datos.

Las redes neuronales recurrentes, poco a poco, están siendo sustituidas por los *Transformer*. En el caso particular de este trabajo, con las redes *Transformer* se ha visto que los resultados obtenidos durante el entrenamiento rápidamente han sido superados al ser aplicados sobre los datos de validación. Según varios autores *[32][40]* se ha visto que esto es habitual con el modelo *BERT* debido a que el modelo al ser tan genérico y haber sido construido con millones de palabras de la *wikipedia* y otros artículos, consigue una gran generalización aplicando la técnica de *embedding [32]*. Esto nos permite decir que, aunque el mayor rendimiento se ha conseguido sobre las redes neuronales profundas tras haber procesado los datos, el método más generalizable y menos susceptible a verse afectado por cambios en el modelo son las redes basadas en *Transformer*.

Uno de los logros ha consistido en generar gran cantidad de modelos y, sobre todo, aprender cómo deben ser tratados cada uno de ellos en función de las características específicas del conjunto de datos disponible y del problema de clasificación binaria planteado.

Sobre la planificación y metodología, en esencia, se han conseguido los objetivos planteados, aunque durante la planificación se había minusvalorado el tiempo necesario para entrenar los modelos, sobre todo en redes neuronales recurrentes y *Transformer*. Ello ha llevado a la simplificación de algún modelo y no poder profundizar en la búsqueda de hiperparámetros avanzados en alguna de las redes *Transformer*. Se ha solucionado manteniendo la planificación sobre

la redacción de la memoria, pero alargando un par de semanas las pruebas a realizar sobre las redes *Transformer* respecto a lo planificado.

Sobre futuro trabajo, se aprecian las siguientes líneas de mejora (ver *2.1.4*):

1. Ampliar las técnicas "inteligentes" ya aplicadas (ver *2.4.9.3*) sobre los datos previo al entreno de la red neuronal profunda (*DNN*) aplicando técnicas de *Transfer Learning*. Entre ellas se aplicaría el análisis de sentimiento, detección de palabras malsonantes (*profanity check*), detección de errores ortográficos (*spell check*) o detección de publicidad subyacente (*advertisement detection*).

2. Implementación de un servicio web en la red para poder facilitar la consulta de una *URL* con el objetivo de saber que esta *URL* es maliciosa o no (ver *2.1.3.2*). Esto nos obligaría a mejorar algunos aspectos actuales:

   - Mejora en la velocidad de aplicación de las técnicas "inteligentes" sobre los datos. El objetivo es poder retornar si la *URL* es maliciosa o no en milisegundos tras la petición HTTPS.

   - Construcción de un entorno de preproducción para poder entrenar la red neuronal con datos actualizados y la capacidad de poder restaurar los pesos sobre el entorno de producción sin afectar al servicio.

   - Distribuir el servicio en alta disponibilidad preferiblemente en la nube para poder incrementar la disponibilidad manteniendo la velocidad de respuesta, aunque hubiera un número alto de usuarios.

3. Combinar las técnicas aplicadas en la red neuronal profunda (*ver 2.4*), incluyendo el tratamiento de los datos con el procesamiento del contenido de las páginas web tratados en los modelos *Transformer* (ver *3.2.3*).

4. Utilización de redes *DistilBERT* (ver *3.2.3.4*), para optimizar los recursos necesarios y ser utilizadas en máquinas con menos recursos que los necesarios en el modelo de *BERT Base*.

5. Uso de redes pre-entrenadas con *URL Malicious detection* en apps de smartphones mediante algún plugin disponible en los navegadores que verifiquen antes de permitir acceder una *URL* que esta no es maliciosa.

6. Integración del servicio de verificación de *URL* maliciosa en algún *proxy open source* que permita la verificación previa de *URL Malicious detection* para prevenir el acceso a *URL* maliciosas en empresas.

6 Glosario

ACRÓNIMOS	
Siglas	**Significado**
AAAI	*American Association for Artificial Intelligence*
ACM	*Association for Computing Machinery*
ANN	*Artificial Neural Network*
ARES	*Availability, Reliability and Security*
BERT	*Bidirectional Encoder Representations from Transformers*
CNN	*Convolutional Neural Network*
CNSM	*Central Nervous System Monitoring*
CW	*Confidence-Weighted Linear Classification*
DNN	*Deep Neural Network*
DNS	*Domain Name System*
ELM	*Extreme Learning Machine*
GED	*Grammar Error Detection*
HTML	*Hypertext Markup Language*
IA	Inteligencia Artificial
ICC	*International Conference Communications*
ICMLC	*International Conference on Machine Learning and Cybernetics*
IEEE	*Institute of Electrical and Electronics Engineers*
INCIBE	Instituto Nacional Ciberseguridad
IP	*Internet Protocol*
LDA	*Latent Dirichlet Allocation*
LSTM	*Long Short-Term Memory*
ML	*Machine Learning*
NLP	*Natural Language Processing*
OSI	*Open Systems Interconnection*
PA	*Passive-Aggressive (algorithms)*
PEC	Práctica Evaluación Continua
PRA	*Probabilistic Risk Assessment*

RELU	*Rectified Linear Activation Unit*
RNN	*Recurrent Neural Network*
SIGKDD	*Special Interest Group on Knowledge Discovery and Data Mining*
SP	*Security and Privacy*
SVM	*Support Vector Machines*
TFM	Trabajo Fin de Máster
TIST	*Transactions on Intelligent Systems and Technology*
URL	*Uniform Resource Locator*

DEFINICIONES

Definición	Significado
en: *Artificial Neural Network* (*ANN*) es: Red Neuronal Artificial (*RNA*)	Consiste en un conjunto de unidades, llamadas neuronas artificiales, conectadas entre sí para transmitirse señales. La información de entrada atraviesa la red neuronal (donde se somete a diversas operaciones) produciendo unos valores de salida.
en: *Convolutional Neural Network* (C*NN*) es: Red Neuronal Convolucional (*RNC*)	Consiste en un tipo de Red Neuronal Artificial con aprendizaje supervisado que procesa sus capas imitando al cortex visual del ojo humano para identificar distintas características en las entradas que hacen que pueda identificar objetos. Dispone de varias capas ocultas especializadas y con una jerarquía: por ejemplo, las primeras capas pueden detectar líneas, curvas y se van especializando hasta llegar a capas más profundas que reconocen formas complejas como un rostro.
en: *Deep Neural Network* (D*NN*) es: Red Neuronal Profunda	Consiste en un conjunto de unidades, llamadas neuronas artificiales, conectadas entre sí para transmitirse señales. A diferencia de las ANN hay más de una capa oculta con al menos una célula por cada capa.
en: *Gate Recurrent Unit* (*GRU*) es: Unidad de puertas recurrentes	GRU mecanismo de compuerta en redes neuronales recurrentes, es como una memoria a corto plazo (LSTM) con una puerta de olvido, pero tiene menos parámetros que LSTM, ya que carece de una puerta de salida.
en: *Long short-term memory* (*LSTM*) es: Memoria de largo a corto plazo	LSTM tiene conexiones de retroalimentación mediante unidades que se compone de una celda , una puerta de entrada , una puerta de salida y una puerta de olvido . La celda recuerda valores en intervalos de tiempo arbitrarios y las tres puertas regulan el flujo de información dentro y fuera de la celda.
en: *Keras* es: *Keras*	*Keras* es una biblioteca de Redes Neuronales de código abierto escrita en Python que es capaz de ejecutarse sobre *TensorFlow*, *Microsoft Cognitive Toolkit* o *Theano*
en: *Phishing*	Conjunto de técnicas que persiguen el engaño a una víctima ganándose su confianza haciéndose pasar por una persona, empresa o servicio de confianza

es: Suplantación de identidad	(suplantación de identidad de tercero de confianza), para manipularla y hacer que realice acciones que no debería realizar
en: *Recurrent Neural Network* (*RNN*) es: Red Neuronal Recurrente (*RNR*)	Una red neuronal recurrente no tiene una estructura de capas definida, sino que permiten conexiones arbitrarias entre las neuronas, incluso pudiendo crear ciclos, con esto se consigue crear la temporalidad, permitiendo que la red tenga memoria.
en: *Spam* es: Correo no deseado	Los términos spam, correo basura, correo no deseado o correo no solicitado hacen referencia a los mensajes de correo electrónico no solicitados, no deseados o con remitente no conocido (o incluso correo anónimo o de falso remitente)
en: *Transformer*.	Un *transformer* es un modelo de aprendizaje profundo que adopta el mecanismo de atención , ponderando diferencialmente la importancia de cada parte de los datos de entrada. Se utiliza principalmente en el campo del procesamiento del lenguaje natural (NLP).
en: *Uniform Resource Locator* (*URL*). es: Localizador Uniforme de Recursos	Consiste en *URL* es la dirección específica que se asigna a cada uno de los recursos disponibles en la red con la finalidad de que estos puedan ser localizados o identificados.

7 Bibliografía

[1] |Online Courses & Credentials from Top Educators. Join for Free. (2020). Coursera. Fine Tune BERT for Text Classification with TensorFlow. Retrieved November 14, 2021, from https://www.coursera.org/learn/fine-tune-bert-tensorflow/ungradedLti/ack5t/fine-tune-bert-for-text-classification-with-tensorflow.

[2] ¡Las REDES NEURONALES ahora prestan ATENCIÓN! - TRANSFORMERS ¿Cómo funcionan? (2021, September 27). [Video]. YouTube. https://www.youtube.com/watch?v=aL-EmKuB078.

[3] A developer-friendly guide to model quantization with PyTorch. (2020, December). Spell. Retrieved November 15, 2021, from https://spell.ml/blog/pytorch-quantization-X8e7wBAAACIAHPhT.

[4] Ajuste DistilBERT para tareas de clasificación de texto de etiquetas múltiples. (2020, November 26). ICHI.PRO. Retrieved November 15, 2021, from https://ichi.pro/es/ajuste-distilbert-para-tareas-de-tensorclasificacion-de-texto-de-etiquetas-multiples-168563311180108.

[5] Akib Zabed KhanSaif, Musfique Anwar Mahmud Parvez, Mahbubur Rahman. (2021). Efficient Advertisement Slogan *Detection* and Classification Using a Hierarchical BERT and BiLSTM-BERT Ensemble Model. 17 octubre 2021, de Springer Link Sitio web: https://link.springer.com/chapter/10.1007/978-3-030-68154-8_81

[6] Alammar, J. (2020). The Illustrated Transformer. Jalammar.Github.Io. Retrieved November 1, 2021, from https://jalammar.github.io/illustrated-transformer/.

[7] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, & Illia Polosukhin. (2017). Attention Is All You Need.

[8] BERT (estado del arte) VS regresión logística simple para el procesamiento del lenguaje natural. (2020, 26 noviembre). ICHI.PRO. Recuperado 20 de noviembre de 2021, de https://ichi.pro/es/bert-estado-del-arte-vs-regresion-logistica-simple-para-el-procesamiento-del-lenguaje-natural-216457078276615.

[9] Binary, Multi-class and Multi-label Text Classification w BERT Fine Tuning [90% Accuracy]. (2020, 12 mayo). Reddit. Recuperado 20 de noviembre de 2021, de https://www.reddit.com/r/learnmachinelearning/comments/gijwsm/binary_multiclass_and_multilabel_text/.

[10] Brownlee, J. (2015, 19 agosto). 8 Tactics to Combat Imbalanced Classes in Your Machine Learning Dataset. machinelearningmastery. Recuperado 29 de octubre de 2021, de https://machinelearningmastery.com/tactics-to-combat-imbalanced-classes-in-your-machine-learning-dataset/.

[11] Brownlee, J. (2019, 16 septiembre). A Gentle Introduction to Transfer Learning for Deep Learning. Machine Learning Mastery. Recuperado 17 de octubre de 2021, de https://machinelearningmastery.com/transfer-learning-for-deep-learning/.

[12] Brownlee, J. (2019, August 14). How to use Different Batch Sizes when Training and Predicting with LSTMs. Machine Learning Mastery. Retrieved November 1, 2021, from https://machinelearningmastery.com/use-different-batch-sizes-training-predicting-python-keras/.

[13] Brownlee, J. (2020, August 28). How to Avoid Exploding Gradients With Gradient Clipping. Machine Learning Mastery. Retrieved November 1, 2021, from https://machinelearningmastery.com/how-to-avoid-exploding-gradients-in-neural-networks-with-gradient-clipping/.

[14] Brownlee, J. (2020a, August 14). A Gentle Introduction to Backpropagation Through Time. Machine Learning Mastery. Retrieved November 1, 2021, from https://machinelearningmastery.com/gentle-introduction-backpropagation-time/.

[15] Brownlee, J. (2020c, September 11). Understand the Impact of Learning Rate on Neural Network Performance. Machine Learning Mastery. Retrieved November 1, 2021, from https://machinelearningmastery.com/understand-the-dynamics-of-learning-rate-on-deep-learning-neural-networks/.

[16] Brownlee, J. (2021, January 26). Undersampling Algorithms for Imbalanced Classification. Machine Learning Mastery. Retrieved October 31, 2021, from https://machinelearningmastery.com/undersampling-algorithms-for-imbalanced-classification/.

[17] Brownlee, J. (2021b, February 7). Weight Initialization for Deep Learning Neural Networks. Machine Learning Mastery. Retrieved November 1, 2021, from https://machinelearningmastery.com/weight-initialization-for-deep-learning-neural-networks/.

[18] Chawla, N. V. (2009). Data Mining for Imbalanced Datasets: An Overview. Data Mining and Knowledge Discovery Handbook, 875–886. https://doi.org/10.1007/978-0-387-09823-4_45.

[19] Chowdhury, K. (2021, May 25). 10 Hyperparameters to keep an eye on for your LSTM model — and other tips. Medium. Retrieved October 31, 2021, from https://medium.com/geekculture/10-hyperparameters-to-keep-an-eye-on-for-your-lstm-model-and-other-tips-f0ff5b63fcd4.

[20] Community · TFX. (2020, 11 marzo). Part 1: Fast, scalable and accurate NLP: Why TFX is a perfect match for deploying BERT. Blog.Tensorflow.Org. Recuperado 20 de noviembre de 2021, de https://blog.tensorflow.org/2020/03/part-1-fast-scalable-and-accurate-nlp-tensorflow-deploying-bert.html.

[21] Culurciello, E. (2019, January 10). The fall of RNN / LSTM - Towards Data Science. Medium. Retrieved November 1, 2021, from https://towardsdatascience.com/the-fall-of-rnn-lstm-2d1594c74ce0.

[22] Danka, T. (2021, 12 noviembre). Can a neural network train other networks? - Towards Data Science. Medium. Recuperado 20 de noviembre de 2021, de https://towardsdatascience.com/can-a-neural-network-train-other-networks-cf371be516c6.

[23] De Wang, Shamkant B Navathe, Ling Liu, Danesh Irani, Acar Tamersoy, and Calton Pu. 2013. Click traffic analysis of short *url* spam on twitter. In 9th Intl Conference on Collaborative Computing: Networking, Applications and Worksharing (Collaboratecom). IEEE.

[24] Dhami, D. (2021, 30 julio). Understanding BERT — *Word Embeddings* - Dharti Dhami. Medium. Recuperado 20 de noviembre de 2021, de https://medium.com/@dhartidhami/understanding-bert-word-embeddings-7dc4d2ea54ca.

[25] DistilBERT. (2020). Huggingface.Co. Retrieved November 15, 2021, from https://huggingface.co/transformers/model_doc/distilbert.html.

[26] Doyen Sahoo, Chenghao Liu, & Steven C. H. Hoi. (2019). *Malicious URL Detection* using Machine Learning: A Survey.

[27] Eckhardt, K. (2018, November 29). Choosing the right Hyperparameters for a simple LSTM using Keras. Medium. Retrieved November 1, 2021, from https://towardsdatascience.com/choosing-the-right-hyperparameters-for-a-simple-lstm-using-keras-f8e9ed76f046#:%7E:text=Generally%2C%202%20layers%20have%20shown,to%20find%20reasonably%20complex%20features.

[28] Enrico Sorio, Alberto Bartoli, and Eric Medvet. 2013. *Detection* of hidden fraudulent *url*s within trusted sites using lexical features. In Availability, Reliability and Security (ARES), 2013 Eighth International Conference on. IEEE.

[29] Farzad, A. (2017, October 19). A comparative performance analysis of different. . . Neural Computing and Applications. Retrieved October 31, 2021, from https://link.springer.com/article/10.1007/s00521-017-3210-6?error=cookies_not_supported&code=3dfd8133-94ff-4b50-8d84-854e8e346eb6#:%7E:text=The%20most%20popular%20activation%20functions,functions%20have%20been%20successfully%20applied.

[30] Fernández, A., García, S., Galar, M., Prati, R. C., Krawczyk, B., & Herrera, F. (2018). Learning from Imbalanced Data Sets (2018 ed.). Springer.

[31] Gomila Salas, J. G. G. S. (2021, julio). Deep Learning de A a Z: redes neuronales en Python desde cero. Udemy. Recuperado 17 de octubre de 2021, de https://www.udemy.com/course/deep-learning-a-z/.

[32] Gomila Salas, J. G. G. S. (2021b, agosto). Aprende BERT, el algoritmo de NLP más avanzado de Google. Udemy. Recuperado 17 de octubre de 2021, de https://www.udemy.com/course/bert-nlp/.

[33] Grogan, M. (2021, January 7). CNN-LSTM: Predicting Daily Hotel Cancellations - Towards Data Science. Medium. Retrieved November 1, 2021, from https://towardsdatascience.com/cnn-lstm-predicting-daily-hotel-cancellations-e1c75697f124.

[34] Hernández, A. (2018, 6 mayo). Problema del desvanecimiento del gradiente (vanishing gradient problem). mlearninglab. Recuperado 20 de noviembre de 2021, de https://mlearninglab.com/2018/05/06/problema-de-desvanecimiento-del-gradiente-vanishing-gradient-problem/.

[35] Hsing-Kuo Pao, Yan-Lin Chou, and Yuh-Jye Lee. 2012. *Malicious URL detection* based on Kolmogorov complexity estimation. In Proceedings of the The 2012 IEEE/WIC/ACM International Joint Conferences on Web Intelligence and Intelligent Agent Technology-Volume 01. IEEE Computer Society.

[36] Huajun Huang, Liang Qian, and Yaojun Wang. 2012. A SVM-based technique to detect *phishing URLs*. Information Technology Journal (2012).

[37] Hung Le, Quang Pham, Doyen Sahoo, and Steven CH Hoi. 2018. *URLNet*: Learning a *URL* Representation with Deep Learning for *Malicious URL Detection*. arXiv preprint arXiv:1802.03162 (2018).

[38] INCIBE. (2019). Boletin informativo 5 diciembre 2019. 17 octubre 2021, de Instituto de ciberseguridad Sitio web: https://www.incibe.es/sites/default/files/contenidos/boletines/doc/05.12.19_boletin_seguridad_incibe.pdf

[39] Jacob Devlin, Ming-Wei Chang, Kenton Lee, & Kristina Toutanova. (2019). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding.

[40] Jordi Torres i Viñals. (2020). Python Deep Learning. Introducción práctica con Keras y TensorFlow 2. Barcelona: Marcombo.

[41] Joshua Saxe and Konstantin Berlin. 2017. eXpose: A Character-Level Convolutional Neural Network with Embeddings For Detecting *Malicious URLs*, File Paths and Registry Keys. arXiv preprint arXiv:1702.08568 (2017).

[42] Justin Ma, Lawrence K Saul, Stefan Savage, and Geoffrey M Voelker. 2009. Beyond blacklists: learning to detect *malicious* web sites from suspicious *URLs*. In Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining.

[43] Justin Ma, Lawrence K Saul, Stefan Savage, and Geoffrey M Voelker. 2009. Identifying suspicious *URLs*: an application of large-scale online learning. In Proceedings of the 26th Annual International Conference on Machine Learning.

[44] Justin Ma, Lawrence K Saul, Stefan Savage, and Geoffrey M Voelker. 2011. Learning to detect *malicious urls*. ACM Transactions on Intelligent Systems and Technology (TIST) (2011).

[45] K. (2021, October 2). Using BERT as an Embedder. Python Wife. Retrieved November 14, 2021, from https://pythonwife.com/using-bert-as-an-embedder/.

[46] Kurt Thomas, Chris Grier, Justin Ma, Vern Paxson, and Dawn Song. 2011. Design and evaluation of a real-time *url* spam filtering service. In Security and Privacy (SP), 2011 IEEE Symposium on. IEEE.

[47] Lam, E. (2019, December 13). What happens after Bert? Summarize those ideas behind. Medium. Retrieved November 15, 2021, from https://medium.com/analytics-vidhya/what-happens-after-bert-summarize-those-ideas-behind-ee02f1eae5d9.

[48] Li Xu, Zhenxin Zhan, Shouhuai Xu, and Keying Ye. 2013. Cross-layer *detection* of *malicious* websites. In Proceedings of the third ACM conference on Data and application security and privacy. ACM.

[49] Mingxing He, Shi-Jinn Horng, Pingzhi Fan, Muhammad Khurram Khan, Ray-Shine Run, Jui-Lin Lai, Rong-Jian Chen, and Adi Sutanto. 2011. An efficient *phishing* webpage detector. *Expert Systems with Applications* (2011).

[50] Mohammed Nazim Feroz and Susan Mengel. 2014. Examination of data, rule generation and *detection* of *phishing URLs* using online logistic regression. In *Big Data (Big Data), 2014 IEEE International Conference on*.

[51] Nagpal, A. (2019, December 11). L1 and L2 Regularization Methods - Towards Data Science. Medium. Retrieved November 1, 2021, from https://towardsdatascience.com/l1-and-l2-regularization-methods-ce25e7fc831c.

[52] Phi, M. (2020, June 28). Illustrated Guide to LSTM's and GRU's: A step by step explanation. Medium. Retrieved October 31, 2021, from https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21.

[53] Poulopoulos, D. (2020, 27 noviembre). Cómo utilizar el suavizado de etiquetas para la regularización. ICHI.PRO. Recuperado 20 de noviembre de 2021, de https://ichi.pro/es/como-utilizar-el-suavizado-de-etiquetas-para-la-regularizacion-187142990929339.

[54] Pranam Kolari, Tim Finin, and Anupam Joshi. 2006. SVMs for the Blogosphere: Blog Identification and Splog *Detection*. In AAAI Spring Symposium: Computational Approaches to Analyzing Weblogs.

[55] Samuel Marchal, Jérôme François, Radu State, and Thomas Engel. 2014. PhishScore: Hacking phishers' minds. In Network and Service Management (CNSM), 2014 10th International Conference on. IEEE.

[56] Samuel Marchal, Jérôme François, Radu State, and Thomas Engel. 2014. PhishStorm: Detecting Phishing With Streaming Analytics. Network and Service Management, IEEE Transactions on (2014).

[57] Sepp Hochreiter, Jürgen Schmidhuber; Long Short-Term Memory. Neural Comput 1997; 9 (8): 1735–1780. doi: https://doi.org/10.1162/neco.1997.9.8.1735.

[58] Singh, A. K. (2020, 2 mayo). Dataset of *Malicious* and Benign Webpages. Mendeley Data. Recuperado 17 de octubre de 2021, de https://data.mendeley.com/datasets/gdx3pkwp47/2

[59] Sudharsan Ravichandiran. (2021). Getting Started with Google BERT. Birmingham: Packt Publishing Ltd.

[60] Sushma Nagesh Bannur, Lawrence K Saul, and Stefan Savage. 2011. Judging a site by its content: learning the textual, structural, and visual features of *malicious* web pages. In *Proceedings of the 4th ACM Workshop on Security and Artificial Intelligence.* ACM.

[61] TensorFlow. (2021, November 12). TensorFlow Hub. Text Embedding. Bert_en_uncased_L-12_H-768_A-12. Retrieved November 14, 2021, from https://tfhub.dev/tensorflow/bert_en_uncased_L-12_H-768_A-12/4.

[62] Tensorflow.org. (2020, September 21). The Sequential model | TensorFlow Core. TensorFlow. Retrieved November 1, 2021, from https://www.tensorflow.org/guide/keras/sequential_model

[63] Understanding LSTM Networks -- colah's blog. (2015, August 27). Colah.Github.Io. Retrieved October 31, 2021, from https://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[64] Université Paris-Est-Marne-la-Vallée. (2021) GanttProject (Versión 3.1) [Programa de ordenador] https://github.com/bardsoftware/ganttproject.

[65] Victor Sanh, Lysandre Debut, Julien Chaumond, & Thomas Wolf. (2020). DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter.

[66] Weibo Chu, Bin B Zhu, Feng Xue, Xiaohong Guan, and Zhongmin Cai. 2013. Protect sensitive sites from phishing attacks using features extractable from inaccessible phishing *URL*s. In *Communications (ICC), 2013 IEEE International Conference on.* IEEE.

[67] Wen Zhang, Yu-Xin Ding, Yan Tang, and Bin Zhao. 2011. *Malicious* web page *detection* based on on-line learning algorithm. In Machine Learning and Cybernetics (ICMLC), 2011 International Conference on. IEEE.

[68] Xiang Zhang, Junbo Zhao, and Yann LeCun. 2015. Character-level convolutional networks for text classification. In Advances in neural information processing systems.

[69] Yazan Alshboul, Raj Nepali, and Yong Wang. 2015. Detecting *malicious* short *URL*s on Twitter. (2015)

[70] Ying Pan and Xuhua Ding. 2006. Anomaly based web *phishing* page *detection.* In Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual. IEEE.

[71] Yung-Tsung Hou, Yimeng Chang, Tsuhan Chen, Chi-Sung Laih, and Chia-Mei Chen. 2010. *Malicious* web content *detection* by machine learning. Expert Systems with Applications (2010).

8 Anexos

A continuación, se muestra la lista de documentos adjuntos a la memoria que complementan el trabajo realizado para este trabajo final de master:

- ***TFM_PEC2_2021_ofrances_dataset_analysis_v1***. Este es un documento autocontenido (*notebook* de *Python*) en el que se explica en detalle el conjunto de datos *[58]* seleccionado. En él se explica uno por uno todos los campos y la influencia que tienen para poder resolver el problema planteado. Parte del capítulo *Fase de investigación y primera DNN* se nutre de este informe de análisis previo.

- ***TFM_PEC2_2021_ofrances_modelling_v1***. Este es un documento autocontenido (*notebook* de *Python*) en el que se explica en detalle el modelado de los datos originales aumentando de 11 dimensiones a más de 250 dimensiones que luego serán la entrada a la primera red neuronal (ANN). A continuación, se genera la primera red neuronal y se obtienen los primeros resultados. Parte del capítulo *Fase de investigación y primera DNN* se nutre de este informe de análisis previo.

- ***TFM_PEC3_2021_ofrances_RNN_Masking_LSTM_content_v1.*** Este es un documento autocontenido (*notebook* de *Python*) en el que se explica en detalle el modelado de los datos originales en dos dimensiones (‘*url*’ y ‘*content*’) así como el modelado en una red neuronal recurrente *LSTM* con un tratamiento de máscaras (*Masking*).

- ***TFM_PEC3_2021_ofrances_RNN_Embedding_LSTM_content_v1_v1.*** Este es un documento autocontenido (*notebook* de *Python*) en el que se explica en detalle el modelado de los datos originales en dos dimensiones (‘*url*’ y ‘*content*’) así como el modelado en una red neuronal recurrente *LSTM* con un tratamiento de vectorización llamado *embedding*.

- ***TFM_PEC3_2021_ofrances_RNN_Embedding_CNN_LSTM_content_v1.*** Este es un documento autocontenido (*notebook* de *Python*) en el que se explica en detalle el modelado de los datos originales en dos dimensiones (‘*url*’ y ‘*content*’) así como el modelado en una red neuronal recurrente con una capa previa al *LSTM* convolucional.

- ***TFM_PEC3_2021_ofrances_RNN_Embedding_bidirectional_LSTM_content_v1***. Este es un documento autocontenido (*notebook* de Python) en el que se explica en detalle el modelado de los datos originales en dos dimensiones (‘*url*’ y ‘*content*’) así como el modelado en una red neuronal recurrente bidireccional LSTM.

- ***TFM_PEC3_2021_ofrances_RNN_Embedding_GRU_content_v1.*** Este es un documento autocontenido (*notebook* de *Python*) en el que se explica en detalle el modelado de los datos originales en dos

dimensiones (‘*url’* y ‘*content’*) así como el modelado en una red neuronal recurrente *Gated Recurrent Units* (*GRU*).

- ***TFM_PEC3_2021_ofrances_BERT_content_v1.*** Este es un documento autocontenido (*notebook* de *Python*) en el que se explica en detalle el modelado de los datos originales en dos dimensiones (‘*url’* y ‘*content’*) así como el modelado en una red neuronal con el *tokenizer* de BERT con una capa de *embedding* propia.

- ***TFM_PEC3_2021_ofrances_BERT_embedding_content_v1.*** Este es un documento autocontenido (*notebook* de *Python*) en el que se explica en detalle el modelado de los datos originales en dos dimensiones (‘*url’* y ‘*content’*) así como el modelado en una red neuronal con el tokenizer de BERT, así como la capa de *embedding* también de *BERT*. Este modelo no permite *fine tuning*.

- ***TFM_PEC3_2021_ofrances_BERT_embedding_fine_tuning_content_v1.*** Este es un documento autocontenido (*notebook* de *Python*) en el que se explica en detalle el modelado de los datos originales en dos dimensiones (‘*url’* y ‘*content’*) así como el modelado en una red neuronal con el tokenizer de BERT, así como la capa de *embedding* también de *BERT*. Este modelo permite *fine tuning*.