MÁSTER EN INGENIERÍA EN TECNOLOGÍAS DE TELECOMUNICACIÓN

ANÁLISIS DE GENERADORES DE CONTRASEÑAS BASADOS EN DEEP LEARNING PARA APLICACIONES CRIPTOGRÁFICAS

Autor:

Alejandro Rodríguez García

Director:

Jaime Boal Martín-Larrauri

Declaro, bajo mi responsabilidad, que el proyecto presentado con el título

"ANÁLISIS DE GENERADORES DE CONTRASEÑAS BASADOS EN DEEP LEARNING PARA APLICACIONES CRIPTOGRÁFICAS"

en la ETS de Ingeniería - ICAI de la Universidad Pontificia Comillas en el curso académico 2022/23 es de mi autoría, original e inédito y no ha sido presentado con anterioridad a otros efectos.

El Proyecto no es plagio de otro, ni total ni parcialmente y la información que ha sido tomada de otros documentos está debidamente referenciada.

Fdo.: Alejandro Rodríguez García Fecha: 13 / 07 / 2023

Autorizada la entrega del proyecto

EL DIRECTOR DEL PROYECTO

Firmado digitalmente por BOAL MARTIN LARRAURI JAIME - 05304600H
Fecha: 2023.07.13 19:57:02 +02'00'

Fdo.: Dr. Jaime Boal Martín-Larrauri Fecha: 13 / 07 / 2023

MÁSTER EN INGENIERÍA EN TECNOLOGÍAS DE TELECOMUNICACIÓN

ANÁLISIS DE GENERADORES DE CONTRASEÑAS BASADOS EN DEEP LEARNING PARA APLICACIONES CRIPTOGRÁFICAS

Autor:

Alejandro Rodríguez García

Director:

Jaime Boal Martín-Larrauri

ANÁLISIS DE GENERADORES DE CONTRASEÑAS BASADOS EN DEEP LEARNING PARA APLICACIONES CRIPTOGRÁFICAS

Autor: Rodríguez García, Alejandro
Directores: Dr. Jaime Boal Martín-Larrauri
Entidad Colaboradora: ICAI- Universidad Pontificia Comillas

ABSTRACT

El uso de modelos de deep learning ha estado revolucionando el estado del arte de la adivinación de contraseñas en los últimos años. En este proyecto se analizan los modelos más recientes de generación de contraseñas, especialmente aquellos que usan redes generativas antagónicas (GAN). Se han estudiado los aspectos teóricos y la implementación de los modelos poniendo el foco en GNPassGAN, la alternativa que se ha encontrado que presenta mejores resultados. De los resultados obtenidos se ha concluido que GNPassGAN se puede considerar como la mejor solución de todos los estudiados alcanzando con una tasa de adivinación de contraseñas del 7,57% en el conjunto de contraseñas RockYou y del 23% en el conjunto de Pwned. Finalmente, se han propuesto nuevas líneas de trabajo para mejorar la seguridad de los sistemas que hacen uso de contraseñas, protegiéndolos ante el uso ofensivo de los modelos de adivinación de contraseñas.

Palabras clave: PassGAN, GAN, Contraseñas, Deep Learning, Criptografía.

1. Motivación y Objetivos

La investigación en métodos de autenticación y predicción de contraseñas está en constante desarrollo. Estos métodos no solo son relevantes para los delincuentes cibernéticos y las personas que buscan protegerse, sino que también tienen diversas aplicaciones como las de los cuerpos de seguridad oficiales que requieren acceder legalmente a archivos encriptados por los criminales. Por ello es vital estudiar y desarrollar estas herramientas de manera abierta y transparente.

En este proyecto se evalúa modelos de redes generativas antagónicas (GAN) basados en el estado del arte de las técnicas de generación de contraseñas. El objetivo es contribuir con un análisis exhaustivo del uso de modelos basados en PassGAN [1], modelo original basado en una GAN para la generación de contraseñas, sobre los sistema actuales basados en contraseñas. La aplicación directa del trabajo es en el ámbito de la criptografía ofensiva, donde la capacidad de poder enriquecer las bases de datos de contraseñas permite incrementar la probabilidad de romper los hash de contraseñas capturados entre otras muchas aplicaciones. Para poder defenderte ante una amenaza como suponen PassGAN para los sistemas de seguridad hay que primero conocer su potencial y tomar contramedidas.

2. Metodología y Recursos

Para poder evaluar el rendimiento de estos modelos, en primer lugar, se ha realizado una análisis de las diferencias teóricas de cada modelo para situar cada uno dentro del estado del arte y bajar al

detalle de que les diferencia. En segundo lugar, se ha estudiado la viabilidad de la implementación de los modelos alternativos considerando los repositorios de código asociados y, de los que se consideren de interés, se implementarán y se compararán los resultados. En tercer lugar, se ha hecho el análisis exhaustivo del mejor modelo, GNPassGAN [2], analizando tanto la evolución del entrenamiento como la capacidad generativa y calidad de las contraseñas.

3. Análisis teórico de los modelos

Todos los modelos analizados nacen a raíz de estudios con un mismo hilo conductor, el conseguir estabilizar el entrenamiento de las GAN. Este es uno de los principales retos de las GAN, especialmente, hablando de la generación de texto debido a que si en el proceso de entrenamiento no se encuentran en igualdad de condiciones al generador y al discriminante siempre ganará el mismo y, por lo tanto, no se retroalimentaran en el aprendizaje.

El primer avance en este ámbito fue con el modelo *WGAN*, añadiendo durante el entrenamiento el uso de la distancia de *Wasserstein* para medir la diferencia entre la distribución de probabilidad de las muestras generadas y la distribución real de los datos. Dicha distancia es la utilizada en la función de perdidas.

Sin embargo, *WGAN* tiene otros defectos como el "colapso de modos" y la falta de una métrica para detectar la convergencia. El "colapso de modos" se produce cuando el generador produce una variedad limitada de salidas, ignorando u omitiendo la diversidad en los datos reales. Buscando solucionar dicho problema se propuso *IWGAN*, que utiliza una función de pérdida mejorada para el generador incluyendo un término adicional de regularización basado en la divergencia de Jensen-Shannon (JS).

El siguiente trabajo realizado con el objetivo de estabilizar aún más el entrenamiento fue con *WGAN-GP*, que propone una penalización del gradiente conocida como la restricción de *Lipschitz*. La penalización del gradiente ayuda a suavizar las funciones de activación y limitar las derivadas parciales del discriminador, lo cual promueve una mayor estabilidad en el entrenamiento. *PassGAN* se basa la arquitectura *WGAN-GP*, usando su penalización del gradiente y el mismo protocolo de entrenamiento.

Después de estas propuestas se introdujo *GNPassGAN*, que incorpora varias propuestas nuevas de normalización de gradiente, de función de activación y de función de perdidas. Destaca la mejora en la normalización del gradiente mediante el cálculo de un factor de normalización adaptativo para cada gradiente parcial (derivada parcial de la función objetivo con respecto a un parámetro concreto del modelo).

4. Análisis empírico

En esté capitulo se ha analizado en primer lugar el estado del arte de la implementación de los modelos disponibles, se ha concluido que en la mayoría de los casos no están publicados los códigos en los que se basan los artículos y de los que si hay código publicado casi ninguno se realizó por los autores, si no por implementaciones de particulares a posteriori. Además, la documentación de los repositorios es escasa y se usan en casi todos los modelos versiones antiguas de *TensorFlow*, *PyTorch* y *Python*, lo que complica significativamente su replicabilidad. Para asegurar la estandarización de los experimentos se decide partir del código *GNPassGAN* para evaluar la propuesta de usar GANs para la generación de contraseñas.

GNPassGAN se ha entrenado utilizando el conjunto de contraseñas filtradas llamado *RockYou*, guardando el modelo cada distinto número de iteraciones de entrenamiento. Una vez terminado el entrenamiento, para cada modelo guardado, se ha usado el generador para producir nuevas contraseñas.

Los principales resultados obtenidos del análisis *GNPassGAN* en cuanto a la evolución del entrenamiento son que el modelo sigue aprendiendo con bastante ruido (a pesar de todas las normalizaciones propuestas), pero aun así es bastante estable y equilibrado entre generador y discriminante.

En cuanto a la capacidad generativa se ha obtenido una mejora del *accuracy* en *RockYou* de más del 2% respecto al modelo original pre-entrenado por los autores del repositorio como de los resultados reflejados en el artículo [2]. Esto supone que consigue un accuracy total del 7.57%. Además, se ha observado que el punto óptimo en términos de *accuracy* se alcanza después de 250.000 iteraciones de entrenamiento lo que implica que a medida que el modelo se entrena durante más iteraciones, se logra una mayor precisión en el conjunto de *RockYou* en la generación de contraseñas que coinciden con el conjunto de prueba. En el caso del análisis del *accuracy* en *Pwned* se han conseguido *accuracy* muchos mayores, de hasta el 23%. Sin embargo, se ha observado que el modelo puede llegar a estar sobreaprendiendo el conjunto *RockYou*, ya que los mejores *accuracy* se obtienen para el modelo entrenado con 50.000 iteraciones y no el de 250.000 (el óptimo en el caso del *accuracy* con *RockYou)*.

Otro resultado es la tendencia en la disminución de la entropía de las contraseñas a medida que se incrementa el número de contraseñas generadas. Hay que considerar que el mayor decremento, cuando se pasa de 10^7 a 10^8 es del 15% de contraseñas únicas, se podría argumentar que es un decremento significativo, pero es que has multiplicado por 10 el

número de contraseñas generadas. Esto significa que un con 10^8 has generado 80 millones de contraseñas únicas, frente a los 9'5 millones con 10^7 por lo que sigue saliendo muy a cuenta continuar generando contraseñas, aunque la eficiencia disminuya.

5. Conclusiones y Trabajos futuros

La principal conclusión de este trabajo es que en el análisis de un total de 12 repositorios de código relacionados con modelos basados en PassGAN se ha podido comprobar que existe una gran cantidad de artículos que proponen mejoras sobre PassGAN, sin embargo, ninguno alcanza el *accuracy* que consigue GNPassGAN. Está afirmación está basada en los resultados de *accuracy* sobre la base de datos de contraseñas pública más grande existente a día de hoy, *Pwned*.

En el afán de analizar de manera exhaustiva la capacidad de adivinación de *GNPassGAN*, no solo se ha orientado el proyecto al estudio de la *accuracy*, si no que se ha contribuido con el análisis de múltiples aspectos de *GNPassGAN* que no se contemplan en los artículos publicados: la evolución de las perdidas en el entrenamiento, la proporción de contraseñas únicas, el ruido de generación de contraseñas distintas en función del número de iteraciones de entrenamiento o la distribución de las contraseñas generadas en cuanto a caracteres y *n-grams* y longitud.

En cuanto a futuros trabajos, como consecuencia de las conclusión extraída de que *GNPassGAN* es el principal método actual de adivinación de contraseñas, se propone dos nuevas líneas de trabajo: La primera es el estudio de la capacidad del discrimínate para detectar contraseñas seguras generadas por el modelo. De esta forma se puede evaluar cada vez que un usuario introduzca una nueva contraseña en el sistema como de probable es que un atacante usando *GNPassGAN* la pueda adivinar. La segunda línea es la implementación de una sistema de *honeyword* que permita detectar si se han producido filtraciones de contraseñas en el sistema. El objetivo sería integrar *GNPassGAN* dentro del sistema de gestión de las contraseñas, añadiendo a la base de datos contraseñas muy similares a las existentes pero que a su vez no hayan sido creadas por ningún usuario. De esta forma se consigue detectar si han intentado con algunas de las contraseñas ficticias si se ha producido un fuga de información.

6. Referencias

[1]. Hitaj, P. Gasti, G. Ateniese, and F. Perez-Cruz, "PassGAN: A Deep Learning Approach for Password Guessing." arXiv, 2017. doi: 10.48550/ARXIV.1709.00440.

[2]. F. Yu and M. V. Martin, "GNPassGAN: Improved Generative Adversarial Networks For Trawling Offline Password Guessing," in 2022 IEEE European Symposium on Security and Privacy Workshops

ANALYSIS OF DEEP LEARNING BASED PASSWORD GENERATORS FOR CRYPTOGRAPHIC APPLICATIONS

Author: Rodríguez García, Alejandro
Directors: Dr. Jaime Boal Martín-Larrauri
Collaborating Institution: ICAI- Universidad Pontificia Comillas

ABSTRACT

The use of deep learning models has been revolutionizing the state of the art of password guessing in recent years. This project analyses the most recent models of password generation, especially in the use of generative antagonistic networks (GAN). The theoretical aspects and the implementation of the models have been studied, focusing on GNPassGAN, one of the alternatives that has been found to present the best results. From the results obtained, it has been concluded that GNPassGAN can be considered as the best solution of all those studied, reaching a password guessing rate of 7.57% in the RockYou password set and 23% in the Pwned set. Finally, new lines of work have been proposed to improve the security of systems that make use of passwords, protecting them against the offensive use of password guessing models.

Keywords: PassGAN, GAN, Passwords, Deep Learning, Cryptography.

1. Motivation and Objectives

Research into authentication and password prediction methods is constantly developing. These methods are not only relevant for cybercriminals and individuals seeking protection, but also have diverse applications such as official law enforcement agencies that require legal access to files encrypted by criminals. It is therefore vital to study and develop these tools in an open and transparent manner.

This project evaluates generative antagonistic network (GAN) models based on state-of-the-art password generation techniques. The objective is to contribute with a comprehensive analysis of the use of models based on PassGAN [1], the original GAN-based model for password generation, over current password-based systems. The direct application of the work is in the field of offensive cryptography, where the ability to enrich password databases allows to increase the probability of breaking captured password hashes among many other applications. In order to be able to defend against a threat such as PassGAN to security systems, you must first understand its potential and take countermeasures.

2. Metodología y Recursos

In order to evaluate the performance of these models, firstly, an analysis of the theoretical differences of each model has been carried out to situate each one within the state of the art and to go down to the detail of what differentiates them. Secondly, the feasibility of implementing the alternative models has been studied considering the associated code repositories and, of those

considered to be of interest, they will be implemented and the results compared. Thirdly, an exhaustive analysis of the best model, GNPassGAN [2], has been carried out, analyzing both the training evolution and the generative capacity and quality of the passwords.

3. Theoretical analysis of the models

All the models analyzed are the result of studies with the same common thread: stabilizing the training of GANs. This is one of the main challenges of GANs, especially in the case of text generation, because if the generator and the discriminant are not on an equal footing in the training process, the same one will always win and, therefore, there will be no feedback in the learning process.

The first advance in this area was with the WGAN model, adding during training the use of the Wasserstein distance to measure the difference between the probability distribution of the generated samples and the real distribution of the data. This distance is the one used in the loss function.

However, WGAN has other shortcomings such as "mode collapse" and the lack of a metric to detect convergence. Mode collapse" occurs when the generator produces a limited variety of outputs, ignoring or omitting diversity in the real data. In order to solve this problem, IWGAN was proposed, which uses an improved loss function for the generator by including an additional regularization term based on the Jensen-Shannon (JS) divergence.

The next work done with the aim of further stabilizing the training was with WGAN-GP, which proposes a gradient penalty known as the Lipschitz constraint. The gradient penalty helps to smooth the activation functions and limit the partial derivatives of the discriminator, which promotes greater stability in training. PassGAN is based on the WGAN-GP architecture, using its gradient penalty and the same training protocol.

Following these proposals, GNPassGAN was introduced, which incorporates several new proposals for gradient normalization, activation function and loss function. Of particular note is the improvement in gradient normalization by calculating an adaptive normalization factor for each partial gradient (partial derivative of the objective function with respect to a specific model parameter).

4. Empirical analysis

In this chapter we have first analyzed the state of the art of the implementation of the available models, and we have concluded that in most cases the codes on which the articles are based are not published, and of those for which there is published code, almost none was

done by the authors, but by private implementations a posteriori. Furthermore, the documentation in the repositories is scarce and almost all the models use old versions of TensorFlow, PyTorch and Python, which significantly complicates their replicability. To ensure the standardization of the experiments, it was decided to start from the GNPassGAN code to evaluate the proposal to use GANs for password generation.

GNPassGAN has been trained using the set of filtered passwords called RockYou, saving the model every different number of training iterations. Once the training was finished, for each saved model, the generator was used to produce new passwords.

The main results obtained from the GNPassGAN analysis in terms of training evolution are that the model continues to learn with quite a lot of noise (despite all the normalizations proposed), but it is still quite stable and balanced between generator and discriminant.

Regarding the generative capacity, an improvement of the accuracy in RockYou of more than 2% has been obtained with respect to the original model pre-trained by the authors of the repository and the results reflected in the article [2]. This means that it achieves a total accuracy of 7.57%. Furthermore, it has been observed that the optimal point in terms of accuracy is reached after 250,000 training iterations, which implies that as the model is trained for more iterations, a higher accuracy is achieved in the RockYou set in the generation of passwords that match the test set. In the case of the accuracy analysis in Pwned, much higher accuracies of up to 23% have been achieved. However, it has been observed that the model may be overlearning the RockYou set, since the best accuracy is obtained for the model trained with 50,000 iterations and not the one with 250,000 (the optimum in the case of accuracy with RockYou).

Another result is the trend of decreasing password entropy as the number of passwords generated increases. Consider that the largest decrease from 107 to 108 is 15% of unique passwords, one could argue that this is a significant decrease, but you have multiplied the number of passwords generated by 10. This means that with 108 you have generated 80 million unique passwords, compared to 9.5 million with 107, so it is still very cost-effective to continue generating passwords, even if the efficiency decreases.

5. Conclusions and Future Work

The main conclusion of this work is that in the analysis of a total of 12 code repositories related to PassGAN-based models, it has been found that there is a large number of articles that propose improvements on PassGAN, however, none reaches the accuracy achieved by

GNPassGAN. This statement is based on the results of accuracy on the largest public password database existing today, Pwned.

In order to exhaustively analyze the guessing capacity of GNPassGAN, the project has not only focused on the study of accuracy, but has also contributed to the analysis of multiple aspects of GNPassGAN that are not covered in the published articles: the evolution of losses in training, the proportion of unique passwords, the generation noise of different passwords depending on the number of training iterations or the distribution of the passwords generated in terms of characters and n-grams and length.

As for future work, as a consequence of the conclusion drawn that GNPassGAN is the main current method of password guessing, two new lines of work are proposed: The first is the study of the ability of the discriminator to detect secure passwords generated by the model. In this way, each time a user enters a new password into the system, it can be evaluated how likely it is that an attacker using GNPassGAN can guess it. The second line is the implementation of a honeyword system to detect if passwords have been leaked in the system. The objective would be to integrate GNPassGAN into the password management system, adding passwords to the database that are very similar to the existing ones but which in turn have not been created by any user. In this way, it is possible to detect whether any of the fictitious passwords have been tried if there has been an information leak.

6. References

[1]. Hitaj, P. Gasti, G. Ateniese, and F. Perez-Cruz, "PassGAN: A Deep Learning Approach for Password Guessing." arXiv, 2017. doi: 10.48550/ARXIV.1709.00440.

[2]. F. Yu and M. V. Martin, "GNPassGAN: Improved Generative Adversarial Networks For Trawling Offline Password Guessing," in 2022 IEEE European Symposium on Security and Privacy Workshops

Índice de la memoria

Índice de figuras

1. INTRODUCCIÓN

En este trabajo fin de máster se evalúa el estado del arte de las técnicas de generación de contraseñas centrándose en una de las implementaciones más modernas, PassGAN [1] un generador basado en redes generativas antagónicas (GAN). La aplicación directa del trabajo es en el ámbito de la criptografía ofensiva, donde la capacidad de poder enriquecer las bases de datos de contraseñas permite incrementar la probabilidad de romper los *hash* de contraseñas capturados entre otras muchas aplicaciones.

1.1. MOTIVACIÓN DEL PROYECTO

La investigación en métodos de autenticación y predicción de contraseñas está en constante desarrollo. Estos métodos no solo son relevantes para los delincuentes cibernéticos y las personas que buscan protegerse, sino que también tienen diversas aplicaciones. Desde usuarios regulares que han olvidado sus contraseñas y necesitan recuperarlas, hasta cuerpos de seguridad oficiales como la policía o el ejército, que requieren acceder legalmente a archivos encriptados por los criminales. Es por eso que resulta vital estudiar y desarrollar estas herramientas de manera abierta y transparente.

Lo cierto es que, aunque se ha avanzado mucho en la implementación de distintas alternativas para la autentificación, como el uso de múltiples factores, en la práctica no se han llegado a extender de manera masiva. Por un lado, debido a la complejidad que supone implementar dichos sistemas por parte de los desarrolladores frente a la facilidad de establecer un método clásico de contraseñas. Normalmente su implementación resulta compleja por la necesidad de disponer de dispositivos auxiliares y, sobre todo, que se puedan integrar con el sistema al que se quiere acceder, ya que se necesita una aplicación cliente instalada en todos los dispositivos involucrados. Por otro lado, plataformas como Google lleva utilizando métodos MFA (*Multifactor Autentification*) para autentificarse desde hace tiempo pero, sin embargo, los usuarios no le dan uso debido a la necesidad de tener un teléfono móvil siempre cerca, tener que responder a preguntas de autentificación o la necesidad de disponer de dispositivos que permitan autentificación con datos biométricos.

Aparte de lo ya expuesto sobre la autentificación MFA hay que tener en cuenta que muchas aplicaciones delegan la autentificación en plataformas como Google permitiendo iniciar sesión

con cuentas de dichas plataformas. Se puede argumentar que es una ventaja, ya que permite que plataformas en las que la autentificación no es tan crítica y no están dispuestas a dedicar una gran cantidad de recursos a la tarea puedan delegarla a las plataformas o sistemas en los que se presupone un alto nivel de seguridad. Sin embargo, hay que considerar que el hecho de que la autentificación en una única plataforma pueda dar acceso a una gran variedad de servicios implica que si dicha autentificación se ve comprometida la fuga de información o el daño que puede hacer un ciberdelincuente se incrementa sustancialmente.

Si se suma el hecho de que la autenticación MFA de las plataformas "seguras" se limita muchas veces por parte del usuario a la utilización de contraseñas simples, junto con el hecho de que el acceso con esta plataforma permite entrar en una amplia gama de aplicaciones, nos lleva a concluir el hecho de que tener contraseñas seguras es crítico para la ciberseguridad actual.

1.2. OBJETIVOS

El objetivo de este trabajo fin de máster es evaluar el estado del arte de las técnicas de generación de contraseñas centrándose en una de las implementaciones más modernas, PassGAN, un generador basado en redes generativas antagónicas (GAN). Alineado con las motivaciones del proyecto, se busca contribuir con un análisis exhaustivo de los nuevos retos puedes suponer el uso de PassGAN sobre los sistema actuales basados en contraseñas.

Por un lado, se ha realizado un análisis teórico de la arquitectura del modelo de generación y su proceso de entrenamiento, enmarcándolo dentro de las distintas alternativas de modelos de inteligenciar artificial generativa. Por el otro, se han probado las implementaciones de distintos artículos relacionados con PassGAN escogiendo el que presenta un mejor comportamiento. Finalmente, se han elaborado una serie de pruebas que arrojan luz no solo sobre el potencial del generador para generar contraseñas (a lo que se limitan prácticamente la totalidad de los artículos publicados), sino que también lo hacen sobre aspectos importantes como: la evolución del aprendizaje del modelo, la proporción de contraseñas únicas, la proporción de cada caracteres alfanumérico o la similitud de las contraseñas generadas con distintas iteraciones de entrenamiento. Además, para realizar una evaluación lo más exhaustiva posible de la adivinación del modelo se han evaluado las contraseñas generadas en la base de datos de acceso público más grande de contraseñas filtradas, Pwned.

La planificación seguida para alcanzar los objetivos ha sido, en primer lugar, el estudio de la implementación del modelo del artículo original y la correspondiente evolución de los resultados.

En segundo lugar, la viabilidad de la implementación de los modelos alternativos considerando los repositorios de código asociados y, de los que se consideren de interés, se implementarán y se compararán los resultados. En tercer lugar, el análisis exhaustivo de la mejor implementación entrando en detalle en su análisis teórico y práctico.

En la *Figura 1* y *Figura 2* se describe las tareas y su evolución temporal:

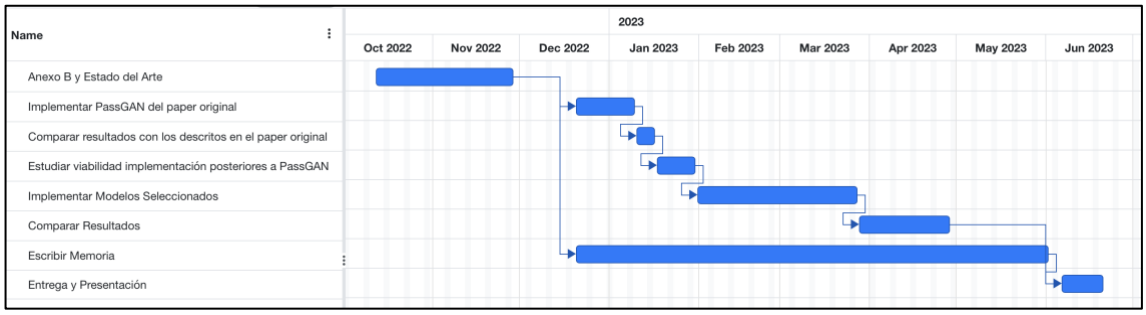

Figura 1: Diagrama Gantt con la planificación.

Name	Start Date	End Date
Anexo B y Estado del Arte	Oct 12, 2022	Nov 28, 2022
Implementar PassGAN del paper original	Dec 20, 2022	Jan 09, 2023
Comparar resultados con los descritos en el paper original	Jan 10, 2023	Jan 16, 2023
Estudiar viabilidad implementación posteriores a PassGAN	Jan 17, 2023	Jan 30, 2023
Implementar Modelos Seleccionados	Jan 31, 2023	Mar 27, 2023
Comparar Resultados	Mar 28, 2023	Apr 28, 2023
Escribir Memoria	Dec 20, 2022	Jun 01, 2023
Entrega y Presentación	Jun 06, 2023	Jun 20, 2023

Figura 2: Detalle de las fechas de inicio y fin de cada tarea

1.3. ESTRUCTURA DE LA MEMORIA

La memoria se estructura en siete capítulos y se divide en cuatro bloques principales. El primer bloque, conformado por el *Capítulo 1* y el *Capítulo 2*, donde se introduce el proyecto abordando sus objetivos, planificación y motivación. En el *Capítulo 2* , se realiza un análisis exhaustivo de los avances relacionados con métodos de autentificación centrándose en las contraseñas, así como los métodos de precocción o adivinación de contraseñas. Además, se entra en detalle de que son y como se aplican en la adivinación de contraseñas las redes generativas antagónicas (GAN) y las múltiples aplicaciones que tienen en el ámbito de la autenticación por contraseñas.

El segundo bloque, el *Capítulo 3*, se enfoca en el análisis teórico de los modelos en los que se basa *PassGAN* y a la vez de los que se basan en él. Analizando sus diferencias, como se implementa su entrenamiento, sus función de pérdidas y las distintas implementaciones para la normalización del gradiente.

El tercer bloque, el *Capítulo 4*, se enfoca en el análisis teórico de los modelos estudiados. En el *apartado 4.1* se evalúa la implementación de los modelos existentes relacionados con *PassGAN*, se presentan conclusiones de las implementaciones y se escoge un modelo para su estudio en detalle. A continuación, en el *apartado 4.2.*, se detalla la metodología del entrenamiento del modelo escogido junto con los parámetros utilizados. En el resto de los apartados del capítulo se analiza el entrenamiento, centrándose en la función de pérdidas y la distribución de los *N-Grams*, se examinan las contraseñas generadas y se analizan aspectos como la precisión en las bases de datos RockYou y Pwned, la proporción de contraseñas únicas y la distribución de longitudes y caracteres. Además, dentro de este mismo bloque, en el *apartado 4.5.* se realiza un análisis de *GNPassGAN* con léxico real, definiendo la metodología específica utilizada, la evaluación de la evolución del entrenamiento y finalmente el análisis de las palabras generadas.

El cuarto bloque formado por los capítulos *5* y *6*, donde se presentan las conclusiones, los futuros trabajos derivados del estudio realizado y la bibliografía de la memoria. Finalmente, se incluyen dos anexos: uno sobre los distintos entornos de ejecución utilizado y otro que describe la alineación del proyecto con los ODS (Objetivos de Desarrollo Sostenibles).

2. ESTADO DE LA CUESTIÓN

Históricamente las contraseñas han supuesto, suponen y seguramente supondrán uno de los métodos más comunes de autentificación. En nuestro día a día hacemos uso de contraseñas para acceder al correo electrónico, a las aplicaciones del banco, a compras en línea y un largo etcétera. Entre las múltiples contraseñas que manejamos se encuentran muchos patrones, fórmulas que nos ayudan a recordarlas o incluso tendemos a repetirlas por completo. Esto las convierte en uno de los principales objetivos de los cibercriminales para comprometer la seguridad de los sistemas.

2.1. CONTRASEÑAS

La finalidad de las contraseñas es cumplir con uno de los requisitos principales para afirmar que un sistema sea seguro, la autentificación, es decir, que el sistema al que accede un usuario sea capaz de verificar si el usuario es quien asegura ser. Tener contraseñas fuertes y únicas para proteger la información es uno de los pasos más simples, pero a menudo ignorados. La mayoría de los usuarios tienden a establecer contraseñas simples y fáciles de recordar cómo se apunta en [2] y [3]. Además, normalmente no se da suficiente información a los usuarios sobre la importancia de las contraseñas seguras y se limita a mensajes como “La contraseña debe contener minúsculas, mayúsculas y números”.

En [4] se realiza un estudio a gran escala sobre la creación de contraseñas en la que una muestra de 2.931 usuarios hace uso de 14 “medidores de contraseñas”. Encontraron que los medidores mejoraban la calidad de las contraseñas generadas por los usuarios y, sin embargo, las contraseñas resultantes solo eran marginalmente más resistentes a los ataques de predictores de contraseñas, ya que estas seguían teniendo patrones similares a las contraseñas originales. Otra de las conclusiones interesantes extraídas es que el medidor más estricto fue el que más incomodó a los usuarios a la hora de crear la contraseña y no proporcionó beneficios de seguridad frente a los medidores más permisivos. Se han realizado numerosos estudios similares [5] [6] [7] [8] [9], donde los resultados revelan que la mayoría de los usuarios repiten contraseñas, usan contraseñas correlacionadas entre sí y que pueden recordar fácilmente vinculándolas a objetos o acciones cotidianos. Además, muestran que no se cumplen las suposiciones de las publicaciones NIST SP 800-63 [10] donde se definen métodos de estimación de la entropía de las contraseñas generadas por el usuario en función de si se usan palabras conocidas, la longitud, el tipo de caracteres usados,

etc. Dentro de estas suposiciones se encuentra, por ejemplo, que los usuarios creen contraseñas con la longitud mínima requerida o que introduzcan un número mínimo de caracteres especiales o numéricos. La implicación directa de que la estimación de entropía de la NIST no se cumpla es que sus recomendaciones de longitudes mínimas de contraseñas no sean suficientes para que el sistema sea seguro y, por lo tanto, sean vulnerables ante ataques.

Posteriormente dicho método de estimación de la calidad de las contraseñas basado en la entropía se eliminó en la última versión del documento, la NIST SP 800-63-3 [11]. En su documento asociado NIST SP 800-63B [12] se expone una serie de requisitos que deben tener cada uno de los métodos de autentificación definidos, que en el caso de las contraseñas es *Memorized Secrets*:

- Deben tener una longitud mínima de 8 caracteres si es establecida por el usuario o 6 si es establecida automáticamente.
- No debe contener patrones o contraseñas típicas como “password” o “1234”, que deben estar almacenados en una *blacklist*[1].
- Un usuario tiene que ser capaz de crear contraseñas de por lo menos 64 caracteres.
- Las contraseñas deben almacenarse como *hash* y *salted* y nunca truncados. De ese modo se dificulta, en caso de fuga de información en el sistema, descifrar la contraseña original al atacante, como se explica detalladamente en el *apartado 2.2.1.*
- Las contraseñas nunca deben caducar, ya que si lo hace se pierde la única forma de autentificar al usuario para acceder al sistema. Lo cual no quiere decir que no se le pueda forzar al usuario a que la renueve.
- Todos los caracteres ASCII/Unicode deben permitirse, incluidos los emojis y espacios.

2.1.1. MÉTODOS DE AUTENTICACIÓN

El estudio formal del uso e implementación de contraseñas se puede establecer en las contribuciones de Fernando Corbató y su equipo en el MIT en la década de los 60 los cuales desarrollaron el sistema conocido como *Compatible Time-Sharing System* (CTSS) [13]. En dicho

[1] Base de datos con el histórico de contraseñas filtradas.

sistema se establece una primera aproximación para la gestión de acceso a sistemas informáticos mediante el uso de contraseñas [14]. En la década de los 70, Robert Morris mientras trabajaba en *Bell Laboratories* realizo las primeras propuestas para el uso de funciones *hash* para guardar las contraseñas de manera segura en sistemas UNIX, trabajo que culminó con la publicación de *Password Security: A Case History* junto con Ken Thomson [15]. En dicha publicación se realiza un análisis de los posibles ataques y vulnerabilidades de los sistemas de almacenamiento de contraseñas, así como, la importancia de almacenar las contraseñas en formato encriptado con una función *hash* para dificultar a los atacantes su uso en caso de fuga de información. Sin embargo, la investigación en nuevos métodos de almacenamiento y protección de las contraseñas se estancaron hasta la década de los 90, cuando se empezaron a obtener evidencias de fugas masivas de contraseñas.

En los últimos años se han desarrollado múltiples métodos de autentificación con el objetivo de dificultar a los atacantes romper la autentificación. Estos factores se pueden clasificar según [11] y [16] como:

- *Factores de conocimiento*: algo que solo el usuario conoce, cómo contraseñas, frase de paso o una respuesta a una pregunta concreta.
- *Factores de posesión*: algo que el usuario posee, como un *token* (*hardware* o *software*), un teléfono móvil o una tarjeta bancaria.
- *Factores biométricos*: algo que el usuario es, como datos biométricos de la cara, las huellas dactilares, etc.

Lo cierto es que, aunque se ha avanzado mucho en la implementación de distintos factores de autentificación, así como en la integración de manera simultánea de varios factores (MFA – *Multifactor Autentification*), en la práctica no se han llegado a extender de manera masiva aún. Como ya se anticipaba en las motivaciones del proyecto (*apartado 1.1.*) la adopción de la autenticación MFA de las plataformas consideradas como “seguras” (Google, Microsoft, etc.) parte del usuario es baja. Si consideramos, además, que muchas aplicaciones delegan la autenticación a estas plataformas, nos lleva a afirmar que escoger contraseñas y métodos seguros es crítico para la ciberseguridad actual.

2.2. PREDICCIÓN DE LAS CONTRASEÑAS

Las contraseñas pertenecen a los factores de autentificación denominados como de conocimiento, como se explica en la sección *2.1*. Esto significa que son vulnerables a los ataques de adivinación, si se conoce lo que se presupone como "secreto" se pierde la capacidad de autenticación y, por lo tanto, el sistema pierde toda su seguridad. Estos ataques han sido ampliamente estudiados y su análisis sigue siendo un área de interés tanto por cibercriminales como por investigadores.

2.2.1. MÉTODOS CLÁSICOS

En primer lugar, es necesario diferenciar entre aquellas técnicas buscan capturar la información directamente del usuario o usuarios a los que se desea robar las credenciales y aquellos métodos que simplemente se basan en adivinar la contraseña basándose en el conocimiento previo de otras fugas de información. Dentro del primer grupo se encuentran ataques como el *phishing* [17] y sus múltiples variantes, *malware* diseñado para recopilar credenciales de un sistema , *SQL Injection* como en una de las filtraciones más famosas, *RockYou* [18], *cold boot attack* [19] donde se deja sin alimentación al dispositivo y se carga sobre este un sistema operativo ligero para leer la memoria temporal y capturar sus claves, y cualquier otra técnica de *ingeniería social* que persiga obtener credenciales[20].

El segundo grupo, el que se basa en el conocimiento de fugas de información previas, se puede dividir a su vez en tres grupos según [21] y [22]: ataques de fuerza bruta, ataques basados en diccionarios y ataques híbridos.

- El *ataque de fuerza bruta* consiste en probar iterativamente múltiples combinaciones de números, letras y caracteres especiales. Es el más fácil de todos los métodos, pero es el que requiere mayor capacidad de cómputo. Sin embargo, a medida que el número de caracteres de la contraseña aumenta el número de combinaciones posibles crece exponencialmente. Existen herramientas como *crunch* [23] que generan listas de contraseñas por fuerza bruta en las que puedes definir patrones de la contraseña como, por ejemplo: todas las contraseñas que comiencen por un "1", seguido de 3 caracteres numéricos y 2 letras desde la "a" hasta la "h".

- El *ataque basado en diccionario* consiste en usar diccionarios que pueden generarse a partir de contraseñas filtradas, combinaciones de palabras del diccionario o números que

la gente puede usar como contraseñas reales. Este tipo de ataques en comparación con los de fuerza bruta permite incrementar la eficiencia del ataque significativamente, ya que se basa en contraseñas reales que usan típicamente los usuarios. Sin embargo, el alcance del ataque está limitado tanto por el número de contraseñas que contiene el diccionario de ataque, como por la calidad de dicho diccionario.

- El *ataque híbrido*, según lo describen en [21] y [22], es un ataque que aumenta el rendimiento de los ataques basados en diccionarios apoyándose en la transformación de las contraseñas candidatas con distintas combinatorias. La generación de nuevas contraseñas se realiza definiendo máscaras de caracteres variables y estableciendo reglas como el cambio de minúsculas por mayúsculas.

Las tres categorías de ataque descritas anteriormente son la clasificación genérica, pero en la práctica existen una gran diversidad de técnicas que se pueden clasificar dentro de las anteriores. La categoría que da mejores resultados y en la que nos centraremos es la de los diccionarios, ya que existen una gran diversidad de métodos que permiten aumentar el tamaño de los conjuntos de datos de contraseñas reales con contraseñas ficticias siguiendo los mismos patrones y distribuciones.

Una de las principales variantes del ataque basado en diccionarios es el conocido como *Rainbow Attack* o *Pre-compute Dictionary Attack*. Este método se basa en que en la mayoría de las fugas de contraseñas no se filtra el texto plano, sino que se filtra el hash. Como ya se expuso en el apartado *2.1.* en la década de los 70 se propuso el uso de funciones *hash* para almacenar las contraseñas [15], lo que permite al sistema identificar si el hash de la contraseña que introduce el usuario coincide con el hash que tiene almacenado, pero no a la inversa. Es decir, si los *hash* son descubiertos por un atacante este no es capaz de recuperar la contraseña original, ya que se trata de un función teóricamente irreversible. En la práctica, mediante combinatoria los *hash* se pueden romper, pero para ello se necesita una capacidad de computacional altísima, muchas veces inalcanzable. Sin embargo, si el atacante compara los *hash* capturados con los hash preprocesados de diccionarios con miles de contraseñas, la probabilidad de adivinar pasa a ser mucho mayor. Esto es lo que se conoce como el *Rainbow Attack* del que se han realizado análisis detallados en [24] y [25]. Su principal ventaja es descifrar la contraseña con un tiempo computacional constante al basarse en funciones *hash* y no depender de la longitud de la contraseña como sucede en los ataques por fuerza bruta. La principal contramedida es añadir un valor aleatorio a la contraseña

antes de generar el hash (conocido como *salt*), pero que es necesario almacenar también vinculado al *hash*.

De las técnicas tradicionales para generar contraseñas a partir de conjuntos de contraseñas filtradas destacan las que se basan en *modelos de Márkov* [26], los cuales aprenden las contraseñas en base a unas reglas definidas como secuencias de caracteres que guardan ciertas dependencias entre sí y añadiendo cierto factor de aleatoriedad. Están basados en la propuesta de Shannon para modelar y predecir el lenguaje inglés con las *cadenas de Márkov* [27]. La capacidad de predicción depende de la complejidad de las contraseñas que se modela con la conocida como *Kolmogorov complexity* [28] y que se define como la longitud de la instrucción de la máquina de Turing más corta que la describe.

El siguiente avance en el estado del arte de las técnicas de generación de contraseñas vino de la mano de *Probabilistic Context-Free Grammars* (PCFGs) [29], un método que genera estructuras de contraseñas en orden probabilístico. Primero crea automáticamente una gramática probabilística libre de contexto basada en un conjunto de contraseñas reales y, a partir de estas, se generan las reglas. En el artículo original [29] consiguen descifrar entre un 28% y un 129% más de contraseñas que con las técnicas desarrolladas hasta la fecha de su publicación. Siguiendo el mismo planteamiento, los trabajos de [30] y [31], avanzaron en la mejora de los modelos estadísticos.

2.2.2. MÉTODOS CON TÉCNICAS DE APRENDIZAJE AUTOMÁTICO

El conjunto de métodos descritos hasta ahora se basa en los modelos estadísticos tradicionales, pero incluso antes de la publicación de PassGAN, se realizaron distintas propuestas de técnicas de aprendizaje automático para la predicción de contraseñas. Uno de los primeros artículos que trataban el tema es [32], donde se estudia la capacidad de redes neuronales sencillas para predecir las contraseñas. En el estudio se comparan redes *Single-Layer Perceptron* (SLP) y *Multi-Layer Perceptron* (MLP) entre 4 y 10 capas ocultas.

En *Guessability Using Neural Networks* [33] desarrollan una arquitectura con tres capas *Long Short-Term Memory* (LSTM) y dos capas recurrentes (RNN) densas. Realizan un estudio muy amplio de las distintas implementaciones, con una gran variedad de conjuntos de contraseñas y con una metodología muy bien definida. Además, demuestran que las redes neuronales pueden ser de gran utilidad para comprimir la información de los conjuntos de contraseñas hasta en cientos de kilobytes. A partir de estos resultados, implementan en JavaScript el modelo ejecutable en el lado del cliente y que permite analizar la resistencia de una contraseña a un ataque de

duración arbitraria con una latencia de menos de un segundo. Usa *WebWorker browser API* para ejecutar la red neuronal en un hilo único en el propio dispositivo del cliente.

Otra de las propuestas basada en redes LSTM son [34] y [35], donde definen el generador de contraseñas *GENPass*, una combinación de PCFG [29] con LSTM y *progressive learning* (PL). Los resultados obtenidos en el artículo son que el uso de PL para la solución con LSTM mejora la predicción, incluso al mezclar conjuntos de contraseñas de distintas fuentes, a diferencia de los métodos clásicos.

2.2.3. HERRAMIENTAS

En las secciones anteriores se han descrito métodos para la adivinación de contraseñas. Sin embargo, en la práctica la mayoría de usuarios no realizan sus propias implementaciones de los métodos, sino que usan herramientas que ya los aglutinan. En la plataforma *sectools.org* [36] realizan una clasificación de las herramientas de predicción de contraseñas más usadas actualmente. Las aplicaciones por orden de usuarios de mayor a menor en el momento que de su consulta son: *Aircrack, Cain and Abel, John the Ripper, THC Hydra, ophcrack* y *Medusa*.

Un ejemplo de la gran variedad de herramientas que se pueden encontrar es *Cupp* [37], una herramienta que genera diccionarios personalizados según patrones de la elaboración del perfil de un individuo, como el nombre de usuario, el nombre de la mascota, la fecha de nacimiento del cónyuge, etc.

Dentro del estado del arte de las publicaciones consultadas se recogen mayoritariamente dos herramientas: *HashCat* [38] y *John the Ripper* (JTR) [39]. Las dos herramientas son un conjunto de métodos dedicados a encontrar las contraseñas por fuerza bruta, ataques basados en diccionarios, modelos de Márkov, etc. Como se explicaba en la sección de métodos de adivinación clásicos es necesario definir reglas de patrones de contraseñas para los métodos más avanzados. En el caso de *HashCat* las reglas se pueden encontrar en [40] y para *John the Ripper* en [41].

2.3. REDES GENERATIVAS ANTAGÓNICAS

Las herramientas y métodos gráficos descritos en el apartado *2.2.*, aunque se han demostrado que presentan una gran tasa de éxito, son fruto de la definición de reglas *ad-hoc* sobre los patrones que siguen los usuarios para definir sus contraseñas. La creación de estas herramientas requiere una gran inversión de tiempo y su evolución o la diversificación de sus aplicaciones es muy poco escalable. En contraposición, con las técnicas de aprendizaje profundo se consigue un enfoque

completamente distinto de cómo alcanzar el problema, al eliminarse la necesidad de establecer unas reglas genéricas y delegar a la propia herramienta el aprendizaje automático sobre un conjunto de datos amplio de contraseñas filtradas.

La idea de usar técnicas de aprendizaje automático ya se introdujo en el *apartado 2.2.2.*, sin embargo, dichas aproximaciones no han tenido la repercusión que sí ha tenido PassGAN [1]. Su principal aportación es la introducción de las redes antagónicas generativas (GAN) dentro del ecosistema de predictores de contraseñas, consiguiendo según afirman sus autores, una mejorar de la capacidad de predicción de contraseñas de las herramientas tradicionales como *HashCat* entre un 51% y 73%.

Las GAN, propuestas por primera vez por *Goodfellow et al.* en [42], definen una arquitectura donde hay un bloque que produce candidatos, el generador G, y otra que los evalúa e intenta diferenciarlos de los de un conjunto de datos real, el discriminante D. Para la definición de la arquitectura y el protocolo de entrenamiento existe una gran variedad de alternativas que dependen de la aplicación. PassGAN se diseñó basándose en la arquitectura *Wasserstein*, en concreto en la mejora del modelo propuesta en [43].

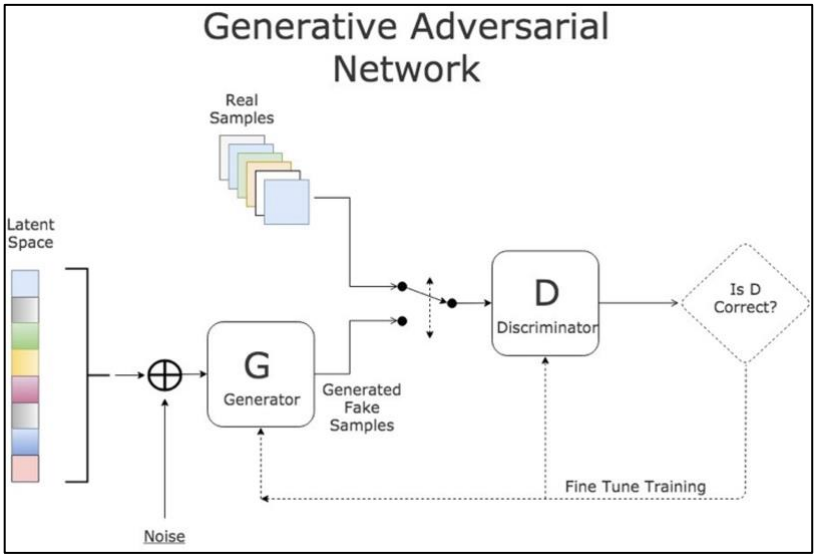

Figura 3: Arquitectura genérica de una GAN [2]

[2] Imagen tomada del artículo [42]

2.3.1. PASSGAN: ARTÍCULOS POSTERIORES

En este apartado se recopilan un conjunto de artículos posteriores a la publicación de [1] donde se analiza el artículo original y se proponen mejoras y diseños alternativos. Se han encontrado publicaciones en las que no han conseguido realizar una predicción fiable con conjuntos de datos distintos a los usados en el artículo original, como en [44] donde solo consiguieron predecir un 6,19% de las contraseñas de un conjunto amplio de 254 millones de cuentas de correo rusas. Sin embargo, la gran mayoría consiguen mejorar los resultados del artículo original.

Entre los artículos que se centran en analizar y evaluar los retos que genera la incorporación de PassGAN al conjunto de herramientas de predicción de contraseñas se puede citar a [22] o [45] que consiguen una mejora de entre un 5% y 10% realizando ajustes en la arquitectura de [1]. No obstante, existen muchos otros que proponen distintas alternativas:

- [46] evalúa el comportamiento de PassGAN frente a distintos conjuntos de datos, comparándolo además contra el sistema *zxcvbn* [47], un estimador de contraseñas de bajo coste computacional desarrollado por los ingenieros de Dropbox.

- [48] basándose en que los modelos GAN sufren problemas de generación debido a la naturaleza discreta de las contraseñas proponen dos arquitecturas alternativas: GS-PassGAN que utiliza la relajación Gumbel-Softmax y el S-PassGAN que utiliza una representación suave de una contraseña real obtenida por un autoencoder adicional.

- [49] realiza un estudio de las distintas técnicas de aprendizaje profundo aplicables a la predicción de contraseñas y proponen un nuevo modelo, GNPassGAN, donde consiguen una mejora del 88,03% en la adivinación de contraseñas y generan un 31,69% menos de duplicados frente a PassGAN.

- [50] propone dos enfoques para la predicción de contraseñas, uno no probabilístico que se centra en los sesgos arbitrarios de las contraseñas y otro probabilístico que no solo pone el foco en la generación de posibles contraseñas, sino que estima su distribución. En el artículo hacen un análisis de PassGAN, proponiendo mejoras capaces de duplicar la probabilidad de predicción de contraseñas. También, analiza otras alternativas como el uso de autoencoders, redes neuronales compuestas por dos elementos, un encoder que comprime la información y un decoder que la descomprime, también para generar contraseñas. El modelo propuesto lo bautizan como Context Wasserstein Autoencoder (CWAE) y su arquitectura está basada en Wasserstein Autoencoder (WAE) [51] con la

regularización mediante la correspondencia de momentos aplicada al espacio latente (WAE-MMD) [51].

- [52] se centra en construir un modelo ligero que pueda reducir el tiempo necesario para su entrenamiento pero que siga manteniendo la misma capacidad predictiva de PassGAN. Su propuesta es VAEPass, un modelo basado en un Variational Auto-Encoder (VAE) que consta de un codificador y un decodificador establecidos mediante una Gated Convolutional Neural Network (GCNN). Sus resultados son una mejora de entre un 2,7% y 9,3% frente a los resultados de PassGAN. Además, siguiendo sus objetivos de implementar un modelo ligero, consiguen que los parámetros de VAEPass sean aproximadamente el 32% de los de PassGAN y el tiempo de entrenamiento requerido sea aproximadamente el 11% del requerido por PassGAN.

2.3.2. PASSGAN: APLICACIONES

Honeywords

Honeywords es un mecanismo que tiene el objetivo de detectar fugas de información lo antes posible. El sistema crea contraseñas falsas (*honeywords*) para cada cuenta de usuario y las mezcla con las contraseñas reales (*sugarwords*). Cuando un atacante consigue una filtración de contraseñas este tiene que ser capaz de distinguir cuáles son las contraseñas reales de las falsas. Además, si el administrador del sistema detecta que se están introduciendo contraseñas falsas puede identificar rápidamente que se ha producido una fuga de información. Aunque a priori el planteamiento teórico parece trivial, su implementación en la práctica no lo es tanto. Por un lado, el sistema tiene que ser capaz de generar *honeywords* que se parezcan lo máximo posible a las *sugarwords*, ya que si esto no es así el atacante podría distinguir unas de otras y solo atacar con las *sugarwords*. Por otro lado, la arquitectura del sistema es de vital importancia ya que, si el atacante tiene acceso tanto al servidor de contraseñas con las *honeywords* y *sugarwords* como al sistema encargado de distinguir las reales de las falsas, el *Honeychecker*, el atacante de igual forma podría distinguir entre unas y otras.

En [53] simulan un juego entre el atacante y el defensor para evaluar las posibles estrategias seguidas por cada uno. El defensor utiliza 5 estrategias distintas de generación de *honeywords* basadas en PassGAN, mientras que el atacante emplea 3 estrategias de adivinación, dos de ellas basadas en PassGAN y una tercera basada en Top-PW ataque introducido por Wang et al. [54]. Top-PW se basa en que las contraseñas originales cumplen con la *Zipf's law* [55], es decir, que

las contraseñas reales no seguirán una distribución uniforme, sino que habrá contraseñas mucho más comunes que otras y, sin embargo, la distribución de las *honeywords* generadas (si no se establecen reglas específicas adicionales) será uniforme.

También en [56] se estudian distintas aproximaciones para evitar el problema del ataque Top-PW combinando PassGAN con PCFG y métodos basados en aleatoriedad a la generación de *honeywords* que sigan una distribución de probabilidad real.

Verificador de seguridad de contraseñas

Como ya se ha ido apuntando a lo largo del estado del arte, los esquemas de autentificación basados en contraseñas son seguros siempre y cuando la adivinación de la contraseña sea compleja. Por ello, una de las aplicaciones de los predictores de contraseñas puede ser un verificador o evaluador de contraseñas que sea capaz determinar en que porcentaje una contraseña puede ser predecible por un atacante. En la práctica la mayoría de los sistemas que utilizan contraseñas como método de autentificación usan sistemas basados en reglas muy simples como, por ejemplo, verificar simplemente que contenga números, letras y caracteres especiales. Sin embargo, el estado del arte actual de métodos de predicción de contraseñas es capaz de romper estas reglas con cierta facilidad. En consecuencia, la integración de los métodos actuales de predicción dentro de los sistemas de verificación de contraseña seguras es de vital importancia.

En este ámbito se han desarrollado múltiples investigaciones como en *Ciaramella et al.* [32] donde usan redes neuronales SLP y MLP o el sistema *zxcvbn* [47] que calcular una nota de lo segura que es una contraseña. También es necesario hacer referencia a [33], explicado ya en 0, donde usan LSTMs implementadas con *JavaScript* en el cliente para evaluar la predictibilidad de la contraseña. No se ha encontrado desarrollos de aplicaciones directas de PassGAN en este ámbito, aunque sí hay artículos que hacen referencian a lo útil que puede ser en este aspecto.

Ataques focalizados

Otra de las posibles líneas de aplicación es el desarrollo de predictores de contraseñas focalizados a un usuario en concreto. Lo cierto es que la mayoría de los usuarios tienden a elegir contraseñas relacionadas con sus datos privados, como cumpleaños, el nombre de su mascota o las iniciales de su nombre. Debido a que es una práctica muy extendida, es relevante el estudio de ataques de adivinación que se basan en los datos disponibles de un usuario en concreto para generar un conjunto de las contraseñas más probables. Como apuntan en [53] es relevante para el

diseño de las *honeywords* y ser capaz de identificar si el acceso al sistema es una contraseña introducida por el usuario o por un atacante con listas de contraseñas genéricas.

3. ANÁLISIS TEÓRICO DE LOS MODELOS

En esté capitulo se analiza como son los modelos teórico en los que se basa *PassGAN* y a su vez los que se basan en él. Se estudia como son sus diferencias en cuanto a funciones de pérdidas, metodología de entrenamiento y distante medidas para la normalización.

Todos los modelos analizados nacen a raíz de estudios con un mismo hilo conductor, el conseguir estabilizar el entrenamiento de las GAN. Uno de los principales retos de las GAN especialmente hablando de la generación de texto es conseguir equilibrar el proceso de entrenamiento poniendo en igualdad de condiciones al generador y al discriminante. Es de vital importancia, ya que si está lucha no se encuentra equilibrada y siempre gana uno de los dos no aprenderán porque cada uno necesita que el otro mejore para mejorar el.

En primer lugar, se estudian los modelo en los que se basa PassGAN por orden cronológico. Empezando por WGAN el modelo base donde se introdujo el uso de la distancia *Wasserstein*. Después se analiza IWGAN (*Improve – WGAN*) un modelo que propone una nueva función de pérdidas y WGAN-GP con penalización del gradiente. El modelo WGAN-GP es en el que se basa PassGAN, utilizando su misma arquitectura y método de entrenamiento.

En segundo lugar, se analizan dos modelos posteriores a PassGAN: GNPassGAN y GS-PassGAN. El primero modelo, GNPassGAN, propone una nueva normalización del gradiente más restrictiva, un nuevo proceso de entrenamiento y una función de activación y pérdidas distinta. El segundo modelo, GS-PassGAN, se centra en abordar el problema de la naturaleza discreta de la codificación de las contraseñas en el entrenamiento.

3.1. WGAN

Para resolver el problema de la estabilización del entrenamiento se propone el modelo WGAN en [57]. Su principal mejora con respecto al GAN tradicionales es el uso de la

distancia de *Wasserstein* para medir la diferencia entre la distribución de probabilidad de las muestras generadas y la distribución real de los datos. La distancia de Wasserstein, también conocida como distancia de *Earth Mover (EMD)*, proporciona una métrica más significativa y estable para cuantificar la disparidad entre las distribuciones.

La distancia de Wasserstein entre dos distribuciones de probabilidad $\boldsymbol{P}$ y $\boldsymbol{Q}$ se define como:

$$\boldsymbol{W(P,Q) = \inf_{\gamma} \int c(x,y)\, d\gamma(x,y)}$$

donde $\boldsymbol{\gamma}$ es una medida de probabilidad en el espacio de muestra $\mathbf{X} \times \mathbf{Y}$, $\boldsymbol{c(x,y)}$ es una función de coste que mide el coste de transportar una unidad de masa desde $\boldsymbol{x}$ a $\boldsymbol{y}$, y el $\boldsymbol{\inf_{\gamma}}$ significa que se toman todas las medidas de probabilidad $\boldsymbol{\gamma}$ que tienen $\boldsymbol{P}$ y $\boldsymbol{Q}$ como marginales.

La función de pérdidas se define como:

$$\boldsymbol{L(D_W, G) = E[D_W(x)] - E\left[D_W\left(G(z)\right)\right]}$$

donde $\boldsymbol{G(z)}$ es la muestra generada por el generador a partir de un vector de ruido $\boldsymbol{z}$, $\boldsymbol{x}$ es una muestra real de los datos, y $\mathbf{E}[\,]$ es la esperanza. El objetivo es minimizar esta función de pérdida con respecto a los parámetros del generador $\boldsymbol{G}$ y maximizarla con respecto a los parámetros del discriminador $\boldsymbol{D_W}$.

3.2. IWGAN

El modelo *Wasserstein GAN* (WGAN) utiliza la distancia de *Wasserstein* para evitar las limitaciones en el entrenamiento de GANs, pero tiene otros defectos como el "colapso de modos" y la falta de una métrica para detectar la convergencia. Se conoce como "colapso de modos" cuando el generador produce una variedad limitada de salidas, ignorando u omitiendo la diversidad en los datos reales.

En [43] se introduce un nuevo modelo llamado *Improved Wasserstein GAN* (IWGAN). La principal diferencia que distingue a IWGAN de WGAN es la introducción de un término de regularización en la función de pérdida del generador.

En IWGAN, se utiliza una función de pérdida mejorada para el generador que incluye un término adicional de regularización. Este término se basa en la divergencia de Jensen-Shannon (JS) entre la distribución generada y una distribución de referencia, como la distribución de datos reales.

La función de pérdida del generador en IWGAN se define como:

$$[L_G = -E[\log(D(x))] + E\left[\log\left(D(G(z))\right)\right] + \lambda E[JS(P_r, P_g)]]$$

Donde(E) denota el valor esperado, ($D(x)$)es la salida del discriminador para una muestra real (x), ($G(z)$) es la muestra generada por el generador a partir de un vector de ruido (z), (P_r) es la distribución de referencia (distribución de datos reales), (P_g) es la distribución generada por el generador, (λ) es un parámetro de regularización y (JS) representa la divergencia de Jensen-Shannon.

Al introducir este término de regularización basado en la divergencia JS, IWGAN busca mejorar la estabilidad y la calidad de las muestras generadas, promoviendo una mejor correspondencia entre las distribuciones generada y de referencia.

3.3. WGAN-GP

Como ya se ha anticipado en los apartados anteriores en el contexto de la GAN es de vital importancia la estabilidad y la convergencia del modelo. En este apartado se analiza con detalle el modelo bautizado como WGAN-GP [58] y que propone una penalización del gradiente conocida como la restricción de Lipschitz. La restricción de Lipschitz se refiere a limitar la magnitud de las derivadas parciales del discriminador en relación con las entradas.

En términos matemáticos, la restricción de Lipschitz se puede expresar de la siguiente manera:

$$[|\nabla \mathcal{D}_W(x)| \leq K]$$

Donde ($|\nabla \mathcal{D}_W(x)|$) representa la magnitud del gradiente del discriminador en un punto de entrada (x), y (K) es una constante que limita dicha magnitud.

En el entrenamiento de la WGAN, se busca ajustar los pesos del discriminador de manera que se cumpla esta restricción de Lipschitz. Esto se logra mediante la penalización del gradiente, que se añade a la función de pérdida. La penalización del gradiente ayuda a suavizar las funciones de activación y limitar las derivadas parciales del discriminador, lo cual promueve una mayor estabilidad en el entrenamiento.

Para garantizar que D_W satisfaga la condición de Lipschitz con una constante K, se utiliza la penalización de gradiente. La función de pérdida actualizada con la penalización de gradiente se denomina WGAN-GP.

La penalización de gradiente se aplica agregando un término adicional a la función de pérdida del WGAN. En la formulación original del WGAN-GP, este término se calcula mediante una media entre muestras reales y generadas. Se penaliza cuando el gradiente de la función discriminador D_W excede un umbral. La función de pérdida queda:

$$L_{\text{WGAN-GP}}(D_W, G) = E[D_W(x)] - E\left[D_W\left(G(z)\right)\right] + \lambda E[(|\nabla_{\tilde{x}} D_W(\tilde{x})|_2 - 1)^2]$$

donde x es una muestra real de los datos, z es un vector de ruido de entrada al generador G, ($D_W(x)$) es la salida del discriminador para una muestra real x, ($D_W\left(G(z)\right)$) es la salida del discriminador para una muestra generada ($G(z)$), ($\tilde{x}$) es una interpolación entre (x) y ($G(z)$) definida como ($\tilde{x} = \epsilon x + (1 - \epsilon)G(z)$) con ($\epsilon$) muestreado de una distribución uniforme entre 0 y 1, (λ) es un hiperparámetro que controla la importancia relativa de la penalización de gradiente, ($|\cdot|_2$) es la norma (L_2) y ($\nabla_{\tilde{x}}$) es el gradiente con respecto a ($\tilde{x}$).

PassGAN se basa la arquitectura WGAN-GP, utilizando las técnicas de penalización de gradiente para lograr una función continua 1-Lipschitz descrita en el *apartado 3.3*. No utiliza ninguna implementación novedosa ni del gradiente ni del entrenamiento ni de la arquitectura.

3.4. GNPASSGAN

GNPassGAN es una variante de PassGAN que incorpora varias propuestas para mejorar también la estabilidad y el rendimiento del entrenamiento del modelo GAN para texto como los modelos de los apartados *3.2.* y *3.3.* Los aspectos innovadores respecto a los anteriores de este modelo son:

3.4.1. Normalización de gradientes

En GNPassGAN se agrega la normalización de gradientes propuesta en [59] y bautizada como *Adaptive Gradient Normalization* (AGN). Al normalizar los gradientes, se ayuda a mitigar el problema de los gradientes que crecen exponencialmente o se anulan, lo cual dificulta el entrenamiento. Esto se logra mediante el cálculo de un factor de normalización adaptativo para cada gradiente parcial (derivada parcial de la función objetivo con respecto a un parámetro concreto del modelo). El factor de normalización adaptativo para un gradiente parcial (∇) se define como:

$$\mathcal{N} = \frac{|\nabla|_2}{\max(|\nabla|_2, \epsilon)}$$

Donde ($|\nabla|_2$) es la norma euclídea del gradiente y (ϵ) es una pequeña constante positiva para evitar divisiones por cero. El gradiente normalizado se obtiene multiplicando el gradiente parcial (∇) por el factor de normalización adaptativo:

$$\nabla_{\text{norm}} = \mathcal{N} \cdot \nabla$$

El gradiente normalizado (∇_{norm}) se utiliza entonces para actualizar los parámetros del modelo durante el proceso de optimización. La idea detrás de la normalización del gradiente adaptativa es asegurar que los gradientes tengan magnitudes similares y evitar que algunos gradientes dominen sobre otros.

3.4.2. Función de activación

En GNPassGAN, se modifica la función de activación en la última capa del generador, pasando de una *softmax* a una *tanh*. La función de activación *softmax* se utiliza comúnmente en tareas de clasificación binaria [0,1], sin embargo, *tanh* aplana la salida

del generador al rango [-1, 1]. El motivo de por qué se cambia la función de activación no lo explican en el artículo donde se publicó [49], pero se intuye que es por la función de perdida escogida.

3.4.3. Función de pérdida

En lugar de utilizar la pérdida de *Wasserstein* como en PassGAN, GNPassGAN emplea la pérdida de entropía binaria dentro de una capa sigmoide como función de pérdida. La pérdida de entropía binaria es una función de pérdida comúnmente utilizada en tareas de clasificación binaria. Al utilizar esta función de pérdida, GNPassGAN busca optimizar el generador y el discriminador para la clasificación binaria, específicamente para distinguir entre contraseñas reales y contraseñas generadas. La capa sigmoide en la función de pérdida asegura que la salida esté limitada entre 0 y 1, representando la probabilidad de que la contraseña generada sea real.

3.4.4. Análisis del entrenamiento

Cada iteración del bucle principal implica actualizar tanto el generador como el discriminador. En cada iteración, se selecciona un lote de ejemplos de entrenamiento y se realiza una actualización del discriminador, seguida de una actualización del generador. Por lo tanto, el número de veces que se actualizan el discriminador y el generador depende de la cantidad de iteraciones sobre la actualización del generador y del número de veces que se actualiza el discriminante por cada actualización del generador. Estos valores determinan cuántas veces se actualizan los parámetros del discriminador y el generador respectivamente en cada iteración del bucle de entrenamiento.

Otro de los parámetros importantes a la hora de entrenar GNPassGAN es el tamaño del lote (*batch size*). Esté es la cantidad de ejemplos de entrenamiento que se procesan en paralelo antes de que se realice una actualización de los parámetros del generador y el discriminador. El uso de un tamaño de lote más grande puede acelerar el proceso de entrenamiento al permitir cálculos en paralelo, pero también puede requerir más memoria. Por otro lado, un tamaño de lote más pequeño puede ser útil si existen limitaciones de memoria o si se desea una actualización más frecuente de los parámetros del modelo. Otro de los aspectos importantes en los que puede afectar el tamaño del lote es en el ruido con el que aprende el modelo. Un tamaño de lote pequeño puede provocar que el modelo

aprenda más rápido porque usa una menor cantidad de datos para esperarse a actualizar los pesos. Sin embargo, esto no significa que aprenda en la dirección correcta, al actualizar sus parámetros basados en un volumen pequeño de datos la probabilidad de equivocarse es mayor. En el caso de escoger un tamaño de lote muy grande pasa lo contrario, se basa la actualización de los parámetros del modelo en muchos datos por lo que se avanza en una dirección mucho más correcta. En contraposición, en este caso, el entrenamiento se vuelve muy lento y el coste de converger en una solución óptima es muy elevado.

El proceso en detalle del entrenamiento es el siguiente:

Entrenamiento del discriminante

En primer lugar, se entrena tantas veces como el parámetro *critic_iters* por iteración. Por defecto tiene un valor de 10 y en cada ejecución se lanzan los siguientes pasos:

1. Primero toma del conjunto de datos de entrenamiento una palabra y la convierte a codificación *One Hot* de tamaño *[batch_size,-1, len(charmap)]*. Siendo el *charmap* el conjunto de caracteres distintos del conjunto de datos de contraseñas y la codificación *One Hot* una representación numérica en la que cada carácter se codifica como un vector binario con una dimensión igual al tamaño del vocabulario.
2. Inicializa a cero los gradientes del discriminante.
3. Genera ruido de dimensión *[batch_size, 128]* y se lo pasa al generador como entrada. Llama la atención que la segunda dimensión sea fija de 128 y no esté parametrizada, que no dependa de la longitud del *charmap*.
4. El ruido lo usa de la entrada del generador y guarda la salida como contraseña **falsa**.
5. Ejecuta la función *normalize_gradient(netC, x)* pasándole la clase de Python del Discriminante y la contraseña **real** ya formateada. Realiza lo mismo, pero pasando le una contraseña **falsa** generada.
6. Calcula la función de perdida para la contraseña real y la ficticia.

   a. La función de perdida usada es *BCEWithLogitsLoss()*. La función combina una *Sigmoid* con una *BCELoss* (Binary Cross Entropy). Su explicación se desarrolla en el apartado *4.5.1*.
   b. La función de pérdidas de la contraseña real la calcula respecto a un vector de unos y la función de pérdidas de la contraseña ficticia la calculas respecto al 0.
7. El resultado de las dos funciones de perdida los suma.
8. Propaga los pesos ejecutando *loss.backward()* y ejecuta el optimizador *optimizerC.step()*. El optimizador del discriminante se ha creado con los siguientes parámetros:

```
optimizerC = optim.Adam(netC.parameters(), lr=1e-4, betas=(0.5, 0.9))
```

Entrenamiento del generador

Solo se ejecuta una vez por iteración.

1. Inicializa a cero los gradientes del generador.
2. Se vuelve a generar ruido y se le pasa al generador.
3. Con la salida del generador se le pasa a la función *normalize_gradient(netC, x)* pasándole la clase de Python del Discriminante.
4. Con el resultado se calcula la función de pérdidas para que la salida sean unos.
5. Propaga los pesos ejecutando *loss.backward()* y ejecuta el optimizador *optimizerG.step()*, el del generador.

Métricas tomadas durante el entrenamiento

Para cada iteración de entrenamiento:

1. Guarda el tiempo de entrenar el discriminante y generador
2. Guarda las pérdidas de la última iteración de las 10 que hace de entrenar el discriminante.
3. Guarda las pérdidas de entrenar el generador.

Cada 100 iteraciones:

1. Evalúa la distribución de los caracteres generados y lo imprime en los logs y por terminal.

3.5. GS-PASSGAN

Por último, el modelo GS-PassGAN propuesto en el artículo [48], se centra en un problema distinto al descrito en los anteriores. Implementa la relajación Gumbel-Softmax para abordar el problema de la naturaleza discreta de las contraseñas en el entrenamiento.

En PassGAN, las contraseñas reales se representan como secuencias de representaciones *one-hot* de sus caracteres. Esto es un tipo de codificación básica que codifica una contraseña como una matriz binaria en la que cada fila le corresponde un carácter de la contraseña y una cada columna por cada posible carácter. La columna tendrá un valor de 1 si es la correspondiente con la del carácter. Sin embargo, al generar contraseñas falsas (por el generador de la GAN) las salidas son las de una capa softmax, no son números enteros, y el discriminante puede aprenderse estas características para distinguir entre las contraseñas reales y falsas.

Para superar esta limitación, GS-PassGAN utiliza la relajación Gumbel-Softmax. Esta técnica permite en lugar de muestrear directamente las contraseñas falsas utilizando la capa softmax, se utiliza una aproximación basada en la distribución Gumbel con la función softmax.

En la siguiente ecuación, las salidas del generador se denotan como $\boldsymbol{h_j}$, donde $\boldsymbol{j}$ representa el carácter en la contraseña generada. En lugar de muestrear directamente $\boldsymbol{y_j}$, la representación *one-hot*, se utiliza la siguiente aproximación diferenciable:

$$\boldsymbol{y_j = softmax\left(\frac{1}{\tau(h_j + g)}\right)}$$

En esta ecuación: ***softmax*** es la función que calcula las probabilidades de cada carácter, $\boldsymbol{\tau}$ es un parámetro que controla la suavidad de la distribución resultante, y $\boldsymbol{g}$ es

un vector de ruido independientes que sigue una distribución *Gumbel* con media cero y escala unitaria.

4. ANÁLISIS EMPÍRICO

En este capítulo se lleva a cabo una evaluación exhaustiva de la implementación de los modelos desarrollados. Se examinan diferentes aspectos relacionados con el entrenamiento y generación de contraseñas utilizando GNPassGAN. Además, se analiza el rendimiento de las contraseñas generadas y se comparará su precisión con bases de datos reales, como RockYou [18] y Pwned [60]. También se exploran otros factores, como la proporción de contraseñas únicas, las diferencias entre las contraseñas generadas en cada iteración, la distribución de las longitudes de las contraseñas, la proporción de números y letras, y finalmente, el rendimiento de GNPassGAN con un léxico real. A través de este análisis, se busca obtener una visión completa y detallada de las capacidades y características del modelo GNPassGAN en términos de generación de contraseñas.

4.1. EVALUACIÓN DE LOS REPOSITORIOS DE CÓDIGO

El objetivo de este apartado es analizar el estado del arte de los repositorios de código existentes relativos a PassGAN y las publicaciones posteriores relacionadas. Para alcanzarlo se ha realizado una búsqueda de los repositorios publicados clasificándolos según distintos factores:

- El sistema operativo y entorno en el que se basan.
- La versión de Python usada.
- La versión de *TensorFlow* [61] o *PyTorch* [62] con la que se ha implementado el modelo.
- En que artículo se basan.
- La similitud del repositorio respecto a la arquitectura de PassGAN descrita en [18].

4.1.1. REPOSITORIOS DE CÓDIGO

En la siguiente tabla se resume todos los repositorios analizados junto con los artículos relacionados con PassGAN descritos en el estado del arte que proponen mejoras y

arquitecturas relacionadas. Nótese que no se han encontrado repositorios para todos los artículos y que hay artículos para los que se han encontrado múltiples repositorios.

La tabla está ordenada según la columna de "Similitud original", columna que ordena por similitud de cada modelo con respecto el artículo original. La similitud se ha establecido, en primer lugar, considerando los artículos asociados a cada repositorio de código: que diferencian a su arquitectura respecto a la original, funciones de perdidas, etc. En segundo lugar, se ha analizado el código prestando detalle si se han usado clases y una estructura similar a la del repositorio original. En tercer lugar, se ha usado como criterio diferenciador la versión de Python y de PyTorch o TensorFlow utilizada, cuanto más cercana a la de PassGAN original más similar.

GitHub Link Resumen	Autor	OS	Python	PyTorch / TensorFlow	Similitud original	Referencia Paper Anexo
PassGAN	brannondorsey	.	2	TF 1.4.1	1	[1]
PassGAN	d4ichi	.	3.x	TF 1.13	2	
PassGAN	rarecoil	.	3.x	TF 1.15	3	
PassGAN	Karlosse	.	3.x	TF 2	4	
PassGAN	Riathoir	Ubuntu 19.10+ o Docker	3.7	TF 2.1	5	
PassGAN	r-khanna	Google Colab	3.x	PT 1.4	6	
PassGAN Based Honeywords	.	.	.	.	7	[53]
PassGAN con PCFG	.	.	.	.	8	[35]
PLR	pasquini-dario	.	3.x	TF 1.14	9	[50]
GNPassGAN	fangyiyu	.	3.x	PT 1.10	10	[49]
Adversarial Password Cracking	.	.	.	.	11	[46]
GS-PassGAN y S- PassGAN	.	.	.	.	12	[48]
VAEPass	.	.	.	.	13	[52]
LSTM con RNN	cupslab	Docker	3.x	TF 1.4	14	[34]
Improved WGAN Training	igul222	.	3.x	TF	15	[43]
WGAN PT	kzkadc	.	3.x	PT	16	
wgan_pytorch	Alexrgg	MacOs	3.x	PT	17	.

Figura 4: Estado del arte de modelos y repositorios relativos a PassGAN.

El siguiente paso ha sido probar el código de los repositorios. En cuanto a los códigos que se basan directamente en PassGAN se han realizado las siguientes observaciones:

- **PassGAN – brannondorsey**: Este repositorio cuenta con unos *requiremnts.txt* bien definidos. La principal problemática has sido que el código está basado en TensorFlow 1 y Python 2 y para que ejecute correctamente se han tenido que instalar versiones de las librerías auxiliares bastante antiguas.
- **PRL - pasquini-dario:** este código es una versión más actualizada que el anterior usando Python 3, pero no se encontraban documentadas las versiones de las librerías usadas por lo que se han tenido que ir ajustando las versiones de las librerías para corregir los errores de ejecución. Para ejecutarlo de manera nativa *TensorFlow* en el ordenador con arquitectura AMD se han seguido los pasos descritos en el apartado del ANEXO *A1.1*. También, se ha considerado el uso de *Google Colab* como entorno alternativo para la ejecución. Sin embarga, no ha sido viable su ejecutar con éxito, al tratarse de *Tensorflow 1 Google Colab* lo considera obsoleto y no permite instalarlo.

- **PASSGAN-IWGAN-Tensorflow-2:** Se ha probado en un entorno con *Tensorflow 2* nativo para AMD. Para ejecutarlo correctamente se han tenido que realizar modificaciones sobre algunas funciones del script *train.py* para adaptar la versión de *Tensorflow 2* usada por el autor a una más actualizada.

- **GNPassGAN:** En este caso usan una versión de *Pytorch* relativamente moderna, la 1.10 y tienen un *requirements.txt* con las versiones de las librerías utilizadas bien documentado. El código no estaba completamente adaptado para ejecutarse sin GPU por lo que se han tenido que modificar para que sea compatible con una ejecución con y sin GPU tanto en un entorno Intel como AMD.

- **PassGAN – PT Notebook:** El código se ha podido ejecutar correctamente sobre el entorno de *Google Colab*, sin embargo, se han identificado múltiples errores en el código y su incompatibilidad cuando el número de caracteres del conjunto de datos de entrada cambiaba. El entrenamiento es más rápido que en *GNPassGAN* pero no consigue buenos resultados, el modelo tiene bajas tasas de precisión (*accuracy*) y el código no está asociado a ninguna publicación.

- **PassGAN – Karlosse:** El resultado de la implementación ha sido muy similar a *PassGAN – PT Notebook*, se consigue ejecutar, pero los aciertos conseguidos son muy escasos en comparación a los que se consiguen con el artículo original de *PassGAN*.

No solo se han analizado modelos inspirados en *PassGAN* si no también los modelos en los que se inspira *PassGAN*:

- **WGAN – PT:** En este caso si está basado en código en *PyTorch*. Se ha conseguido ejecuta correctamente, pero la arquitectura difiere de la usada en el artículo original.
- **Improved WGAN Training:** Se ejecuta correctamente tras realizar modificaciones en el código, pero usa un *dataset* diferente a los usados para las contraseñas. Ene esté caso el código está basado también en una versión antigua de *TesorFlow* 1 y *Python* 2. A raíz de este código he realizado el repositorio *wgan_pytorch* con una implementación actualizada basada en él, pero con *PyTorch*.

4.1.2. CONCLUSIONES DE LAS IMPLEMENTACIONES

Una vez analizado el estado del arte de la implementación de los modelos disponibles, se pueden extraer varias conclusiones:

- En la mayoría de los casos no están publicados los códigos en los que se basan los artículos. Y de los que si hay código publicado casi ninguno se realizó por los autores, si no por implementaciones de particulares a posteriori.
- La documentación de los repositorios es escasa. Aunque cuentan con README.md no suelen describir el entorno en el que lo implementaron y las versiones específicas de las librerías usadas mediante un *requirements.txt*, *environment.yml* o similares.
- Debido a que las dos librerías de *deep learning* usadas (*TensorFlow* y *PyTorch*) han sufrido y siguen sufriendo constantes evoluciones de sus versiones es necesario una constante adaptación del código. A pesar de que siempre se puede trabajar con entornos de *Python* donde con versiones antiguas de las librerías locales, en el caso de *PassGAN* es un problema relevante ya que está basado en las primeras versiones de *Tensorflow 1*. La forma de estructurar tanto la arquitectura como el entrenamiento difiere bastante de como se hace en las últimas versiones y el uso de *Python 2* lo complica aún más.

En general se puede afirmar que a pesar de tratarse de un proyecto basado en *software* la replicabilidad de los experimentos es compleja por los motivos enumerados anteriormente. Para asegurar la estandarización de los experimentos se decide partir del código *GNPassGAN* para evaluar la propuesta de usar *GANs* para la generación de contraseñas.

4.2. METODOLOGÍA DE ENTRENAMIENTO DE GNPASSGAN

4.2.1. PARAMETRIZACIÓN DEL ENTRENAMIENTO

En el proceso de entrenamiento, se ha guardado el modelo en diferentes puntos, realizando el guardado cada 10.000 iteraciones de entrenamiento, empezando por 10.000, y continuado por 50.000 hasta 250.000 iteraciones. Esta estrategia de guardado periódico permite tener copias del modelo en distintos momentos del entrenamiento, lo que facilita su recuperación en caso de interrupciones y el posterior análisis del modelo entrenado para distintas iteraciones.

El modelo se ha entrenado utilizando el conjunto de contraseñas filtradas llamado *RockYou* [18]. Este conjunto se dividió en un 75% para el conjunto de entrenamiento y un 25% para el conjunto de test. La selección de ejemplos para cada conjunto se ha realizado de forma aleatoria. En total, el conjunto de contraseñas contiene 14.344.391 registros, estructurados con una contraseña por línea y con el carácter final '\n'.

El tamaño del lote (*batch size*) utilizado en el entrenamiento es de 64, lo que significa que en cada iteración se procesan 64 pasadas sobre el generador. Además, se ha establecido que el valor de *critic_iters* sea 10, lo que implica que se realizan 10 actualizaciones de los parámetros del discriminador por cada iteración del generador. En consecuencia, en cada iteración se utilizan un total de 64 * 10 = 640 contraseñas para el entrenamiento.

Teniendo en cuenta que el conjunto de contraseñas tiene un total de 14.344.391 registros, y considerando que el conjunto de entrenamiento representa el 75% del total, se utilizan aproximadamente 14.344.391 * 0.75 / (64 * 10) ≈ 17.000 iteraciones para recorrer completamente el dataset durante el entrenamiento. Es importante mencionar que este cálculo depende de la proporción del conjunto de entrenamiento, del tamaño del lote y el número de actualizaciones del discriminador escogidas en este caso en particular.

En cuanto a la longitud de las contraseñas se ha establecido una longitud máxima de 12 caracteres. Cuando una contraseña excede esta longitud, la contraseña no se usa para el entrenamiento. Una alternativa podría ser seleccionar los primeros caracteres hasta

alcanzar la longitud máxima fijada, descartando el resto de la contraseña, pero en este caso estaríamos entrenando el modelo con contraseñas que no son completamente reales.

4.2.2. METODOLOGÍA DE GENERACIÓN

Una vez entrenado el modelo el objetivo es cargar únicamente el generador y usarlo para crear un nuevo conjunto de contraseñas. En el proceso de generación se ha hecho uso de los modelos guardados durante el entrenamiento en intervalos regulares de 50.000 iteraciones. Estos modelos guardados se han generado en iteraciones específicas, comenzando por la iteración 10.000 y continuando cada 50.000 iteraciones hasta alcanzar las 250.000 iteraciones.

Para cada uno de estos modelos guardados, se han generado un total de 10^8 contraseñas en cada número de iteración distinto analizados. Esto significa que se han producido 600 millones de contraseñas (10^8 por cada número de iteración distinto analizado), una cantidad masiva que permite explorar una amplia variedad de opciones y patrones durante el proceso de generación.

Además, cabe destacar que los parámetros restantes utilizados en la generación de contraseñas son los mismos que se emplearon durante el entrenamiento. Esto incluye la longitud máxima de las contraseñas, que se ha establecido previamente. Por lo tanto, las contraseñas generadas seguirán teniendo una longitud máxima específica y se aplicará la misma lógica de truncado en caso de que alguna contraseña generada supere esa longitud.

4.3. ANÁLISIS DEL ENTRENAMIENTO DE GNPASSGAN

En este apartado se estudia cómo han evolucionado dos métricas distintas a lo largo del entrenamiento: las pérdidas del generador y discriminante y la divergencia de Jensen-Shannon para n-grams de hasta 4 caracteres.

4.3.1. EVOLUCIÓN DE PÉRDIDAS

Antes de entrar en el análisis de cómo evolucionan las pérdidas a lo largo del entrenamiento del modelo es necesario entender bien cómo responde la función de

pérdidas. La función de perdidas usada es *BCEWithLogitsLoss()*, una función de entropía cruzada binaria.

La función de pérdida se usa, en este caso en concreto, para medir la diferencia entre la probabilidad que el discriminador asigna a un conjunto de muestras reales y sintéticas de que sean reales. Si la contraseña era verdaderamente real se calculará la diferencia de las probabilidades entre un 0.8 y un 1 (la salida esperada). La manera en la que se pondera esa diferencia es mediante la función *BCEWithLogitsLoss()* definida por la ecuación siguiente:

$$\boldsymbol{L_n = -w_n\left[y_n \cdot log\, \sigma(x_n) + (1 - y_n) \cdot log\left(1 - \sigma(x_n)\right)\right]}$$

Donde los términos representan:

- $\boldsymbol{L_n}$ es la pérdida individual asociada a una muestra de entrenamiento en particular.
- $\boldsymbol{w_n}$ es un peso opcional aplicado a la muestra para ponderar su contribución a la pérdida total.
- $\boldsymbol{y_n}$ representa la etiqueta verdadera de la muestra, que indica a que clase pertenece (0 o 1).
- $\boldsymbol{x_n}$ es la salida del modelo antes de aplicar una función de activación de tipo sigmoide.
- $\boldsymbol{\sigma(x_n)}$ es la función sigmoide, que transforma la salida del modelo en un valor entre 0 y 1, interpretado como la probabilidad de pertenecer a la clase positiva.

Perdidas del discriminante

A continuación, se analiza la salida de la función de perdidas *BCEWithLogitsLoss()* para cuando se trata de una contraseña falsa o verdadera. En las *figuras Figura* 5 *y Figura* 6 se muestra la salida para cualquier valor de entrada entre -1 y 1, el rango posible ya que la salida del discriminante es una *tanh()*.

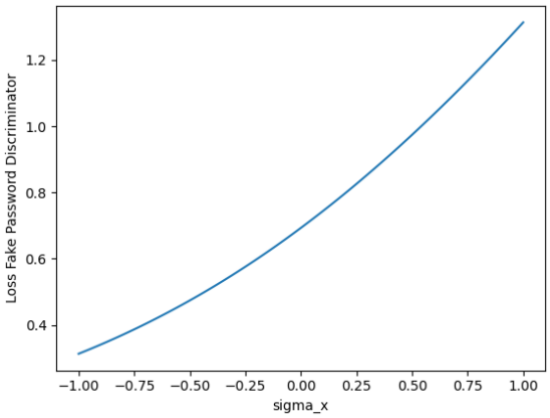

Figura 5: Salida de la función de pérdidas para una contraseña falsa

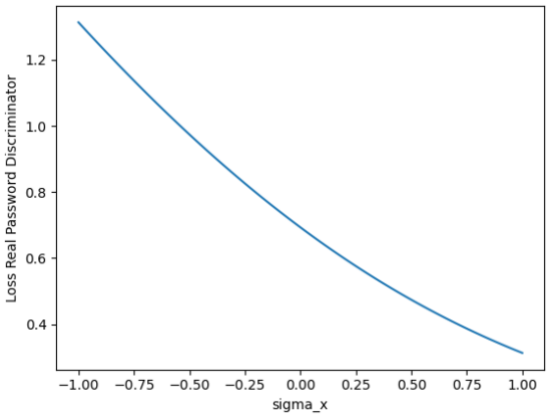

Figura 6: Salida de la función de pérdidas para una contraseña real

Cuando la contraseña es falsa la salida del discriminante (*sigma_x*) se compara contra un 0, las pérdidas del discriminante serán de 0.7 si no se equivoca nada (si su salida es un 0 aunque pueda tomar valores negativos) y de 1.3 si se equivoca por completo. Por otro lado, si la salida del discriminante tiene que ser un 1 (se introduce contraseña real), las perdidas serán de 0.3 si acierta y de 1.2 si se equivoca por completo (da como salida un -1). Finalmente, las pérdidas totales son la suma de ambas pérdidas, dicho valor se define como el coste del discriminante.

Perdidas del generador

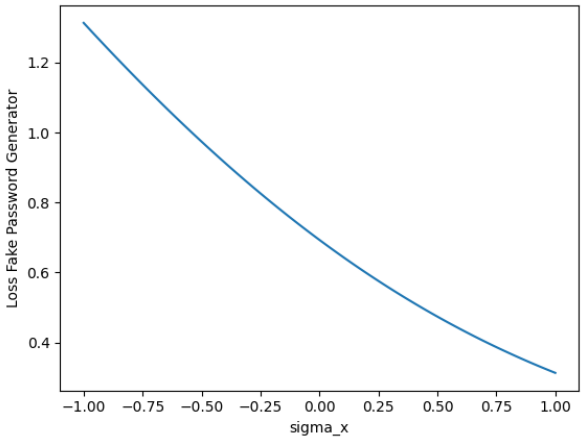

Figura 7: Salida de la función de pérdidas para una contraseña en el generador

Las pérdidas del generador son la predicción del discriminante para la contraseña falsa, pero, en este caso, con respecto a un 1 y no un 0. Esto tiene sentido ya que el objetivo del generador es engañar al discriminante y, por lo tanto, si cuando se le introduce una contraseña falsa el discriminarte predice que es verdadera (toma de salida un 1) las perdidas serán mínimas, de 0.4. Sin embargo, cuando el discriminante acierta que la contraseña es falsa (da una salida cercana al -1) las perdidas serán de 1.2 las más elevadas.

Evolución de las pérdidas del entrenamiento

A continuación, se muestran los costes tanto del discriminante como del generador:

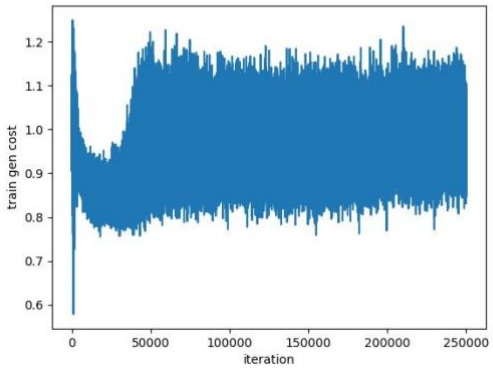

Figura 8: Evolución de las pérdidas del generador para cada iteración

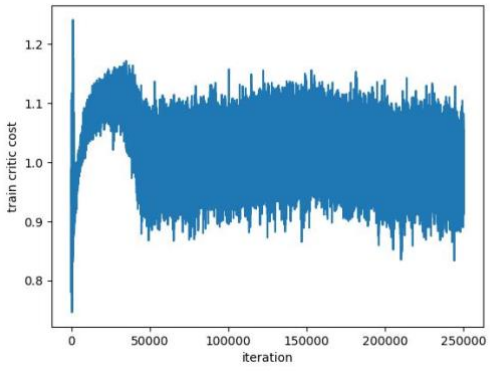

Figura 9: Evolución de las pérdidas del discriminante para cada iteración

En la figuras *Figura* 8 y *Figura* 9 se puede observar cómo en las primeras iteraciones la función de costes tanto para el discriminador como para el generador oscila entre 1.2 y 0.6. A medida que se suceden las iteraciones la función de costes del discriminante tiende a crecer y la del generador a decrecer, es decir, que el discriminador comete menos error. Llega un punto en el que esto se revierte y se estabilizan las dos entorno al 1. Para analizar en detalle lo que sucede una vez se supera la iteración 50.000 se ha representado un histograma de los costes:

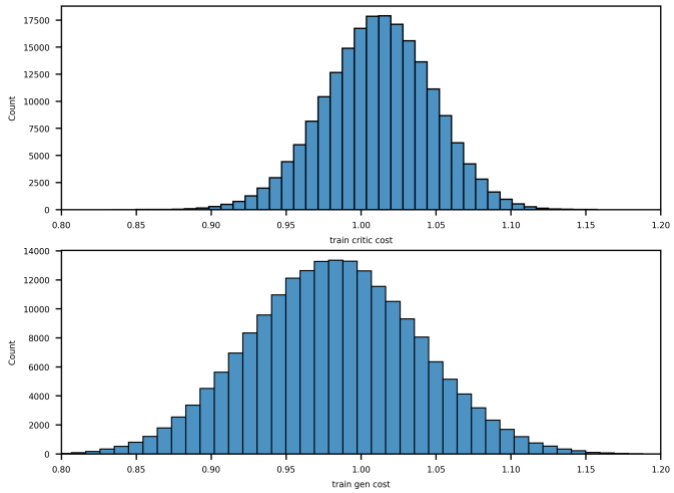

Figura 10: Histograma de las pérdidas del generador y discriminante a partir de las 50.000 iteraciones

Se observa como los costes del discriminante tienen una desviación menor, es decir, su entrenamiento es más estable. Además, se ve como la media de las pérdidas del discriminante 1.01, son mayores que las del generador de 0.98 pero se encuentran bastante igualadas.

Hay que considerar que las pérdidas del discriminante son la suma de las pérdidas para una contraseña falsa y para una verdadera, mientras que las del generador son solo de una falsa. Por lo tanto, para que compitiesen en igualdad de condiciones tendrían que dividirse las pérdidas del discriminante entre 2. En consecuencia, se puede afirmar que el discrimínate tiene aproximadamente de media la mitad de las pérdidas que el generador una vez se estabilizan el entrenamiento.

4.3.2. DISTRIBUCIÓN DE N-GRAMS

A parte de la evaluación de las pérdidas del discriminante y el generador se va a utilizar una segunda métrica para evaluar el aprendizaje de la GAN. La lógica dice que si el generador de la GAN está aprendiendo correctamente este deberá crear contraseñas que tengan la misma distribución de caracteres, sílabas, etc. que las contraseñas reales. En

este caso se ha evaluado con el cálculo de la divergencia de Jensen-Shannon para distintas ventanas o bloques de caracteres conocidos como *n-grams*.

La divergencia de Jensen-Shannon (JSD) es una medida de similitud entre dos distribuciones de probabilidad. La JSD se basa en la entropía de la distribución, cuantizando la incertidumbre del conjunto. Se puede interpretar como la cantidad de información dividida entre 2 necesaria para transformar cada una de las dos distribuciones a la media aritmética de la otra. La ecuación que la define es la siguiente:

La divergencia de Jensen-Shannon (JSD) se define como:

$$JSD(P \parallel Q) = \frac{1}{2}D(P \parallel M) + \frac{1}{2}D(Q \parallel M)$$

donde $M = \frac{1}{2}(P + Q)$ es la media aritmética de las distribuciones P y Q, y D(x) es la divergencia de Kullback-Leibler. JSD siempre va a ser mayor o igual a cero, siendo 0 si las dos distribuciones son idénticas. Por el contrario, si el valor es 1 es que la divergencia es máxima y que las dos distribuciones son lo más distintas posibles.

A continuación, se muestra el cálculo de la JSD para ventanas de *n-gram* de entre un carácter (js1) hasta cuatro caracteres (js4).

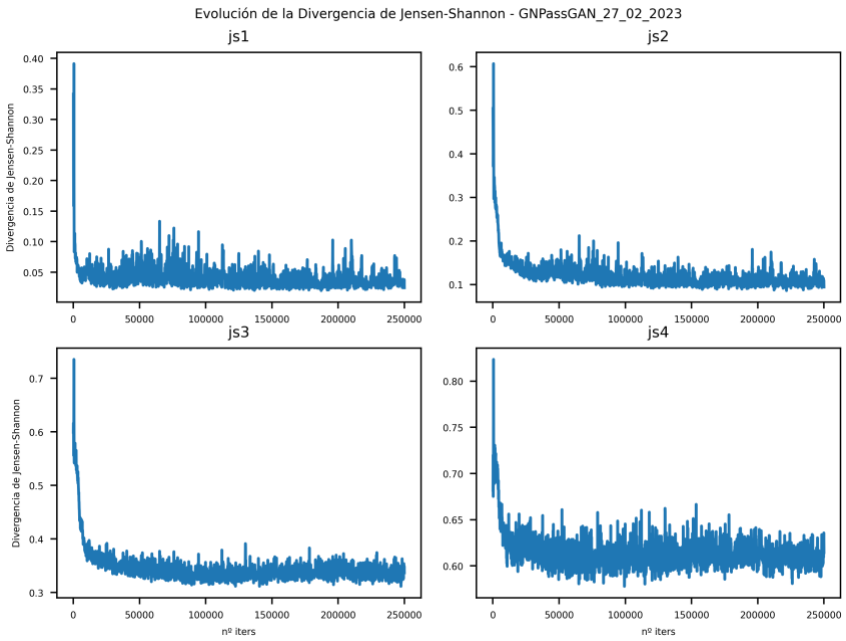

Figura 11: Evolución de la divergencia de Jensen-Shannon para n-grams de hasta 4 caracteres

Lo primero que destaca es que la divergencia decrece significativamente en las primeras iteraciones y una vez se superan las 20.000 iteraciones se estabiliza. Una vez se han estabilizado se puede comprobar que a medida que los *n-gram* abarcan más caracteres:

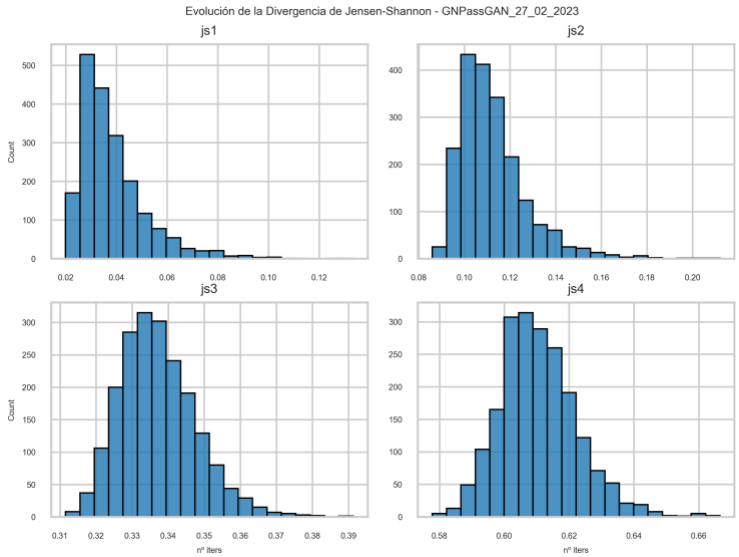

Figura 12: Histograma de la divergencia de Jensen-Shannon a partir de la iteración 50.000

La JSD para *n-grams* de un carácter tiene una cola derecha mucho más pronunciada, sin embargo, a medida que aumenta el número de caracteres de los *n-grams* esta se asemeja más a una normal. Esto tiene una explicación sencilla, para un carácter se suele equivocar o divergir muy poco de la distribución original a lo largo del entrenamiento. Cuando aumenta el número de caracteres se vuelve más complejo el acierto, por lo que la media del error aumenta y hay iteraciones que obtiene resultados muy buenos y otras muy malos.

4.4. ANÁLISIS DE LAS CONTRASEÑAS GENERADAS

En este apartado se estudian las contraseñas generadas por el modelo estudiando, además del *accuracy* en distintas bases de datos, diferentes métricas como la proporción de contraseñas únicas.

4.4.1. ACCURACY EN ROCKYOU

El *accuracy* es una métrica utilizada para evaluar el rendimiento de un modelo de generación de contraseñas. En este análisis, se ha calculado el número de contraseñas generadas que se encuentran en el conjunto de prueba, considerando el *accuracy* como el porcentaje de contraseñas generadas que coinciden con las contraseñas del conjunto de test, en relación con el número total de contraseñas en el conjunto de test.

El *accuracy* se calcula usando únicamente contraseñas que no se utilizaron para entrenar el modelo. Esta distinción es relevante porque permite medir la capacidad del modelo para generar contraseñas que sean consistentes con un conjunto de contraseñas nuevas y desconocidas. Es decir, si el modelo consigue acertar un 5% de contraseñas significa que ha sido capaz de generar contraseñas nuevas que ya uso una persona alguna vez sin que se le haya enseñado previamente esa contraseña.

Probabilidad de acierto con caracteres aleatorios

Al analizar los resultados de *accuracy* se podría argumentar que son porcentajes de acierto muy bajos en relación con el gran volumen de contraseñas usadas para calcularlo. Se podría decir que tomar contraseñas escogiendo caracteres aleatorios tendría la misma

probabilidad de acertar por pura estadística. Por ello se ha realizado el cálculo de la probabilidad de acierto si se tomasen contraseñas aleatorias.

El tamaño del archivo de palabras es de 3.586.097 palabras. Para generar las contraseñas, consideraremos un espacio de búsqueda que incluye 27 letras minúsculas, 27 letras mayúsculas y 10 dígitos, lo que da un total de 64 caracteres posibles (no se tienen en cuenta caracteres especiales). Si consideramos palabras de hasta 12 caracteres de longitud, el tamaño del espacio de búsqueda sería de 64 elevado a la potencia de 12, lo que implica tener un espacio de búsqueda extremadamente grande. Para la búsqueda se van a realizar 10^8 intentos, ya que es el tamaño de los archivos de contraseñas generados.

Para el cálculo de la probabilidad de acierto se ha tenido en cuenta:

$$Probabilidad\ coincidencia = \frac{Tamaño\ del\ archivo\ de\ contraseñas}{Tamaño\ espacio\ de\ búsqueda\ de\ contraseñas} \cdot nº\ de\ intentos$$

Basándonos en la ecuación anterior la probabilidad de que al menos una de las 10^8 palabras generadas esté contenida en el archivo de 3.586.097 palabras es extremadamente baja, aproximadamente $7{,}59 \cdot 10^{-8}$. Por lo tanto, un *accuracy* de tan solo 0,5% ya es una mejora abismal respecto al uso de caracteres aleatorios.

Accuracy para distintas iteraciones de entrenamiento

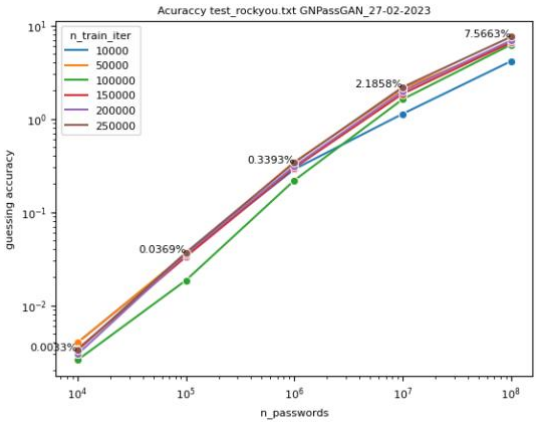

Figura 13: Accuracy en escala logarítmica del modelo entrenado en este proyecto para distinto número de iteraciones y distinto número de contraseñas generadas

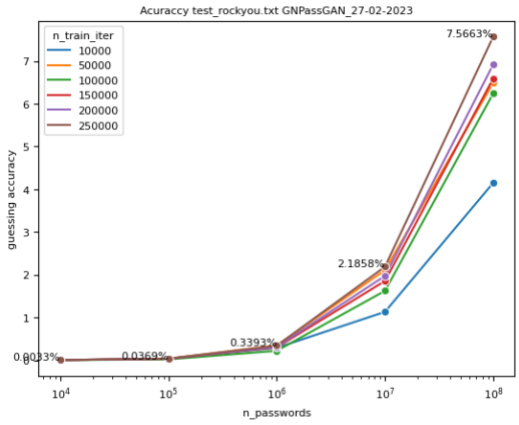

Figura 14: Accuracy del modelo entrenado en este proyecto para distinto número de iteraciones y distinto número de contraseñas generadas

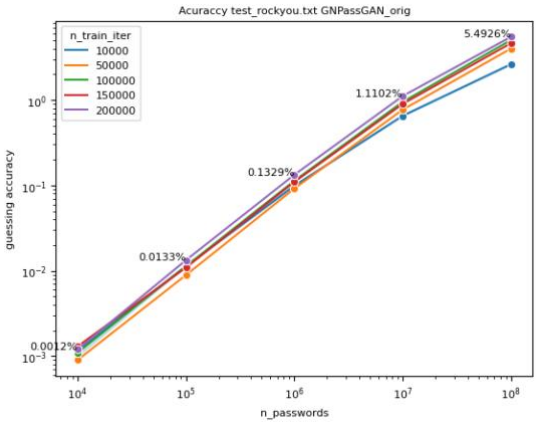

Figura 15: Accuracy en escala logarítmica del modelo del repositorio para distinto número de iteraciones y distinto número de contraseñas generadas

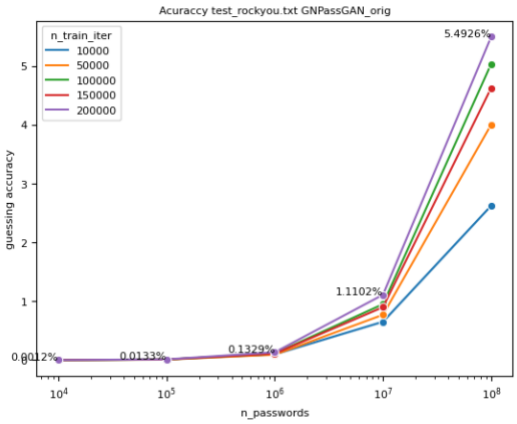

Figura 16: Accuracy del modelo del repositorio para distinto número de iteraciones y distinto número de contraseñas generadas

Se ha observado que existe una relación prácticamente lineal entre el incremento del *accuracy* y el número de contraseñas generadas. A medida que se generan más contraseñas, aumenta la probabilidad de que el modelo genere contraseñas que coincidan con las del conjunto de test. Hay un factor estadístico que explica estos resultados, el *accuracy* se calcula frente a un conjunto de test que no aumenta de tamaño. Esto significa que si tienes un *accuracy* de un 0,1% y multiplicas por 10 el número de intentos (de contraseñas generadas) si son sucesos completamente independientes el *accuracy* se multiplica por 10 también y por lo tanto sería del 1%. Sin embargo, en el caso de 10^8 frente a 10^7 no se está multiplicando por 10 el *accuracy* y esto puede ser porque el modelo está llegando a su límite de contraseñas nuevas que puede generar sin perder entropía.

En comparación con el modelo entrenado por los autores, se ha logrado una mejora significativa en el *accuracy*. Concretamente, se ha obtenido un incremento de más del 2% respecto al modelo pre-entrenado del repositorio como de los resultados reflejados en el artículo [49]. Esto indica que el modelo implementado ha logrado generar contraseñas más precisas y coincidentes con las contraseñas del conjunto de prueba en comparación con el modelo de referencia.

Además, se ha observado que el punto óptimo en términos de *accuracy* se alcanza después de 250.000 iteraciones de entrenamiento. Esto implica que a medida que el modelo se entrena durante más iteraciones, se logra una mayor precisión en la generación de contraseñas que coinciden con el conjunto de prueba. Es importante destacar que este resultado sugiere la importancia de un entrenamiento prolongado para obtener mejores resultados en términos de *accuracy*. Para ilustrar esto, consideremos el caso inicial de 10,000 iteraciones de entrenamiento y tomarlo como un nivel de *accuracy* de referencia. Al aumentar el número de iteraciones a 50.000, se observa un incremento en el *accuracy* en un 2,34%. Continuando con 100.000 iteraciones, se logra un aumento adicional de 0,74% en el *accuracy* en comparación con las 50.000 iteraciones anteriores. El progreso continúa con mejoras de 0,35% y 0,43% para 150.000 y 200.000 iteraciones, respectivamente. Finalmente, con 250.000 iteraciones se produce la mayor mejora dando un salto de 0,65%.

Por último, es crucial destacar que el modelo de generación de contraseñas supera significativamente la probabilidad de acierto que se obtendría al generar contraseñas de forma aleatoria. Esto indica que el modelo ha aprendido patrones y características específicas presentes en las contraseñas reales, lo que mejora la precisión y la utilidad del sistema de generación de contraseñas.

4.4.2. ACCURACY EN PWNED

En este análisis, se ha consultado la API de Pwned [60] para evaluar si 100.000 contraseñas generadas han sido comprometidas en brechas de seguridad. Se ha examinado la evolución del *accuracy*, que representa el porcentaje de contraseñas generadas que han sufrido brechas según la base de datos de Pwned. También se ha analizado cómo varía el *accuracy* en relación con el número de contraseñas generadas y el modelo entrenado con diferentes iteraciones.

El objetivo es evaluar la efectividad del modelo frente a distintitas base de datos de contraseñas filtradas, siendo la base de datos de Pwned la mayor de contraseñas filtradas existente. La única problemática es que esta base de datos es privada (por motivos lógicos

de seguridad) pero permiten consultar si una contraseña concreta se encuentra en su base de datos o no.

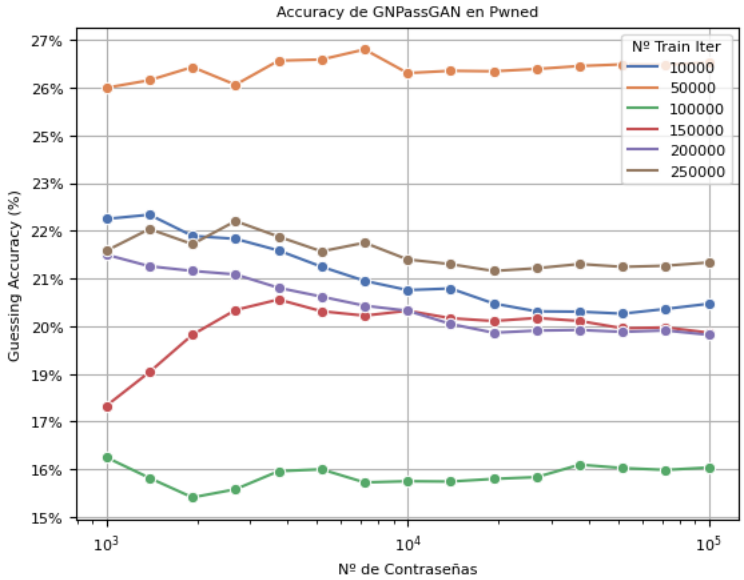

Figura 17: Accuracy del modelo entrenado en la base de datos Pwned

En general, se observa que el *accuracy* no aumenta de manera consistente al incrementar el número de contraseñas evaluadas. Algunos casos muestran un decremento seguido de estabilización, como ocurre en los modelos entrenados con 250,000, 10,000 y 200,000 iteraciones. Por otro lado, también se observa un aumento significativo en el *accuracy* según aumenta el número de contraseñas para el modelo de 150,000 iteraciones. Las variaciones en el *accuracy* para todos los rangos de tamaños analizados no superan el 3%.

En línea con lo descrito antes, se destaca que la mayor variabilidad en el *accuracy* se produce con 1.000 contraseñas evaluadas. Esto puede deberse a que la evaluación con un número tan reducido de contraseñas no proporciona una muestra significativa y, por lo tanto, el ruido en los resultados es más pronunciado. Sin embargo, a medida que se aumenta el rango de contraseñas evaluadas, entre 10.000 y 100.000, se observa una estabilización en las tendencias de variabilidad del *accuracy* lo que apoya al argumento anterior.

Por otro lado, en cuanto al análisis de los resultados para los modelos entrenados con distinto número de iteraciones revela que el mayor *accuracy* se logra con el modelo

entrenado con 50.000 iteraciones, lo que llama la atención en comparación con los resultados expuesto en el *apartado 4.4.1.* Al considerar 100.000 contraseñas (el dato de *accuracy* más significativo), se obtiene un *accuracy* para el modelo de 50.000 iteraciones que es un 5% más alto en comparación con el siguiente modelo más cercano, el de 250.000 iteraciones. En orden descendente de resultados, le siguen los modelos de 10.000, 150.000 y 200.000 iteraciones, con un rango de *accuracy* entre el 20% y el 21%. Por último, el modelo entrenado con 100.000 iteraciones muestra el peor resultado, con un *accuracy* del 16%, es decir, un 5% por debajo del resto de los modelos.

Surge la duda de si el hecho de que los modelos entrenados con un menor número de iteraciones generen contraseñas del conjunto de *train* puede ser la causa de estos resultados. Es posible que el modelo esté generando las mismas contraseñas con las que fue entrenado en las primeras etapas del entrenamiento, lo que dificulta que el discriminante distinga entre las contraseñas generadas y las del conjunto de entrenamiento. En una etapa posterior del entrenamiento puede que el modelo comience a generar contraseñas diferentes, lo que afecta negativamente el *accuracy* en la consulta a Pwned. Esta disminución en el *accuracy* se puede producir porque las contraseñas del conjunto de entrenamiento se han visto comprometidas en un 100% de los casos ya que provienen de la filtración de RockYou. Para analizar está hipótesis se ha realizado el siguiente gráfico:

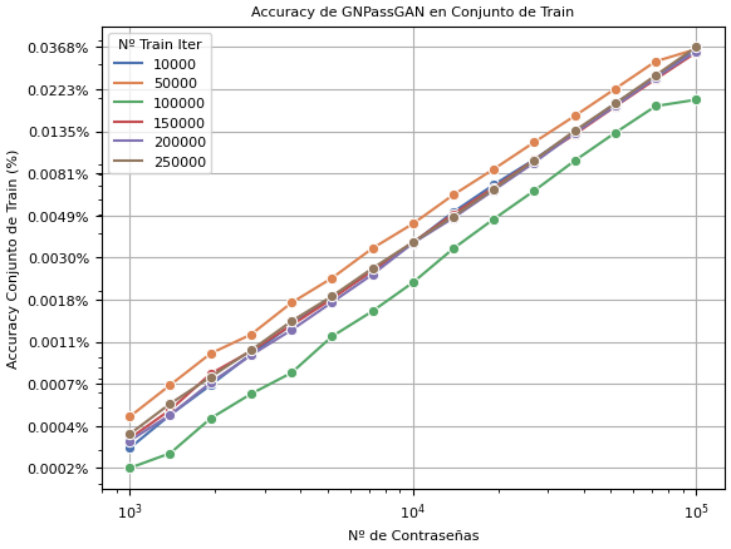

Figura 18: Porcentaje de contraseñas generadas existentes en el conjunto de entrenamiento

En esté grafico se puede ver como el porcentaje de contraseñas generadas por los modelos que se encontraban en el conjunto de *train* es muy bajo, de 0,0368% en el valor más alto con el modelo de 50.000 iteraciones y evaluado para 100.000 contraseñas. Se puede ver que la tendencia de aumento del porcentaje de contraseñas generadas coincidentes en el conjunto de entrenamiento sigue una tendencia lineal, lo cual es el resultado lógico ya que a aumentar el número de contraseñas donde se calcula el porcentaje de acierto aumenta proporcionalmente.

Para descartar el efecto de las contraseñas en el conjunto de entrenamiento se ha descartado para calcular el *accuracy* en Pwned y vuelto a representar:

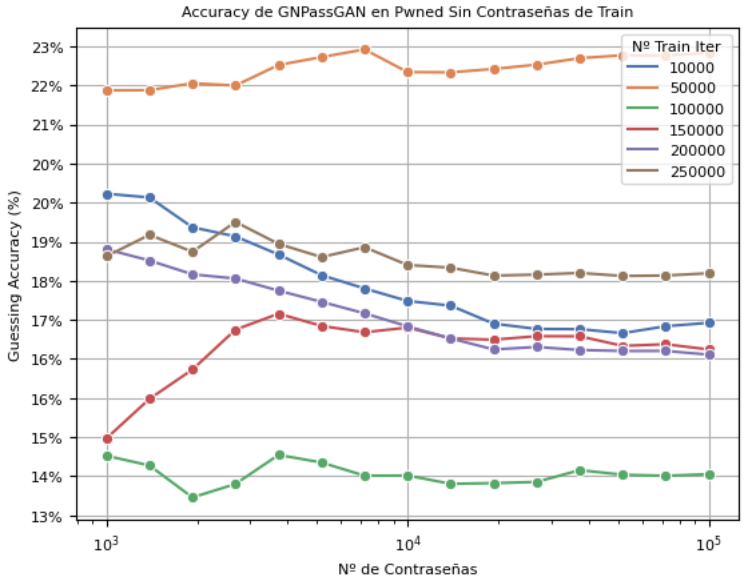

Figura 19: Accuracy del modelo entrenado en la base de datos Pwned descontando las contraseñas generadas existentes en conjunto de entrenamiento

Se ha comprobado que en términos generales el *accuracy* ha descendido un 3% al quitar las contraseñas del conjunto de entrenamiento pero, sin embargo, se han mantenido las diferencias y tendencias para los modelos entrenados con distintas iteraciones y evaluado para distinto número de contraseñas.

Por lo tanto, la hipótesis de que pudiese ser debido a que el modelo con 50.000 iteraciones esté generando mayor proporción de contraseñas de *train* que el resto y eso le beneficie es falsa. El segundo razonamiento que da explicación es que el modelo esté

sobreaprendiendo el estilo de contraseñas de *RockYou* y por eso a medida que aumenta el número de iteraciones el *accuracy* en el conjunto de validación aumente. Pero, al contrastarlo con una base de datos genérica, con miles de millones de contraseñas filtradas esta mejora no aumenta porque el modelo se ha especializado en el estilo de contraseñas de *RockYou* y no generaliza bien.

Es interesante analizar también que el modelo está siendo capaz de generar contraseñas con millones de filtraciones y que sin embargo no estaban en el conjunto de entrenamiento. A continuación, se muestran las nubes de palabras donde el tamaño de la palabra es proporcional al número de brechas identificadas en la base de datos Pwned.

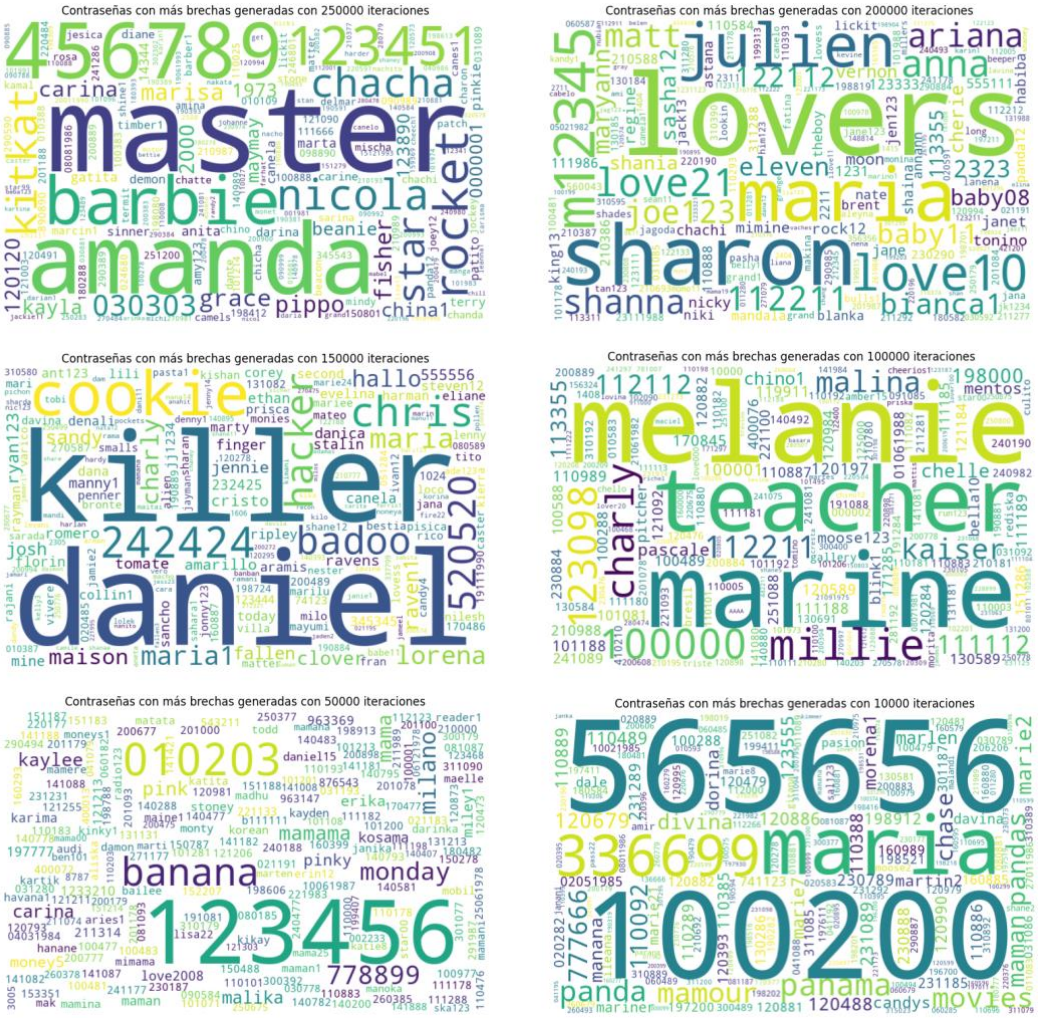

Figura 20: WordCloud para las contraseñas generadas de 6 caracteres y distinto númro de iteraciones de entrenamiento

Por ejemplo, destacan palabras como 'amanda', 'ariana', 'carina' o 'panama' que son nombres españoles comunes. No se encuentran en el conjunto de *RockYou* y, sin embargo, el modelo ha sido capaz de generarlas.

4.4.3. PROPORCIÓN DE CONTRASEÑAS ÚNICAS

El análisis de contraseñas únicas es un componente clave en la evaluación del modelo de generación de contraseñas, ya que por muchas contraseñas que sea capaz de generar si muchas de ellas se repiten pierde su utilidad. Para analizarlo se ha calculado la proporción entre el número total de contraseñas generadas y el número de contraseñas únicas. Este análisis se ha realizado para cada número de iteraciones de entrenamiento y para distintas cantidades de contraseñas generadas. El objetivo es examinar la variabilidad y la eficacia del modelo en la generación de contraseñas únicas, lo cual es fundamental para determinar en que proporción *PassGAN* puede aumentar una base de datos de contraseñas.

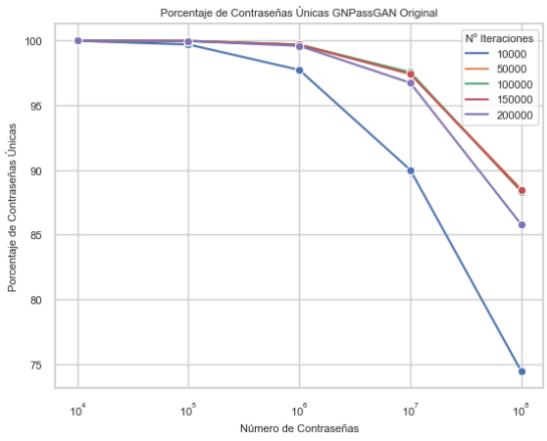

Figura 21: Porcentaje de contraseñas únicas generadas por el modelo pre-entrenado

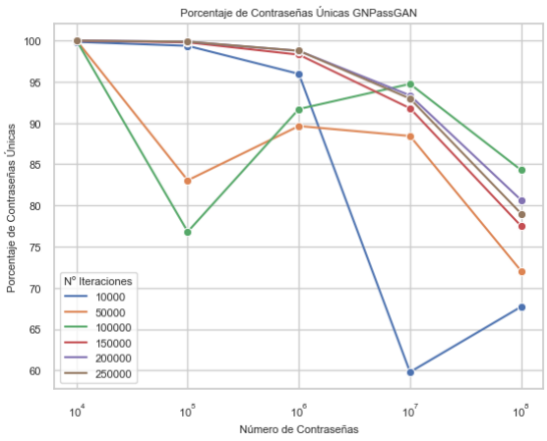

Figura 22: Porcentaje de contraseñas únicas generadas por el modelo entrenado en este proyecto

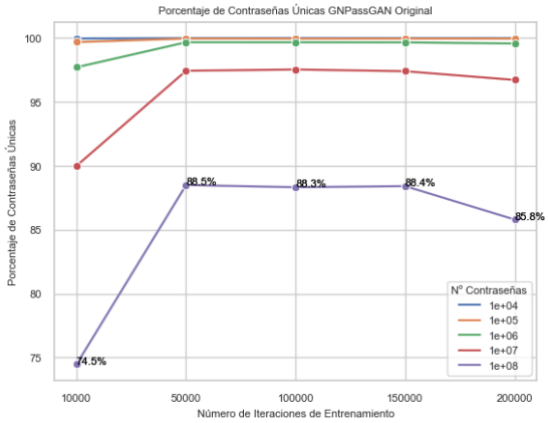

Figura 23: Porcentaje de contraseñas únicas generadas por el modelo pre-entrenado (vista 2)

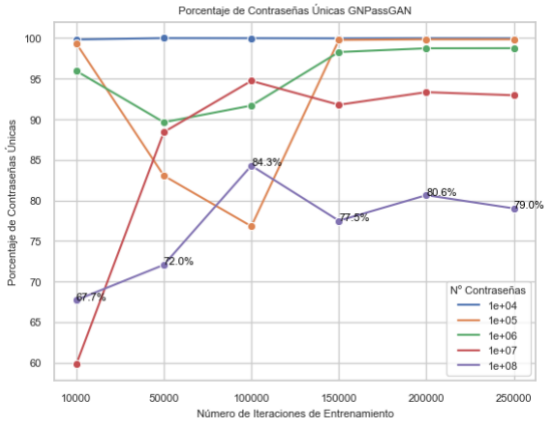

Figura 24: Porcentaje de contraseñas únicas generadas por el modelo entrenado en este proyecto (vista 2)

En los resultados obtenidos, se ha observado una tendencia en la disminución de la entropía del modelo a medida que se incrementa el número de contraseñas generadas. Esto implica que, a medida que se generan más contraseñas, se observa una proporción mayor de contraseñas repetidas en comparación con el total.

En cuanto a la evolución del número de contraseñas únicas según el número de iteraciones se han observado variaciones significativas en el modelo entrenado en este estudio. Específicamente, se ha observado un descenso considerable en el porcentaje de contraseñas únicas para los modelos entrenados con 50.000 y 100.000 iteraciones de un 20% y 25% respectivamente en comparación con el modelo pre-entrenado de los autores.

Además, se ha observado que, para un mismo número de iteraciones de entrenamiento, el modelo entrenado tiene un porcentaje ligeramente menor de contraseñas únicas en comparación con el modelo entrenado en este estudio. Por ejemplo, con 200.000 iteraciones y una muestra de 10^8 contraseñas, el porcentaje de contraseñas únicas para el modelo pre-entrenado es de 88,4%, mientras que para el modelo entrenado es de 80,6%.

Hay que tener también en cuenta que el *accuracy* en el *apartado 4.4.1* se calcula para contraseñas únicas lo que significa que una menor número de contraseñas únicas provoca que la probabilidad de acierto sea menor Sin embargo, que el porcentaje de contraseñas únicas sea menor no está perjudicando al *accuracy* como se demostró en el *apartado*

4.4.1., ya que se obtienen para el modelo entrenado en este proyecto mejores resultados de *accuracy* que con el pre-entrenado.

Estos resultados plantean la pregunta de hasta que punto se pueden generar contraseñas nuevas en función del tamaño del conjunto de datos con el que se ha entrenado. Existirá un punto en el que el aumento del número de contraseñas generadas ya no conlleva un aumento significativo en el porcentaje de contraseñas únicas. Hay que considerar que el mayor decremento, cuando se pasa de 10^7 a 10^8 es del 15% de contraseñas únicas, se podría argumentar que es un decremento significativo, pero es que has multiplicado por 10 el número de contraseñas generadas. Esto significa que un con 10^8 has generado 80 millones de contraseñas únicas, frente a los 9'5 millones con 10^7 por lo que sigue saliendo muy a cuenta continuar generando contraseñas, aunque la eficiencia disminuya.

4.4.4. DIFERENCIA ENTRE LAS CONTRASEÑAS GENERADAS PARA CADA ITERACIÓN

Este apartado se centra en responder a la pregunta de que porcentaje de contraseñas distintas genera el modelo entrenado para distintas iteraciones. Para ello, se ha calculado usando el porcentaje de contraseñas comunes calculado como la división de la intersección de los dos archivos entre la unión de los mismos. Este cálculo permite evaluar el impacto y evolución de las iteraciones de entrenamiento en el modelo y su capacidad generativa.

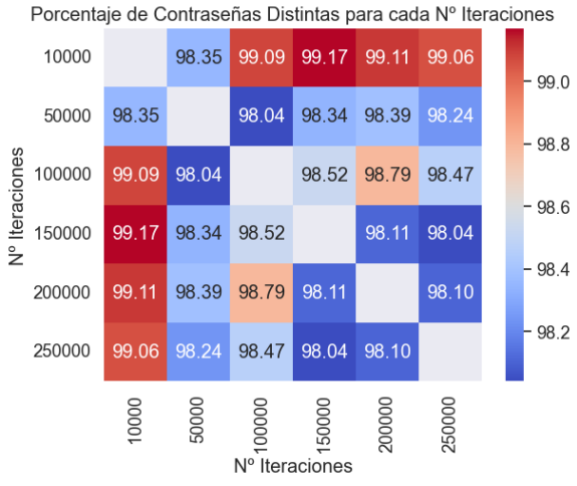

Figura 25: Porcentaje de contraseñas distintas generadas para distinto número de iteraciones

Los resultados obtenidos revelan que, en general, el porcentaje de contraseñas distintas entre archivos de cada iteración es muy alto. Puede indicar un aspecto positivo, que el modelo continúa aprendiendo y no ha alcanzado un punto de saturación en su capacidad de generación de contraseñas. En el lado opuesto puede ser un indicador negativo, el entrenamiento del modelo es muy ruidoso ya que entre iteraciones consecutivas sus contraseñas generadas son muy distintas. En este último punto hay que considerar que el criterio para decir que una contraseña es distinta o no es muy "estricto", ya que contraseñas como "love" y "lover" son muy parecidas, pero a su vez se consideran como contraseñas distintas.

También se observa una diferencia notable cuando se compara el archivo generado con 10.000 iteraciones con el resto de los archivos. En este archivo generado por el modelo menos entrenado, el porcentaje de contraseñas distintas es mayor, lo que sugiere un mayor grado de aprendizaje diferencial en las primeras etapas de entrenamiento, especialmente en el rango de 10.000 a 50.000 iteraciones.

Aunque se evidencia que el archivo generado con 10.000 iteraciones es el más distinto en comparación con los demás, la diferencia porcentual en términos de contraseñas únicas no es significativa. Por ejemplo, al comparar el archivo generado con el modelo de 150.000 iteraciones con el generado con 250.000 iteraciones, se observa que son bastante similares, con un porcentaje de contraseñas únicas de aproximadamente 98,04%. Esta pequeña diferencia porcentual indica que incluso en las iteraciones más avanzadas del entrenamiento, el modelo sigue generando un alto porcentaje de contraseñas únicas.

Figura 26: WordCloud de n-grams de 8 caracteres generados con 150.000 iteraciones de entrenamiento

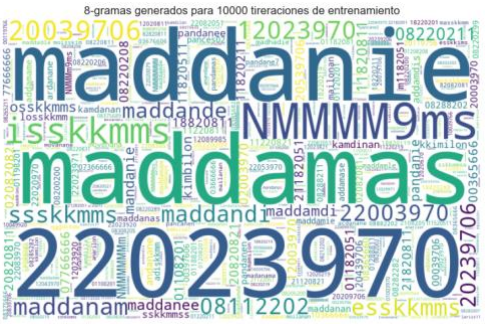

Figura 27: WordCloud de n-grams de 8 caracteres generados con 10.000 iteraciones de entrenamiento

Al analizar las nubes de palabras de los n-gramas de longitud 8 en las contraseñas generadas, se pueden observar diferencias notables entre los resultados obtenidos con 150.000 iteraciones y los obtenidos con 10.000 iteraciones.

Las contraseñas generadas con 150.000 iteraciones muestran una mayor presencia de palabras. Esto indica que el modelo ha aprendido patrones de palabras comunes durante el entrenamiento y las está utilizando en la generación de contraseñas. Estas contraseñas tienen una apariencia más "legible" y coherente en términos de palabras utilizadas.

Por otro lado, las contraseñas generadas con el modelo entrenado con 10.000 iteraciones presentan una apariencia más aleatoria y diversa. Se observa una mayor presencia de caracteres numéricos y una menor asociación con palabras comunes. Esto sugiere que el modelo en esta etapa inicial de entrenamiento aún no ha aprendido suficientes patrones lingüísticos y está generando contraseñas más aleatorias en su estructura.

4.4.5. DISTRIBUCIÓN DE LA LONGITUD DE LAS CONTRASEÑAS

En este análisis, se ha examinado la distribución de las longitudes de las contraseñas generadas por el modelo. Se considera la longitud de la contraseña teniendo en cuenta el carácter final '\n'. Por ejemplo, una contraseña como "lover" se cuenta como una contraseña de 6 caracteres porque en realidad es "lover\n".

Para visualizar esta distribución, se han generado histogramas separados para cada archivo de contraseñas generadas con diferentes números de iteraciones. Los histogramas

muestran la frecuencia de las longitudes de las contraseñas en el eje vertical, mientras que las longitudes se representan en el eje horizontal.

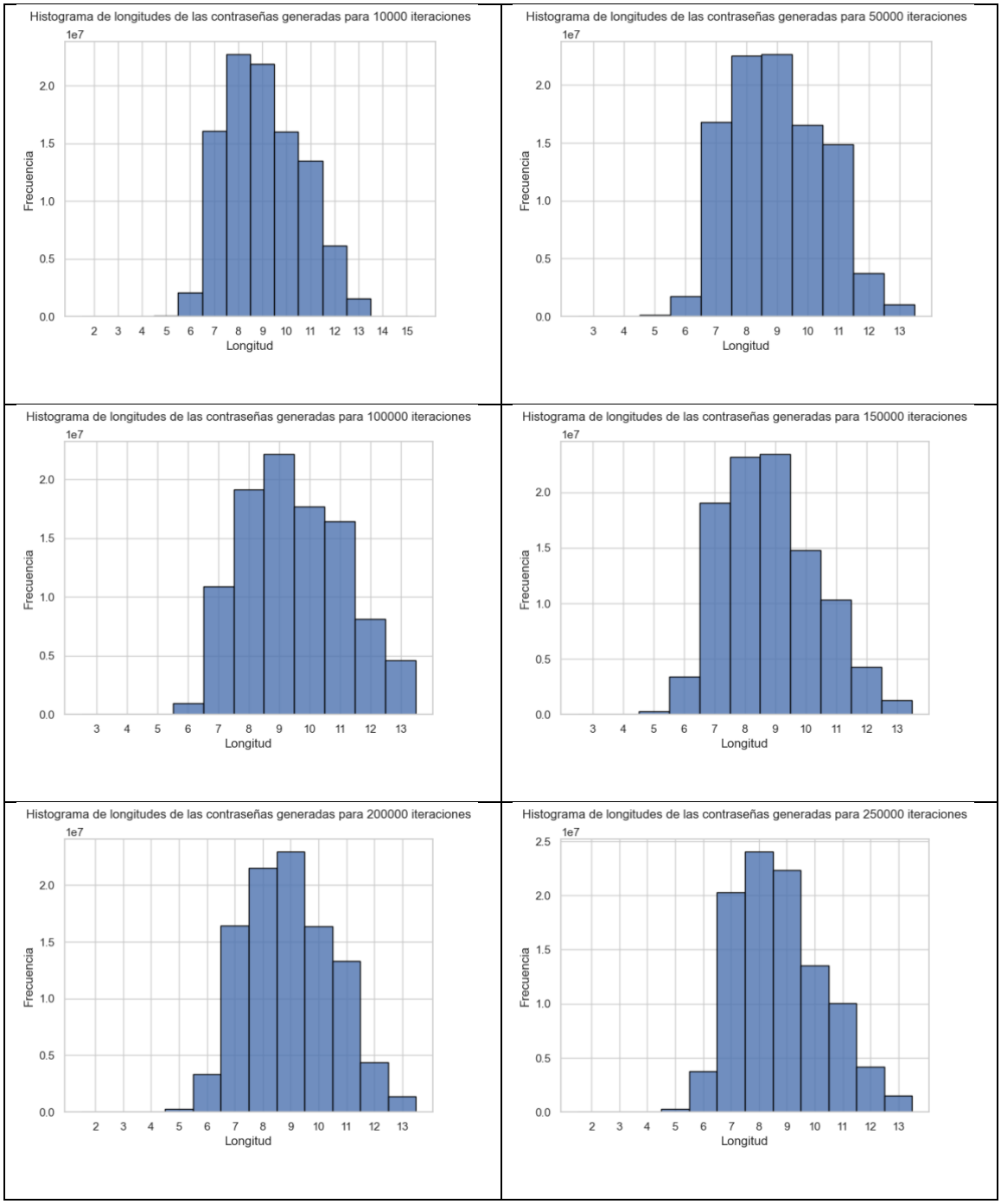

Figura 28: Histogramas de las longitudes de las contraseñas generadas para las distintas iteraciones

En general, se encuentra que las contraseñas más frecuentes tienen una longitud de 8 caracteres, seguidas de 9 y 10 caracteres. A medida que aumenta la longitud de las

contraseñas, la frecuencia disminuye gradualmente. Estos resultados se mantienen en la mayoría de los modelos, aunque existen ligeras variaciones en el orden de frecuencia entre ellos. ¿Pero esto se parece con las longitudes del conjunto de entrenamiento y de test? En las figuras Figura 29, Figura 30, Figura 31 y Figura 32 se muestran las distribuciones del conjunto de train y de test:

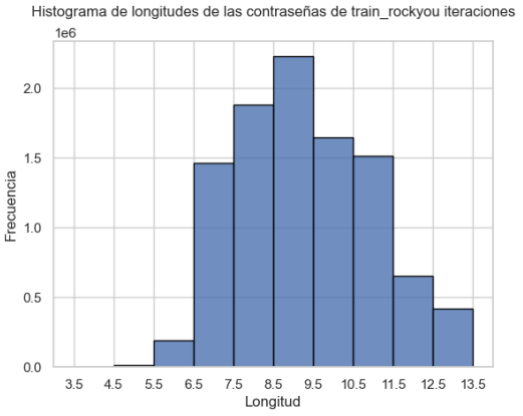

Figura 29: Histograma de la longitud de las contraseñas del conjunto de train de RockYou limitado a 13 caracteres

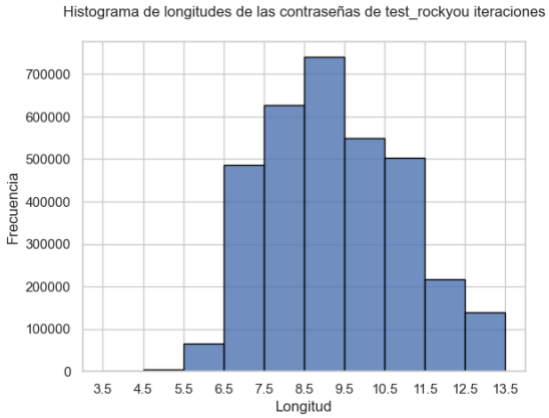

Figura 30: Histograma de la longitud de las contraseñas del conjunto de test de RockYou limitado a 13 caracteres

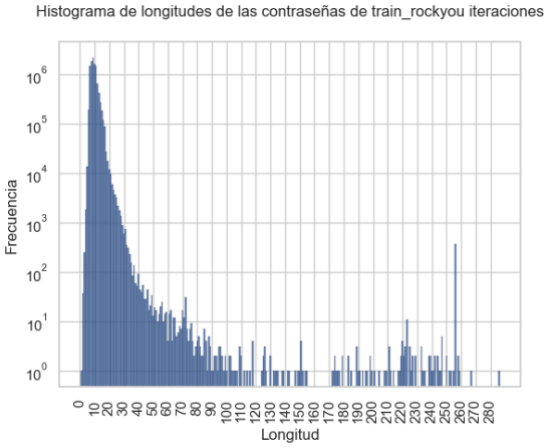

Figura 31: Histograma de la longitud de las contraseñas del conjunto de train de RockYou limitado a 13 caracteres

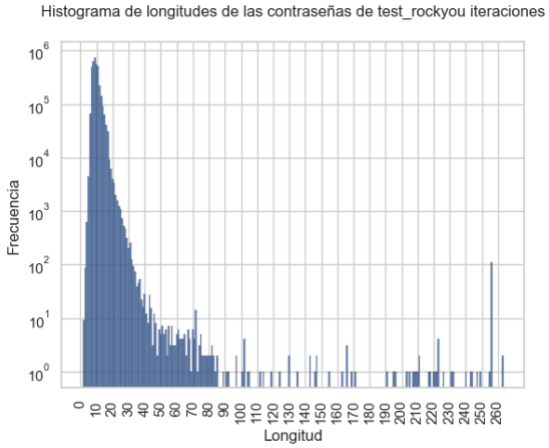

Figura 32: Histograma de la longitud de las contraseñas del conjunto de test de RockYou

Se observa una distribución de longitudes comunes en los caracteres de 8, 9, 10, 11 y 12. A medida que aumenta la longitud de las contraseñas, se aprecia una tendencia decreciente exponencial en su distribución.

Además, es interesante resaltar que se encontraron contraseñas excepcionalmente largas, llegando hasta los 280 caracteres, las cuales parecen ser fragmentos de textos o incluso fragmentos de lenguajes de programación.

También, al analizar las longitudes más largas, se observa una distribución irregular, con frecuencias muy bajas e incluso nulas en algunas longitudes específicas. Esto sugiere que las contraseñas extremadamente largas son mucho menos probables de generarse en comparación con las contraseñas de longitud más corta.

Es importante tener en cuenta que, al comparar estos resultados con datos anteriores de contraseñas generadas para diferentes iteraciones, se puede notar una variabilidad en la distribución de las longitudes. Aunque está diferencia es prácticamente inapreciable resalta la importancia del tamaño de la muestra y el enfoque utilizado por el modelo en la generación de contraseñas, ya que estos factores pueden influir en la distribución resultante.

4.4.6. PROPORCION DE NÚMEROS Y LETRAS

En el análisis de las contraseñas generadas en cada archivo, se ha examinado la proporción de caracteres numéricos y letras presentes en ellas. Este aspecto es de particular interés en el estudio realizado. El gráfico proporciona una representación visual de la proporción de caracteres numéricos, letras y caracteres especiales en los archivos de contraseñas generadas por el modelo entrenado con diferentes iteraciones. Además, se realiza una comparación con la proporción de caracteres en el conjunto de entrenamiento y prueba utilizado en el modelo.

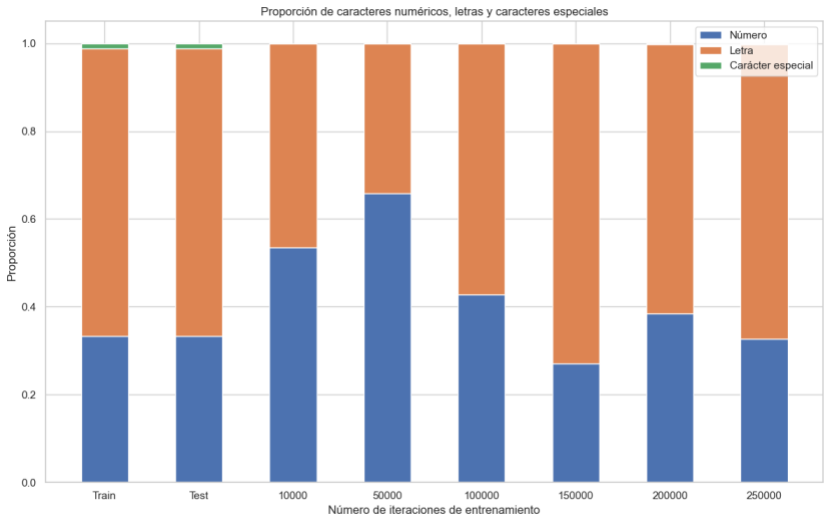

Figura 33: Proporción de caracteres numéricos, letras y caracteres especiales

En los resultados obtenidos, se puede observar una variación en la proporción de números presentes en las contraseñas generadas por los modelos entrenados con diferentes iteraciones. En particular, destaca que los modelos entrenados con menos iteraciones muestran una proporción mayor de números en las contraseñas generadas. Sin embargo, se observa un cambio significativo en esta tendencia con el modelo entrenado con 50.000 iteraciones, donde se logra alcanzar la mayor proporción de números. Esto sugiere que el discriminador del modelo puede estar distinguiendo de manera efectiva las contraseñas falsas, ya que estas tienden a tener una mayor presencia de números.

En contraste, el modelo entrenado con 150.000 iteraciones muestra una disminución significativa en la proporción de números en las contraseñas generadas. Esto puede indicar un aprendizaje más complejo y una mayor diversidad en la generación de contraseñas, con una menor dependencia de los números como componentes clave.

Es interesante destacar que, para el modelo entrenado con 250.000 iteraciones, la proporción de números en las contraseñas generadas es similar a la encontrada en el conjunto de entrenamiento y prueba utilizado en el modelo. Esto podría indicar un mayor nivel de convergencia y consistencia entre el modelo y los datos de entrenamiento.

Si entramos en detalle que caracteres son más frecuentes observamos que:

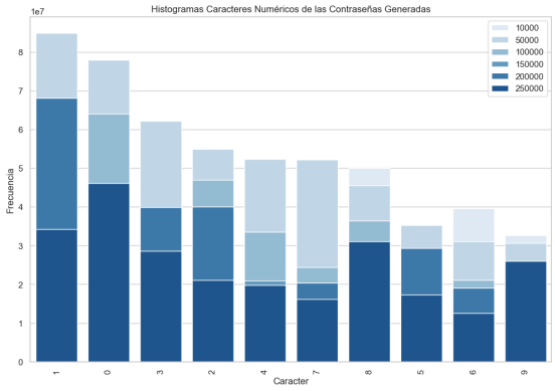

Figura 34: Histograma de los caracteres numéricos más frecuentes para cada número de iteraciones

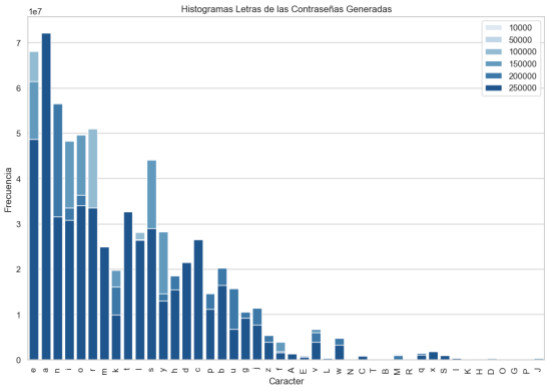

Figura 35: Histograma de los caracteres alfabéticos más frecuentes para cada número de iteraciones

En cuanto a los caracteres numéricos, se observa que los números 1, 0 y 3 son los más frecuentes, siendo el número 1 el más predominante. Además, se aprecia una diferencia significativa en la frecuencia entre los primeros modelos entrenados, donde el número más frecuente es cinco veces más común que el menos frecuente. A medida que avanza el entrenamiento, esta diferencia de frecuencia se reduce a aproximadamente la mitad. Esto puede deberse a que, a medida que se entrena con más contraseñas y el modelo se familiariza con el conjunto de contraseñas, estos sesgos iniciales se van mitigando.

En el caso de las letras, se observa una distribución no homogénea. Las vocales 'a' y 'e' son muy frecuentes, mientras que letras como 'z' y 'f' son menos comunes. Las letras en mayúscula son aún menos frecuentes que las minúsculas.

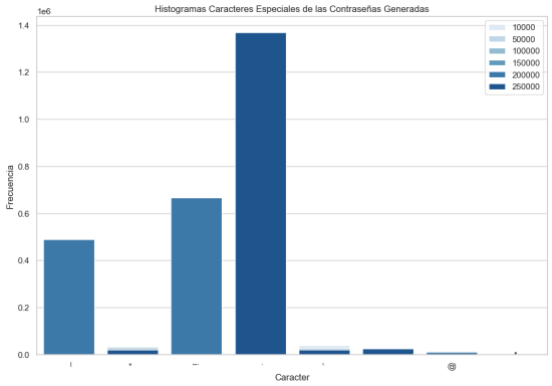

Figura 36: Histograma de los caracteres especiales más frecuentes para cada número de iteraciones

En cuanto a los caracteres especiales, destaca la presencia del punto '.' en las contraseñas generadas. Además, se observa una notable variación en las proporciones en función del número de iteraciones utilizadas para el entrenamiento. Esto puede deberse al hecho de que los caracteres especiales tienen una frecuencia mucho menor en general, lo que implica que aparecen con menos frecuencia en el conjunto de contraseñas de entrenamiento. Como resultado, adaptarse y determinar cuándo el generador debe producir estos caracteres se vuelve más desafiante para el modelo. Por lo tanto, la proporción en la que se generan los caracteres especiales está más sujeta al ruido y las características específicas de las contraseñas utilizadas para el entrenamiento en cada iteración.

Esta variación en las proporciones de caracteres especiales refleja la influencia de los datos de entrenamiento en la capacidad del modelo para generar contraseñas con estos caracteres. Al tener una frecuencia más baja y una aparición menos frecuente en el conjunto de entrenamiento, los caracteres especiales presentan un desafío adicional para el modelo en comparación con los números y las letras. Por lo tanto, es importante tener en cuenta esta variabilidad al evaluar la calidad y la diversidad de las contraseñas generadas.

4.5. GNPASSGAN CON LÉXICO REAL

En este análisis se ha entrenado GNPassGAN utilizando el lenguaje español como base. Una vez entrenado, se han generado nuevas palabras y se ha evaluado cuántas de ellas se encuentran en el conjunto de test, es decir, cuántas son palabras que existen realmente y con las que no se le ha entrenado.

El objetivo es evaluar cómo se comporta el modelo frente a palabras "reales" que son mucho menos aleatorias que las contraseñas de *RockYou*. Evaluar cómo de capaz es el modelo de aprender el lenguaje español y si a partir de esté es capaz de generar palabras de otros idiomas.

4.5.1. METODOLOGÍA

Para el entrenamiento del modelo, se utilizó un conjunto de datos que consiste en palabras en español descargadas del repositorio [63]. Se han utilizado solo las palabras de 10 caracteres siendo un total de 200.000. Para el cálculo del *accuracy* con distintos datasets se han empleado diccionarios de diferentes idiomas descargados del repositorio [64]. Estos diccionarios se han preprocesado con *Python* para prepararlos y darles el formato adecuado.

En este caso el modelo se ha entrenado para con tan solo 20.000 iteraciones, suficientes para tener una idea de su comportamiento. Hay que tener en cuenta que en este caso el conjunto de entrenamiento es mucho menor, el diccionario español consta de 200.000 palabras de 10 caracteres y en el caso de las contraseñas se estaban usando 14.344.391. En este caso solo necesita 200.000/(64 * 10) ≈ 313 iteraciones para recorrer el conjunto de entrenamiento entero, siendo 64 el *batch_size* y 10 el *critic_iters* como en el entrenamiento descrito en el apartado *4.2*. Todos los parámetros de entrenamiento aparte del número de iteraciones son los mismos que los descritos en este apartado.

4.5.2. ANÁLISIS DEL ENTRENAMIENTO

En primer lugar, se pueden analizar la evolución de la función de perdidas tanto para el discriminante (*Figura 37*) como para el generador (*Figura 38*). Dentro de las GAN, la función de pérdida se usa para medir la diferencia entre la probabilidad que el discriminador asigna a un conjunto de muestras reales y sintéticas de que sean reales (en este caso en concreto). Es decir, si la salida del discriminador es 0.8 significará que existe una probabilidad del 80% que la palabra de entrada sea real. Si la palabra era verdaderamente real se calculará la diferencia de las probabilidades entre un 0.8 y un 1 (la salida esperada).

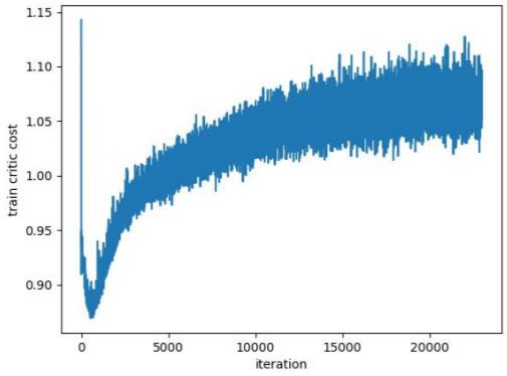

Figura 37: Evolución de las pérdidas durante el entrenamiento del discriminante con léxico real

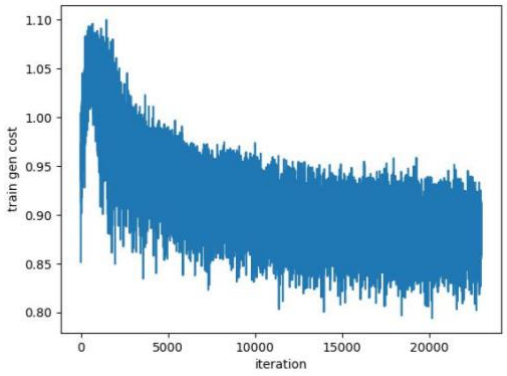

Figura 38: Evolución de las pérdidas durante el entrenamiento del generador con léxico real

En la figura se puede observar cómo en las primeras iteraciones la función de costes tanto para el discriminador como para el generador oscila entre 1.2 y 0.8. A medida que se suceden las iteraciones la función de costes del discrimínate tiende a crecer y la del generador a decrecer, es decir, que el discriminador comete menos error tendiendo a ganar.

Cálculo de la distribución de probabilidad de los N-grams:

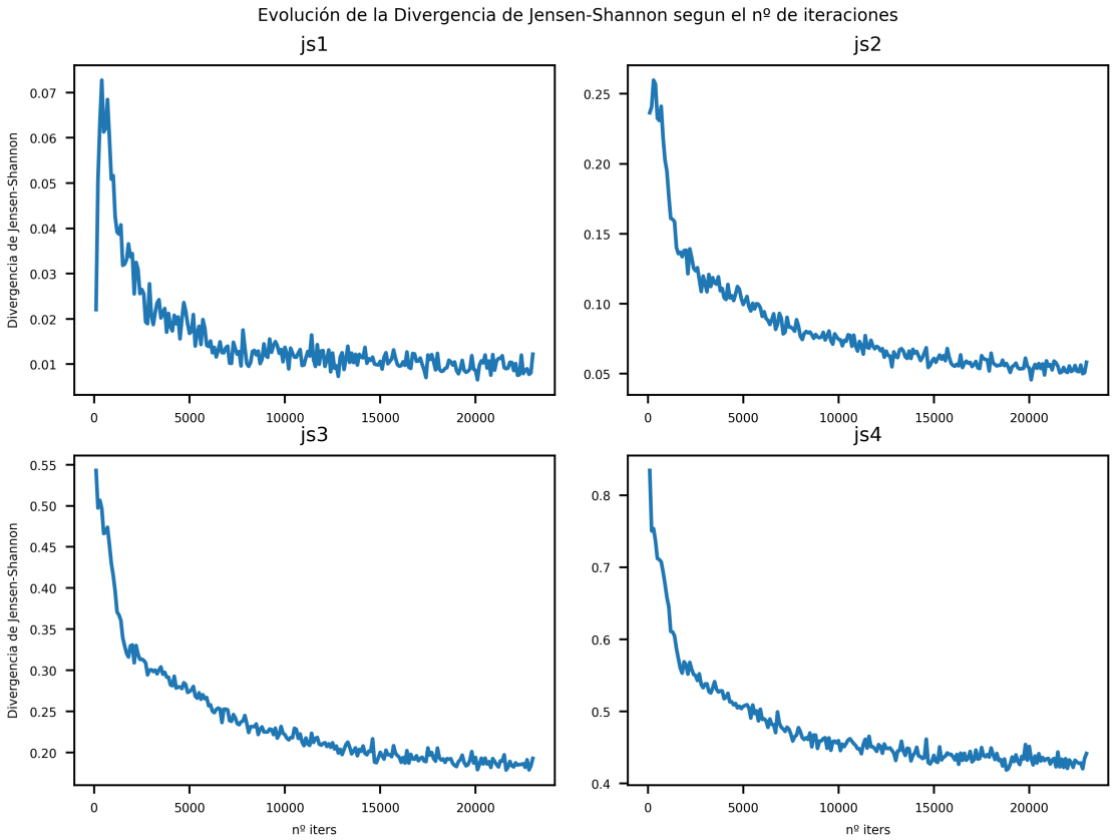

Figura 39: Evolución de la Divergencia de Jensen-Shannon para n-grams de hasta 4 caracteres entrenando con léxico general

En el caso de esté entrenamiento, al evaluar la JSD (explicada en el *apartado 4.3.2.*) se observa como la similitud es alta y que para 20.000 iteraciones ya ha dejado de decrecer por lo que podemos afirmar que basándonos en está métrica el modelo ha aprendido lo suficiente.

4.5.3. ANÁLISIS DE LAS PALABRAS GENERADAS

En esta sección, se busca obtener una medida de la capacidad de la GAN para generar palabras que se ajusten al patrón del idioma deseado, y se podrán establecer comparativas entre los diferentes idiomas evaluados. Además, realiza un análisis del volumen de palabras que puede generar la GAN, diferenciando entre el número total de palabras generadas y aquellas que son únicas.

Para evaluar la calidad de las palabras generadas, se ha realizado una evaluación del acierto en los idiomas francés, italiano, portugués y catalán. El *accuracy* se calcula como el porcentaje de contraseñas generadas que se encuentran en el conjunto de palabras de 10 caracteres del idioma correspondiente.

Es importante destacar que no se puede esperar un *accuracy* elevado en este tipo de evaluación como en un modelo clásico no generativo. Por lo tanto, es razonable obtener un porcentaje en torno al 5% ya que significa que estas acertando palabras con 10 caracteres a la perfección de un idioma con el que no ha sido entrenado. Si se escogiesen aleatoriamente la probabilidad es prácticamente nula como se analiza en el apartado *4.5.1*.

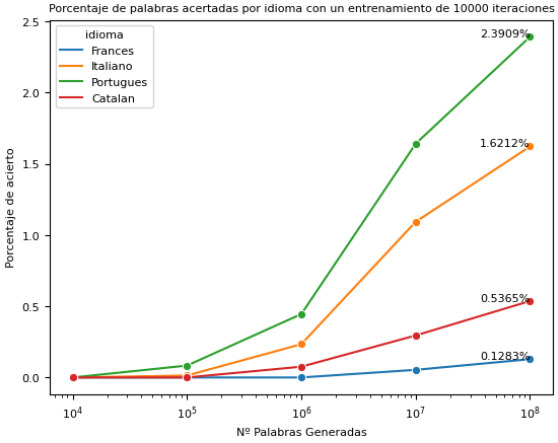

Figura 40: Porcentaje de palabras generadas de cada idioma con 10.000 iteraciones de entrenamiento

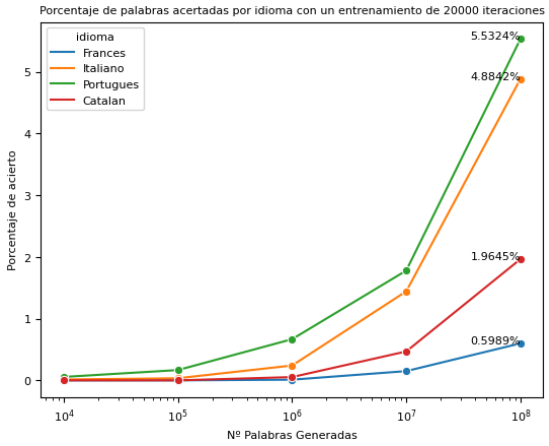

Figura 41: Porcentaje de palabras generadas de cada idioma con 20.000 iteraciones de entrenamiento

Los resultados obtenidos son bastante llamativos, ya que con tan solo 10.000 iteraciones se logran altos niveles de precisión. Por ejemplo, se obtiene un 2.4% de *accuracy* para el portugués y un 1.61% para el italiano. Al aumentar a 20.000 iteraciones, estos porcentajes se duplican en el caso del francés y se triplican en el caso del italiano. Se puede afirmar que el modelo funciona muy bien para replicar y aprender la lingüística de un idioma y que el problema al que se le está enfrentado es muy complejo.

Una de las conclusiones más relevantes que se pueden extraer de estos resultados es que el modelo es capaz de generar palabras que existen en otros idiomas. Esto es especialmente significativo, ya que son palabras reales de otros idiomas con las que el

modelo no ha sido entrenado específicamente. Además, se puede realizar una clasificación de que idiomas tienen palabras más similares al castellano. En este sentido, el portugués se posiciona como el idioma más cercano, seguido de cerca por el italiano, luego el catalán y finalmente el francés. Esta clasificación coincide con los análisis lingüísticos que indican cuáles son los idiomas más similares al castellano.

A continuación, se analiza la capacidad de generar contraseñas únicas del modelo entrenado:

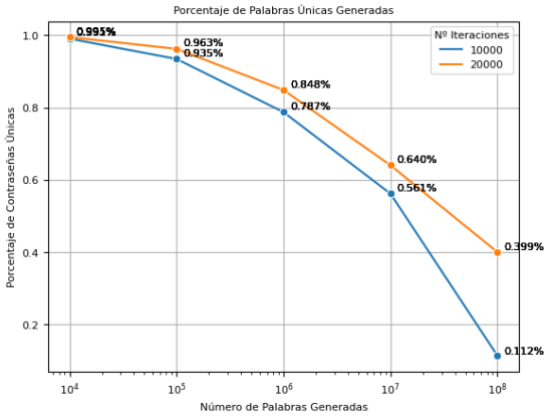

Figura 42: Porcentaje de palabras únicas generadas

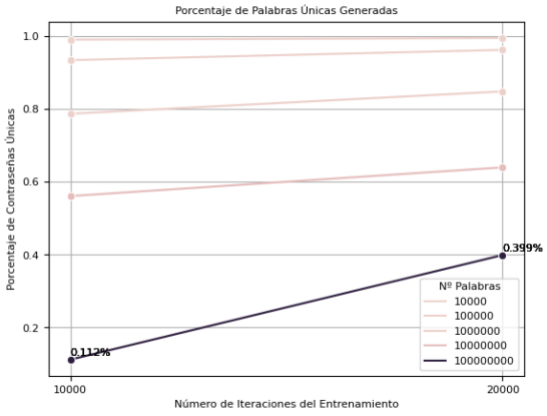

Figura 43: Porcentaje de palabras únicas generadas (vista 2)

En la gráfica se observa como a medid que se genera un volumen más grande de palabras el porcentaje de contraseñas únicas decrece hasta casi un 40% para el modelo entrenado con 20000 iteraciones. Sin embargo, aunque parezca poco que solo el 40% sean contraseñas únicas, hay que considerar que un 40% de 100M es mucho más que un 80% de 1M, por lo que sigue compensando incrementar el número de palabras generadas.

5. CONCLUSIONES Y TRABAJOS FUTUROS

Conclusiones sobre los generadores de contraseñas

Se ha podido extraer distintas conclusiones en el análisis de GNPassGAN, uno de los modelos publicados más recientes y que sus autores presentan como una mejora significativa frente a PassGAN.

La primera es lo crítico que supone ser capaz de estabilizar el entrenamiento de estos modelos. Esto no es una conclusión novedosa en el sentido de por ser una GAN siempre se presupone la complejidad de entrenarlas para que la lucha entre el generador y el discriminante esté igualada. Por ello es importante que el número de veces que gana el generador y el discriminante esté equilibrado a lo largo del entrenamiento para que vayan aprendiendo los dos y realimentarse. Sin embargo, esto provoca que la función de perdidas oscile significativamente y en consecuencia el aprendizaje del generador. Se ha concluido en las medidas que GNPassGAN consigue mejorarlo significativamente con la normalización del gradiente y el cambio de la función de perdidas, consiguiendo unas pérdidas medias de 1.01 para el discriminante y de 0.98 para el generador, muy igualadas.

La segunda conclusión, es que el generador consigue aprender con tan solo 20.000 iteraciones la distribución de los *n-grams* de las contraseñas de 1 a 4 caracteres (medido con la JSD). Pasadas las 20.000 iteraciones se podría afirmar que se estabiliza dejando de aprender, pero sin embargo se ha observado que el *accuracy* sigue incrementando incluso en la 250.000 iteraciones. Por lo tanto, aunque se tiene que considerar la JSD como una métrica para identificar que aprende no es una medida concluyente. Además, como métrica alternativa se ha identificado que la proporción de números frente a las letras y caracteres especiales no es constante, va evolucionando a lo largo del entrenamiento. En las primeras iteraciones se generan más contraseñas conformadas por más números, evoluciona hacia una menor cantidad de número y termina el entrenamiento con 250.000 iteraciones con una proporción idéntica a las del conjunto de entrenamiento y test. Esto nos refleja que, aunque el JSD permanece constante (con algo de ruido) a partir de las 20.000 iteraciones, necesita hasta 250.000 iteraciones para conseguir la misma distribución que en el conjunto de test de números y letras.

La tercera conclusión, es el ruido con el que va aprendiendo el modelo. Esto no solo se ha podido identificar con el análisis de la evolución función de pérdidas, si no que se ha estudiado cómo de distintas son las contraseñas generadas para distinto número de iteraciones. Se ha identificado que para los modelos guardados cada 50.000 iteraciones entre el 99% y 98% de las contraseñas generadas por cada modelo son distintas.

La cuarta conclusión, es que a medida que se aumenta el número de contraseñas generadas aumenta el número de contraseñas repetidas. Sin embargo, la tendencia decreciente del porcentaje de contraseñas únicas es mucho menor que la del incremento del número de contraseñas sobre las que se evalúa. Es decir, que pasando de 10.000.000 a 100.000.000 contraseñas generadas solo está decreciendo un 8% el porcentaje de contraseñas únicas, por lo que se puede seguir incrementando el número de contraseñas generadas manteniendo la eficiencia de generación.

La quinta conclusión, es que se ha conseguido entrenar el modelo alcanzando un *accuracy* de un 7,57% en el conjunto de contraseñas filtradas *RockYou*. Esté resultado es mejor que lo conseguido por el modelo pre-entrenado subido al repositorio, de 5,49%. El mayor porcentaje de *accuracy* se consigue para el mayor número de iteraciones de entrenamiento, 250.000. Por otro lado, al evaluar el modelo con la base de datos de *Pwned* [60] (la mayor de acceso público en la actualidad) se han conseguido *accuracy* de hasta el 23%. El hecho de que el modelo sea capaz de generar un 23% de contraseñas de dataste completamente distintos una de las principales conclusiones de los resultados, ya que se confirma que el modelo es capaz de generalizar y conseguir aumentar un dataste de contraseñas con una eficacia de un 23%.

La última conclusión y relaciona con la anterior, es que en *Pwned* el mayor *accuracy* se ha dado no para el modelo entrenado con 250.000, si no para 50.000 iteraciones. Tras estudiar distintas alternativas se ha concluido que el modelo no generaliza la generación de contraseñas si se excede el número de iteraciones de entrenamiento óptimas. Por lo tanto, aunque según se aumenta el número de iteraciones se consigue la mayor *accuracy* en el conjunto de validación de RockYou, no es óptimo, es preferible entrenarlo hasta 50.000 iteraciones si se quiere generalizar lo máximo posible.

Contribuciones y Futuros trabajos

La principal contribución de este trabajo es el análisis de un total de 12 repositorios de código relacionados con modelos basados en PassGAN. Esta investigación ha permitido obtener una visión más amplia y completa del estado del arte de adivinadores de contraseñas. Se ha podido

comprobar que existe una gran cantidad de artículos que proponen mejoras sobre PassGAN, sin embargo, ninguno alcanza el *accuracy* que consigue GNPassGAN.

Por otro lado, no solo se ha contribuido con el estudio de los últimos modelos publicados, si no que se ha hecho un análisis en detalle del que se ha considerado como el que presenta mejores resultados, GNPassGAN. Fruto del análisis se presentan los resultados de *accuracy* sobre la base de datos de contraseñas pública más grande. La consulta a la base de datos Pwned ha permitido obtener resultados más representativos y relevantes en cuanto a la generación y evaluación de contraseñas de los publicados en los estudios hasta la fecha.

En el afán de analizar de manera exhaustiva la capacidad de adivinación de GNPassGAN, no solo se ha orientado el proyecto al estudio de la *accuracy*, sino que se ha contribuido con el análisis de múltiples aspectos de GNPassGAN que no se contemplan en los artículos publicados: la evolución de las perdidas en el entrenamiento, la proporción de contraseñas únicas, el ruido de generación de contraseñas distintas en función del número de iteraciones de entrenamiento o la distribución de las contraseñas generadas en cuanto a caracteres y *n-grams* y longitud.

En cuanto a futuros trabajos, como consecuencia de las conclusión extraída de que GNPassGAN es el principal método actual de adivinación de contraseñas, se propone dos nuevas líneas de trabajo. La primera es el estudio de la capacidad del discrimínate para detectar contraseñas seguras generadas por el modelo. Esta alternativa permitiría garantizar que las contraseñas introducidas por los usuarios cumplan con los estándares de seguridad de tal forma que un atacante que haga uso de un modelo GNPassGAN o similar no sea capaz de atacar al sistema con contraseñas generadas falsas. De esta forma se puede evaluar cada vez que un usuario introduzca una nueva contraseña en el sistema como de probable es que un atacante usando GNPassGAN la pueda adivinar.

La segunda línea es la implementación de una sistema de *honeyword* que permita detectar si se han producido filtraciones de contraseñas en el sistema. El objetivo sería integrar GNPassGAN dentro del sistema de gestión de las contraseñas, añadiendo a la base de datos contraseñas muy similares a las existentes pero que a su vez no hayan sido creadas por ningún usuario. De esta forma se consigue detectar si han intentado con algunas de las contraseñas ficticias si se ha producido un fuga de información.

6. BIBLIOGRAFÍA

[1] B. Hitaj, P. Gasti, G. Ateniese, and F. Perez-Cruz, "PassGAN: A Deep Learning Approach for Password Guessing." arXiv, 2017. doi: 10.48550/ARXIV.1709.00440.

[2] A. Adams and M. A. Sasse, "Users Are Not the Enemy," *Commun. ACM*, vol. 42, no. 12, pp. 40–46, Dec. 1999, doi: 10.1145/322796.322806.

[3] S. Gaw and E. W. Felten, "Password Management Strategies for Online Accounts," in *Proceedings of the Second Symposium on Usable Privacy and Security*, in SOUPS '06. New York, NY, USA: Association for Computing Machinery, 2006, pp. 44–55. doi: 10.1145/1143120.1143127.

[4] B. Ur *et al.*, "How Does Your Password Measure Up? The Effect of Strength Meters on Password Creation," in *21st USENIX Security Symposium (USENIX Security 12)*, Bellevue, WA: USENIX Association, Aug. 2012, pp. 65–80. [Online]. Available: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/ur

[5] A. Das, J. Bonneau, M. C. Caesar, N. Borisov, and X. Wang, "The Tangled Web of Password Reuse," in *Network and Distributed System Security Symposium*, 2014.

[6] S. Komanduri *et al.*, "Of Passwords and People: Measuring the Effect of Password-Composition Policies," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, in CHI '11. New York, NY, USA: Association for Computing Machinery, 2011, pp. 2595–2604. doi: 10.1145/1978942.1979321.

[7] R. Shay *et al.*, "Encountering Stronger Password Requirements: User Attitudes and Behaviors," in *Proceedings of the Sixth Symposium on Usable Privacy and Security*, in SOUPS '10. New York, NY, USA: Association for Computing Machinery, 2010. doi: 10.1145/1837110.1837113.

[8] R. Wash, E. Rader, R. Berman, and Z. Wellmer, "Understanding Password Choices: How Frequently Entered Passwords Are Re-Used across Websites," in *Proceedings of the Twelfth USENIX Conference on Usable Privacy and Security*, in SOUPS '16. USA: USENIX Association, 2016, pp. 175–188.

[9] E. Stobert and R. Biddle, "The Password Life Cycle: User Behaviour in Managing Passwords," in *10th Symposium On Usable Privacy and Security (SOUPS 2014)*, Menlo Park, CA: USENIX Association, Jul. 2014, pp. 243–255. [Online]. Available: https://www.usenix.org/conference/soups2014/proceedings/presentation/stobert

[10] W. B. Polk Donna Dodson, W., "NIST SP 800-63 Electronic Authentication Guideline," *NationalInstitute of Standards and Technology*, 2004.

[11] P. G. Fenton Michael Garcia James, "NIST SP 800-63-3 Digital Identity Guidelines," *NationalInstitute of Standards and Technology*, 2017.

[12] P. G. Theofanos Elaine Newton ,. James Fenton ,. Ray Perlner ,. Andrew Regenscheid ,. William Burr, Justin Richer ,. Naomi Lefkovitz ,. Jamie Danker ,. Yee-Yin Choong ,. Kristen Greene ,. Mary, "NIST SP 800-63B Digital Identity Guidelines: Authentication and Lifecycle Management," *NationalInstitute of Standards and Technology*, 2020.

[13] F. J. C. Daley Marjorie Merwin Daggett, Robert C., "AN EXPERIMENTAL TIME-SHARING SYSTEM," *Computation Center, Massachusetts Institute of Technology Cambridge, Massachusetts*, 1962, [Online]. Available: https://web.archive.org/web/20090906104446/http://larch-www.lcs.mit.edu:8001/~corbato/sjcc62/

[14] D. W. Scherr Tom Van Vleck,Fernando Corbató,Marjorie Daggett,Robert Daley,Peter Denning,David Alan Grier,Richard Mills,Roger Roach,Allan, "Compatible Time-Sharing System (1961-1973) Fiftieth AnniversaryCommemorative Overview," *IEEE computer society*, 2001.

[15] R. Morris and K. Thompson, "Password Security: A Case History," *Commun. ACM*, vol. 22, no. 11, pp. 594–597, Nov. 1979, doi: 10.1145/359168.359172.

[16] A. Ometov, S. Bezzateev, N. Mäkitalo, S. Andreev, T. Mikkonen, and Y. Koucheryavy, "Multi-Factor Authentication: A Survey," *Cryptography*, vol. 2, no. 1, 2018, doi: 10.3390/cryptography2010001.

[17] S. Gupta, A. Singhal, and A. Kapoor, "A literature survey on social engineering attacks: Phishing attack," in *2016 International Conference on Computing, Communication and Automation (ICCCA)*, 2016, pp. 537–540. doi: 10.1109/CCAA.2016.7813778.

[18] A. L.-F. Han, D. F. Wong, and L. S. Chao, "Password cracking and countermeasures in computer security: A survey," *arXiv preprint arXiv:1411.7803*, 2014.

[19] P. Simmons, "Security through Amnesia: A Software-Based Solution to the Cold Boot Attack on Disk Encryption," in *Proceedings of the 27th Annual Computer Security Applications Conference*, in ACSAC '11. New York, NY, USA: Association for Computing Machinery, 2011, pp. 73–82. doi: 10.1145/2076732.2076743.

[20] T. Kakarla, A. Mairaj, and A. Y. Javaid, "A Real-World Password Cracking Demonstration Using Open Source Tools for Instructional Use," in *2018 IEEE International Conference on Electro/Information Technology (EIT)*, 2018, pp. 0387–0391. doi: 10.1109/EIT.2018.8500257.

[21] P. Tasevski, "Password Attacks and Generation Strategies," 2015, doi: 10.13140/RG.2.1.1247.8807.

[22] S. Nam, S. Jeon, and J. Moon, "A New Password Cracking Model with Generative Adversarial Networks," in *Information Security Applications*, I. You, Ed., Cham: Springer International Publishing, 2020, pp. 247–258.

[23] Sourceforge, "crunch - wordlist generator." Accessed: Dec. 04, 2022. [Online]. Available: https://sourceforge.net/projects/crunch-wordlist/

[24] P. Oechslin, "Making a Faster Cryptanalytic Time-Memory Trade-Off," in *Advances in Cryptology - CRYPTO 2003*, D. Boneh, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 617–630.

[25] N. Mentens, L. Batina, B. Preneel, and I. Verbauwhede, "Time-Memory Trade-Off Attack on FPGA Platforms: UNIX Password Cracking," in *Reconfigurable Computing: Architectures and Applications*, K. Bertels, J. M. P. Cardoso, and S. Vassiliadis, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 323–334.

[26] A. Narayanan and V. Shmatikov, "Fast Dictionary Attacks on Passwords Using Time-Space Tradeoff," in *Proceedings of the 12th ACM Conference on Computer and Communications Security*, in CCS '05. New York, NY, USA: Association for Computing Machinery, 2005, pp. 364–372. doi: 10.1145/1102120.1102168.

[27] C. E. Shannon, "A mathematical theory of communication," *The Bell system technical journal*, vol. 27, no. 3, pp. 379–423, 1948.

[28] M. Li, P. Vitányi, and others, *An introduction to Kolmogorov complexity and its applications*, vol. 3. Springer, 2008.

[29] M. Weir, S. Aggarwal, B. de Medeiros, and B. Glodek, "Password Cracking Using Probabilistic Context-Free Grammars," in *2009 30th IEEE Symposium on Security and Privacy*, 2009, pp. 391–405. doi: 10.1109/SP.2009.8.

[30] J. Ma, W. Yang, M. Luo, and N. Li, "A Study of Probabilistic Password Models," in *2014 IEEE Symposium on Security and Privacy*, 2014, pp. 689–704. doi: 10.1109/SP.2014.50.

[31] M. Dürmuth, F. Angelstorf, C. Castelluccia, D. Perito, and A. Chaabane, "OMEN: Faster Password Guessing Using an Ordered Markov Enumerator," in *Engineering Secure Software and Systems*, F. Piessens, J. Caballero, and N. Bielova, Eds., Cham: Springer International Publishing, 2015, pp. 119–132.

[32] A. Ciaramella, P. D'Arco, A. De Santis, C. Galdi, and R. Tagliaferri, "Neural Network Techniques for Proactive Password Checking," *IEEE Transactions on Dependable and Secure Computing*, vol. 3, no. 4, pp. 327–339, 2006, doi: 10.1109/TDSC.2006.53.

[33] W. Melicher *et al.*, "Fast, Lean, and Accurate: Modeling Password Guessability Using Neural Networks," in *25th USENIX Security Symposium (USENIX Security 16)*, Austin, TX: USENIX Association, Aug. 2016, pp. 175–191. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/melicher

[34] L. Xu *et al.*, "Password Guessing Based on LSTM Recurrent Neural Networks," in *2017 IEEE International Conference on Computational Science and Engineering (CSE) and IEEE International Conference on Embedded and Ubiquitous Computing (EUC)*, 2017, pp. 785–788. doi: 10.1109/CSE-EUC.2017.155.

[35] Y. Liu *et al.*, "GENPass: A General Deep Learning Model for Password Guessing with PCFG Rules and Adversarial Generation," in *2018 IEEE International Conference on Communications (ICC)*, 2018, pp. 1–6. doi: 10.1109/ICC.2018.8422243.

[36] "sectools.org." Accessed: Dec. 04, 2022. [Online]. Available: https://sectools.org/tag/pass-audit/

[37] "CUPP – Common User Passwords Profiler." Accessed: Dec. 04, 2022. [Online]. Available: https://www.geeksforgeeks.org/cupp-common-user-passwords-profiler/

[38] "HashCat.net." Accessed: Dec. 04, 2022. [Online]. Available: https://hashcat.net/hashcat/

[39] "John the Ripper password cracker." Accessed: Dec. 04, 2022. [Online]. Available: https://www.openwall.com/john/

[40] "Reglas HashCat." Accessed: Dec. 04, 2022. [Online]. Available: https://github.com/hashcat/hashcat/tree/master/rules

[41] "Reglas John the Ripper." Accessed: Dec. 04, 2022. [Online]. Available: http://contest-2010.korelogic.com/rules.html

[42] I. Goodfellow *et al.*, "Generative Adversarial Nets," in *Advances in Neural Information Processing Systems*, Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Q. Weinberger, Eds., Curran Associates, Inc., 2014. [Online]. Available: https://proceedings.neurips.cc/paper/2014/file/5ca3e9b122f61f8f06494c97b1afccf3-Paper.pdf

[43] I. Gulrajani, F. Ahmed, M. Arjovsky, V. Dumoulin, and A. Courville, "Improved Training of Wasserstein GANs." arXiv, 2017. doi: 10.48550/ARXIV.1704.00028.

[44] V. Garg and L. Ahuja, "Password Guessing Using Deep Learning," in *2019 2nd International Conference on Power Energy, Environment and Intelligent Control (PEEIC)*, 2019, pp. 38–40. doi: 10.1109/PEEIC47157.2019.8976614.

[45] A. Chen, "Presenting New Dangers: A Deep Learning Approach to Password Cracking".

[46] S. Nepal, I. Kontomah, I. Oguntola, and D. Wang, "Adversarial Password Cracking," 2019.

[47] D. L. Wheeler, "zxcvbn: Low-Budget Password Strength Estimation," in *25th USENIX Security Symposium (USENIX Security 16)*, Austin, TX: USENIX Association, Aug. 2016, pp. 157–173. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/wheeler

[48] B. N. Vi, N. Ngoc Tran, and T. G. Vu The, "A GAN-based approach for password guessing," in *2021 RIVF International Conference on Computing and Communication Technologies (RIVF)*, 2021, pp. 1–5. doi: 10.1109/RIVF51545.2021.9642098.

[49] F. Yu and M. V. Martin, "GNPassGAN: Improved Generative Adversarial Networks For Trawling Offline Password Guessing," in *2022 IEEE European Symposium on Security and Privacy Workshops (EuroS&amp$\mathsemicolon$PW)*, IEEE, Jun. 2022. doi: 10.1109/eurospw55150.2022.00009.

[50] D. Pasquini, A. Gangwal, G. Ateniese, M. Bernaschi, and M. Conti, "Improving Password Guessing via Representation Learning." arXiv, 2019. doi: 10.48550/ARXIV.1910.04232.

[51] I. Tolstikhin, O. Bousquet, S. Gelly, and B. Schoelkopf, "Wasserstein Auto-Encoders." arXiv, 2017. doi: 10.48550/ARXIV.1711.01558.

[52] K. Yang, X. Hu, Q. Zhang, J. Wei, and W. Liu, "VAEPass: A lightweight passwords guessing model based on variational auto-encoder," *Computers & Security*, vol. 114, p. 102587, 2022, doi: https://doi.org/10.1016/j.cose.2021.102587.

[53] M. A. Fauzi, B. Yang, and E. Martiri, "PassGAN Based Honeywords System for Machine-Generated Passwords Database," in *2020 IEEE 6th Intl Conference on Big Data Security on Cloud (BigDataSecurity), IEEE Intl Conference on High Performance and Smart Computing, (HPSC) and IEEE Intl Conference on Intelligent Data and Security (IDS)*, 2020, pp. 214–220. doi: 10.1109/BigDataSecurity-HPSC-IDS49724.2020.00046.

[54] D. Wang, H. Cheng, P. Wang, J. Yan, and X. Huang, "A Security Analysis of Honeywords," in *NDSS 2018*, Oct. 2017. doi: 10.14722/ndss.2018.12345.

[55] D. Wang, H. Cheng, P. Wang, X. Huang, and G. Jian, "Zipf's Law in Passwords," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 11, pp. 2776–2791, 2017, doi: 10.1109/TIFS.2017.2721359.

[56] M. A. Fauzi, B. Yang, and E. Martiri, "Password Guessing-Based Legacy-UI Honeywords Generation Strategies for Achieving Flatness," in *2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC)*, 2020, pp. 1610–1615. doi: 10.1109/COMPSAC48688.2020.00-25.

[57] M. Arjovsky, S. Chintala, and L. Bottou, "Wasserstein GAN." arXiv, Dec. 06, 2017. Accessed: May 10, 2023. [Online]. Available: http://arxiv.org/abs/1701.07875

[58] T. Milne and A. Nachman, "Wasserstein GANs with Gradient Penalty Compute Congested Transport." arXiv, Jun. 30, 2022. Accessed: Jul. 01, 2023. [Online]. Available: http://arxiv.org/abs/2109.00528

[59] Y.-L. Wu, H.-H. Shuai, Z.-R. Tam, and H.-Y. Chiu, "Gradient Normalization for Generative Adversarial Networks." arXiv, Oct. 10, 2021. Accessed: May 30, 2023. [Online]. Available: http://arxiv.org/abs/2109.02235

[60] "have i been pwned?" [Online]. Available: https://haveibeenpwned.com/

[61] "TensorFlow." [Online]. Available: https://www.tensorflow.org/

[62] "PyTorch." [Online]. Available: https://pytorch.org/

[63] J. Duenas Lerin, "diccionario-espanol-txt." GitHub. Accessed: May 30, 2023. [Online]. Available: https://github.com/JorgeDuenasLerin/diccionario-espanol-txt.

[64] titoBouzout, "Dictionaries for Sublime Text." GitHub. Accessed: May 30, 2023. [Online]. Available: https://github.com/titoBouzout/Dictionaries.

ANEXO I: ENTORNO DE EJECUCIÓN

En este anexo se presenta el entorno de ejecución utilizado para la configuración de TensorFlow en Apple Silicon a través de Miniforge.

A1.1. TENSORFLOW EN APPLE SILICON EN MINIFORGE

Para configurar el entorno de ejecución de TensorFlow en Apple Silicon utilizando Miniforge, se deben seguir los siguientes pasos:

1. Configurar los comandos xcode si no se han instalado previamente. Esto se logra ejecutando el siguiente comando en la terminal:

   ```
   xcode-select –install
   ```

   Los comandos xcode son necesarios para compilar ciertas dependencias y bibliotecas.

2. Descargar y ejecutar el instalador de Miniforge3 desde el enlace de la web. Luego, se debe otorgar permisos de ejecución al archivo descargado utilizando el siguiente comando:

   ```
   chmod +x ~/Downloads/Miniforge3-MacOSX-arm64.sh
   ```

   ```
   sh ~/Downloads/Miniforge3-MacOSX-arm64.sh
   ```

   Este comando iniciará el proceso de instalación de Miniforge en su sistema. Es importante utilizar Miniforge en lugar de Miniconda, ya que Miniconda aún no es compatible con una ejecución nativa en ARM.

3. Crear un entorno conda para TensorFlow y activarlo. Esto se logra ejecutando los siguientes comandos:

   ```
   conda create –name tfm_tensorflow python=3.8
   ```

```
conda activate tfm_tensorflow
```

El primer comando crea un nuevo entorno conda llamado "tfm_tensorflow" utilizando Python 3.8 como intérprete. El segundo comando activa el entorno recién creado.

4. Instalar TensorFlow y sus dependencias utilizando los siguientes comandos:

```
python3 -m pip install tensorflow-macos
```

```
python3 -m pip install tensorflow-metal
```

Estos comandos instalan TensorFlow y las dependencias necesarias para su funcionamiento en Apple Silicon. La primera línea instala TensorFlow para macOS, mientras que la segunda línea instala el soporte de aceleración de hardware utilizando Metal.

Cabe destacar que no es compatible con versiones antiguas de TensorFlow. Aunque se puede traducir el código para que sea compatible, aún pueden existir incompatibilidades con otras bibliotecas, como `tensorflow_hub`.

Si da problemas la instalación se puede realizar con el siguiente comando:

```
`SYSTEM_VERSION_COMPAT=0 pip install tensorflow-macos tensorflow-metal`
```

La variable `SYSTEM_VERSION_COMPAT` establecida en 0 se utiliza para desactivar la verificación de compatibilidad del sistema operativo. Dado que TensorFlow para macOS en Apple Silicon aún está en desarrollo activo, esta configuración permite omitir la verificación y permitir la instalación del paquete.

Instalación mediante YAML

Otra forma de configurar el entorno conda es mediante un archivo YAML. Con esta forma es con la que se han tenido menos problemas para conseguir un entrono funcional.

Para ello, se deben seguir los siguientes pasos en sustitución de los pasos 9 y 10:

- Crear un archivo YAML con el siguiente contenido:

```
name: apple_tensorflow
channels:
  - conda-forge
  - nodefaults
dependencies:
  - absl-py
  - astunparse
  - gast
  - google-pasta
  - grpcio
  - h5py
  - ipython
  - keras-preprocessing
  - numpy
  - opt_einsum
  - pip=20.2.4
  - protobuf
  - python-flatbuffers
  - python=3.8
  - scipy
  - tensorboard
  - tensorflow-estimator
  - termcolor
  - typeguard
  - typing_extensions
  - wheel
  - wrapt
```

Este archivo YAML especifica las dependencias necesarias para TensorFlow en Apple Silicon.

- Guardar el archivo YAML con el nombre *environment.yml*.
- Crear el entorno conda utilizando el archivo YAML ejecutando el siguiente comando:

```
conda env create
```

A1.2. PYTORCH EN APPLE SILICON EN MINIFORGE

Para comenzar a utilizar PyTorch en un sistema Apple Silicon M1, es necesario realizar algunos pasos de configuración. A continuación, se detallan los comandos necesarios:

1. Creación del entorno virtual: En primer lugar, se recomienda crear un entorno virtual para aislar las dependencias del proyecto. Esto se puede lograr mediante el siguiente comando:

```
conda create -n torch_3.8 python=3.8
```

   Este comando creará un entorno virtual llamado torch_3.8 con Python 3.8 instalado.

2. Instalación de PyTorch: A continuación, se debe instalar PyTorch en el entorno virtual recién creado. Para ello, se puede utilizar el siguiente comando:

```
conda install -c conda-forge pytorch
```

   Este comando instalará PyTorch, una popular biblioteca de aprendizaje automático y procesamiento de tensores, en el entorno virtual.

3. Instalación de Jupyter Notebook: Si se desea utilizar Jupyter Notebook para desarrollar y ejecutar código de PyTorch, es recomendable instalarlo en el entorno virtual. Esto se puede hacer mediante el siguiente comando:

```
conda install -c conda-forge notebook
```

A1.3. TRABAJANDO DENTRO DE UN ENTORNO DOCKER

Se ha probado la alternativa de utilizar un entorno Docker para instalar con aplicaciones y bibliotecas específicas. A continuación, se detallan los comandos usados con el entorno Docker:

1. Construcción de una imagen Docker se ha realizado a partir de un archivo Dockerfile con un contenedor Linux con TensorFlow preinstalado. La imagen predefinida con tensorflow utilizada es `tensorflow/tensorflow`. El comando es:

   ```
   docker build -t <IMAGE_NAME> <DOCKERFILE PATH>
   ```

   Este comando construye una nueva imagen Docker con el nombre especificado y utilizando el archivo Dockerfile proporcionado.

2. Para ejecutar un contenedor Docker específico, se ha usado el comando:

   ```
   docker run --platform=linux/x86_64 -it -d –name <CONTAINER_NAME> <IMAGE_NAME>
   ```

   Este comando ejecuta un contenedor Docker con el nombre y la imagen especificados con la opción `--platform` se utiliza para especificar la plataforma del contenedor. De esta forma se virtualiza sobre la arquitectura Amd64 una arquitectura x86_64. Al virtualizar la arquitectura con el Docker no se puede usar la aceleración de la GPU nativa.

3. Para el acceso al shell del contenedor Docker en ejecución, se utiliza el comando:

   ```
   docker exec -it <CONTAINER ID> /bin/bash
   ```

La alternativa al uso de las librerías de manera nativa o con un Docker es el uso de Google Colab. Una de las principales ventajas es que ofrece recursos computacionales en la nube, lo que permite ejecutar código en máquinas virtuales de con acceso a GPUs y TPUs (unidades de procesamiento de tensores). Sin embargo, no permite instalar versiones muy antiguas de las librerías como Tensorflow 1 o Python 2. .

ANEXO II: OBJETIVOS DE DESARROLLO SOSTENIBLE

En este trabajo fin de máster se alinea con varios de los Objetivos de Desarrollo Sostenible (ODS) de las Naciones Unidas.

En primer lugar, se aborda el ODS 9: Industria, Innovación e Infraestructura. Mediante la evaluación y análisis de técnicas de generación de contraseñas, se busca mejorar la seguridad en el ámbito de la criptografía ofensiva. Esto tiene un impacto directo en la industria y en la infraestructura digital, al incrementar la probabilidad de romper los *hash* de contraseñas capturados y así fortalecer la protección de los sistemas y datos.

Además, se contribuye al ODS 16: Paz, Justicia e Instituciones Sólidas. Al estudiar y desarrollar herramientas de generación de contraseñas de manera pública y transparente, se promueve la apertura en la implementación de métodos de autenticación. Esto es esencial para garantizar la seguridad y la confianza en las instituciones, tanto en el ámbito gubernamental como en el sector privado.

Por último, se relaciona con el ODS 4: Educación de Calidad. Este trabajo de investigación y evaluación de técnicas de generación de contraseñas contribuye al avance del conocimiento en el campo de la seguridad informática. Al compartir los resultados y conclusiones obtenidos, se fomenta la difusión de conocimientos y la formación de profesionales en el área de la criptografía, contribuyendo así a una educación de calidad en el campo de la ciberseguridad.

Concluyendo, este trabajo fin de máster se alinea con los ODS 9, 16 y 4, al promover la seguridad digital, la transparencia en la implementación de métodos de autenticación y el avance del conocimiento en el ámbito de la criptografía, impactando positivamente en la industria, la justicia y la educación.