



MÁSTER EN
Big Data e Inteligencia Artificial
ONLINE

**Implementación de una arquitectura Transformer educativa
(NLP).**

TFM elaborado por:

Diego García Muro

Tutor/a de TFM:

Daniel Rubio Yagüe

- Soria, 16 Septiembre del 2025 -

Resumen

La arquitectura Transformer es un modelo que ha revolucionado el campo del procesamiento del lenguaje natural, impulsando avances significativos en tareas como la traducción automática, la generación de texto y la comprensión del lenguaje, siendo la base de los modelos Bert y GPT. A diferencia de los modelos pre-existentes como las redes neuronales recurrentes (RNN), basadas en secuencias, los Transformers se basan en mecanismos de atención que permiten procesar las palabras en paralelo, mejorando la eficiencia y la capacidad de capturar relaciones contextuales a largo plazo.

A lo largo de esta memoria se detalla el proceso de implementación y validación de una arquitectura Transformer Decoder Only orientada a la generación de texto en dos contextos concretos que son las obras de Shakespeare y artículos de Wikipedia. No se busca competir con los modelos de última generación, sino comprender los fundamentos teóricos y prácticos de los Transformers y explorar, de manera didáctica, su funcionamiento en tareas de procesamiento de lenguaje natural.

En un primer capítulo se explica que es el Procesamiento de Lenguaje Natural (NLP), abordando arquitecturas previas como RNN y LSTM, para posteriormente introducir la arquitectura Transformer y fijar el alcance y limitaciones de este proyecto.

Posteriormente, se describen los materiales y métodos empleados, comenzando con un análisis y descripción de los componentes que forman parte de la arquitectura, para comprender su funcionamiento a bajo nivel. Aquí se detalla como el texto es procesado, desde la tokenización y generación de embeddings, hasta los distintos bloques que componen el Transformer, como los mecanismos de atención, capas feed-forward y normalización. Se incluyen las librerías y el lenguaje de programación utilizado, hiperparámetros y características del sistema.

A continuación, en el capítulo de Resultados se detalla el proceso completo de desarrollo, desde la obtención, análisis y preparación de los datos hasta la implementación y entrenamiento del modelo, incluyendo resultados obtenidos, así como mejoras llevadas a cabo. Se incluye la implementación de un modelo LSTM y las distintas técnicas de validación cuantitativas y cualitativas llevadas a cabo. Finalmente, se presentan las conclusiones obtenidas y posibles líneas futuras de trabajo.

Abstract

The Transformer architecture is a model that has revolutionized the field of natural language processing, driving significant advances in tasks such as machine translation, text generation, and language understanding, and serving as the foundation for models such as BERT and GPT. Unlike pre-existing sequence-based models such as recurrent neural networks (RNNs), Transformers rely on attention mechanisms that enable words to be processed in parallel, improving efficiency and the ability to capture long-range contextual relationships.

Throughout this thesis, the implementation and validation process of a Transformer Decoder Only architecture focused on text generation is described, applied to two specific contexts: the works of Shakespeare and Wikipedia articles. The aim is not to compete with state of the art models but to understand the theoretical and practical foundations of Transformers and to explore, in a didactic manner, their functioning in natural language processing tasks.

The first chapter introduces the concept of Natural Language Processing (NLP), addressing earlier architectures such as RNNs and LSTMs, before introducing the Transformer architecture and defining the scope and limitations of this project.

Subsequently, the materials and methods used are described, beginning with an analysis and explanation of the components that make up the architecture to understand its low level functioning. This includes how the text is processed, from tokenization and embedding generation to the various blocks that compose the Transformer, such as attention mechanisms, feed-forward layers, and normalization. The programming language, libraries used, hyperparameters, and system specifications are also detailed.

Next, in the Results chapter, the complete development process is presented, from data acquisition, analysis, and preparation to model implementation and training, including the results obtained and the improvements made. The implementation of an LSTM model and the various quantitative and qualitative validation techniques carried out are also included. Finally, the conclusions and possible future lines of work are presented.

Índice general

Resumen	III
Abstract	v
1. Introducción y antecedentes	1
1.1. Procesamiento del lenguaje natural	1
1.2. Modelos previos: RNN y LSTM	2
1.3. De modelos secuenciales a mecanismos de atención	2
2. Objetivos del proyecto	5
2.1. Objetivos específicos	5
3. Material y métodos	7
3.1. PyTorch y Tensorflow	7
3.2. Flask	7
3.3. Librerías y entorno utilizado	7
3.4. Tokenización Byte Pair Encoding	8
3.5. Embeddings	10
3.6. Arquitectura Transformer	11
3.6.1. Positional encoding	11
3.6.2. Masked Multi-head Attention	11
3.6.3. Add & Norm (Sharma, 2024)	13
3.6.4. Feed Forward	15
3.6.5. Linear y softmax	16
4. Resultados	17

4.1.	Adquisición y análisis de las fuentes de datos	17
4.1.1.	Tiny Shakespeare	17
4.1.2.	WikiText2	18
4.2.	Limpieza y tokenización	18
4.2.1.	Implementación del tokenizador BPE	18
4.3.	Embeddings y Positional Encodding	20
4.4.	Mecanismos de atención	21
4.4.1.	Implementación de las múltiples cabezas de atención	22
4.5.	Normalización (Add & Norm)	23
4.6.	Feed Forward	24
4.7.	Entrenamiento	24
4.7.1.	Configuración de parámetros de entrenamiento	25
4.7.2.	Elementos regulatorios	25
4.7.3.	Función de pérdida	26
4.7.4.	Optimizadores	27
4.7.5.	Resumen del modelo para Tiny Shakespeare	28
4.8.	LSTM	29
5.	Conclusiones	31
5.1.	Primera aproximación con Tiny Shakespeare	31
5.2.	Validación	33
5.2.1.	Resultados	34
A.	Anexos	41
A.1.	Preparación del entorno	41
A.2.	Moses	42
A.3.	GELU	42
A.4.	Xavier Uniform	42
A.5.	Normal uniform	42
A.6.	Entropía	42
A.7.	MAUVE	42
A.8.	Distinct-n	42

A.9. Resultados obtenidos para Tiny Shakespeare	42
---	----

Capítulo 1

Introducción y antecedentes

En los últimos años, el auge de la inteligencia artificial ha estado impulsado por los llamados Large Language Models (LLMs), que han supuesto una revolución, transformando sectores enteros. Estos modelos están presentes tanto como herramientas que poco a poco reemplazan a los buscadores tradicionales, usadas por el público en general, como herramientas open source o de pago por uso, accesibles localmente o a través de la nube, que han dado un giro al negocio en general, originando nuevas oportunidades, automatizando tareas y ofreciendo soluciones a problemas que, o no existían o no se podían abordar con tanta efectividad.

Este tipo de modelos se construyen haciendo uso de enormes recursos computacionales y pueden llegar a tener billones de parámetros, lo que dificulta enormemente su implementación. Es por ello, que en este anteproyecto se propone el desarrollo de un modelo reducido basado en la arquitectura Transformer, con un número reducido de parámetros capaz de entrenarse usando una GPU de propósito general en unas horas. El objetivo principal no es competir con modelos de última generación, sino comprender los fundamentos teóricos y prácticos de los Transformers y explorar, de manera didáctica, su funcionamiento en tareas de procesamiento de lenguaje natural.

1.1. Procesamiento del lenguaje natural

El procesamiento de lenguaje natural (NLP) es una tecnología que permite a los ordenadores interpretar, manipular y comprender el lenguaje humano. Surgió en la década de 1950, derivado de los primeros experimentos sobre traducción automática, ganando popularidad en las décadas siguientes con aplicaciones como el filtrado de correo no deseado, la clasificación de documentos y los chatbots básicos. En 2020 se dió el punto de inflexión, con el auge de los modelos de aprendizaje profundo, que utilizaban arquitecturas de redes neuronales para analizar secuencias de datos, lo que permitió analizar bloques de texto más grandes.

El NLP está presente en la mayoría de las aplicaciones que usamos a diario, desde los asistentes de voz como Siri o Alexa, hasta los sistemas de traducción automática como Google Translate. En el ámbito empresarial los casos de uso aumentan cada día, entre los más comunes se encuentran: clasificación de documentos, análisis de sentimientos, chatbots y asistentes virtuales, motores de recomendación y análisis

de tendencias en redes sociales.

1.2. Modelos previos: RNN y LSTM

Las redes neuronales recurrentes (RNN) y redes de memoria a largo plazo (LSTM) son arquitecturas de redes neuronales capaces de procesar datos secuenciales donde el orden cronológico es importante. Una neurona recurrente, en cada instante de tiempo, recibe como entrada tanto la entrada actual como la salida del instante de tiempo anterior, lo que le permite mantener una memoria, a corto plazo, de estados previos. Por otro lado, LSTM es capaz de recordar dependencias entre los términos a lo largo de tiempos más largos, gracias a su estructura de celdas de memoria y puertas que regulan el flujo de información. En sí, mantiene información en memoria que cada neurona es capaz de borrar, escribir o leer.

1.3. De modelos secuenciales a mecanismos de atención

Antes de la aparición de esta arquitectura, los modelos existentes procesaban las palabras de forma secuencial con el fin de entender el lenguaje. Esto implica una complejidad [$O(n^2)$] de forma que para procesar 3 palabras se tenían que ejecutar 6 operaciones secuenciales (para la tercera palabra necesita recordar la primera y segunda y procesar la tercera, para la segunda tendría que recordar la primera y procesar la segunda y para la primera se debía procesar esa palabra), esto, para 10 palabras aumenta a 45 operaciones, y para 100, 4950. Computacionalmente es muy ineficiente.

Gracias al paper "*Attention is All you Need*" publicado en 2017 (Vaswani et al., 2017), donde se presentan los mecanismos de atención y se logra optimizar el proceso a una complejidad [$O(n)$]. Esto es porque plantean la posibilidad de que cada palabra mire simultáneamente a las demás, es decir, ya no hay un procesamiento secuencial, sino paralelo.

Estos mecanismos constan de 8 operaciones. En primer lugar trabajan con *embeddings* (Neuraforge, 2023), esto es, vectores densos que recogen, tras el entrenamiento, información sobre cada token en relación con el resto, ya sea información semántica, relaciones sintácticas, etc. Estas representaciones, se proyectan a 3 matrices, también entrenables, que son la base de los mecanismos de atención: Q (¿Qué busco?), K (¿Qué apporto?), V (La información real aportada) (Epichka, 2023). A partir de ellas, se calcula una métrica de atención, que indicaría para cada token o palabra, como de relevante es. Esta métrica se convierte a una distribución de probabilidad a través de la función *softmax*, de forma que simula una conciencia humana, pues los seres humanos nos enfocamos en aquella información relevante mientras que el resto queda en la periferia. Por último, se integra la información, dando a la información real (V) un peso, dado por esos pesos de atención calculados previamente.

Todo este proceso no se ejecuta una única vez, sino que se apila en capas, de forma que cada capa o cabeza de atención se encarga de una tarea distinta. Así, las capas más superficiales capturan relaciones sintácticas del tipo sujeto-verbo-objeto, mientras que las capas más profundas abordan el razonamiento abstracto y el procesamiento meta-cognitivo, de hecho, se cree que en las capas 70 a 80 de los modelos GPT la representación se asemeja al comportamiento humano (Plain English AI, 2021). En este punto, un elemento clave es el *flujo residual*. Gracias a él, toda la red comparte y acumula información y cada capa

comparte sus aportaciones al resto.

Todo lo explicado hasta ahora no es más que operaciones matemáticas que permiten al modelo encontrar patrones, pero por sí solo, un transformer no es inteligente, para que exista creatividad, razonamiento y abstracción debe cumplir con las llamadas *Scaling Laws* (Wolfe, 2025). Estas sostienen que el rendimiento del modelo mejora de forma predecible al escalar estas 3 dimensiones: número de parámetros (más de 100B ya conllevan al pensamiento abstracto), datos diversos y cómputo masivo. Es por ello, que no se espera como resultado de este proyecto un modelo capaz de razonar como lo hacen los modelos SOTA del mercado.

Pese a los grandes avances y la repercusión que están teniendo los modelos LLM hoy en día, basados en este tipo de arquitectura, hay que decir que presentan varios problemas. Por un lado la atención es $[O(n^2)]$, pues para 10,000 palabras hay 10,000x10,000 relaciones posibles, por otro lado, las infraestructuras son fijas y, además, todos los parámetros se activan siempre, lo que es ineficiente. Es por ello que están surgiendo soluciones como la atención dispersa (el modelo no atiende a todas las posiciones), arquitecturas capaces de modificarse a sí mismas de forma autónoma y técnicas como *mixture of experts* que permiten usar una fracción de los parámetros en cada paso (Plain English AI, 2021).

Capítulo 2

Objetivos del proyecto

El proyecto tiene como objetivo implementar y analizar una arquitectura Transformer funcional que no busque competir con los modelo SOTA, sino con el propósito de comprender en profundidad su funcionamiento a bajo nivel y evaluar su desempeño en un corpus de texto reducido, siendo capaz de generar texto con cierta coherencia en contextos limitados, como las obras de Shakespeare o fragmentos de artículos de Wikipedia.

2.1. Objetivos específicos

- Analizar los principios teóricos que sustentan la arquitectura *Transformer* y sus diferencias con modelos secuenciales como RNN y LSTM.
- Implementar desde cero una versión reducida de un *Transformer Decoder Only* utilizando librerías de Python.
- Diseñar y aplicar un proceso de tokenización y generación de *embeddings* adecuado al corpus utilizado.
- Entrenar y evaluar el modelo con un conjunto de datos reducido, analizando su capacidad para generar texto coherente con técnicas cualitativas y cuantitativas.
- Comparación de los resultados obtenidos frente a los producidos por un modelo LSTM, también implementado en este proyecto.
- Realizar un análisis exploratorio de los resultados obtenidos y proponer posibles mejoras o líneas futuras de trabajo.

Capítulo 3

Material y métodos

3.1. PyTorch y Tensorflow

Ambos son frameworks de aprendizaje profundo y de código abierto para gestionar e implementar los procesos de aprendizaje automático. Sin embargo, TensorFlow se basa en un gráfico computacional estático, esto es, que antes de la ejecución se definen todas las operaciones y la estructura del modelo, mientras que PyTorch utiliza un gráfico computacional dinámico, lo que permite modificar el modelo sobre la marcha durante la ejecución. En este proyecto se emplea mayormente PyTorch para implementar la arquitectura Transformer, quedando Tensorflow reservado para el desarrollo del modelo LSTM utilizado para comparar los resultados obtenidos.

3.2. Flask

Flask es un framework web escrito en Python que facilita la creación y despliegue de aplicaciones web. En este proyecto se utiliza para desplegar una API capaz de recibir un texto inicial junto con una serie de hiperparámetros que invoca al método de generación de texto en base al modelo entrenado.

3.3. Librerías y entorno utilizado

El proyecto se implementa íntegramente en Python 3.12, montado en un entorno virtual donde se instalarán las librerías necesarias a través de pip3. El ordenador utilizado cuenta con una GPU NVIDIA GeForce RTX 4090 con 15943 MB de memoria total. La versión de CUDA es la 12.9 y las librerías utilizadas se recogen en la siguiente tabla:

En el anexo [A.1](#) se incluyen los pasos de preparación del entorno.

Tabla 3.1: Librerías principales utilizadas en el proyecto

Librería	Versión
Flask	3.1.2
Torch (PyTorch)	2.9.0
Datasets (HuggingFace)	4.1.0
TensorFlow	2.20.0
NumPy	2.3.3
Pandas	2.3.2
Matplotlib	3.10.7
Scikit-learn	1.5.2
NLTK	3.9.1
SentencePiece	0.2.1

3.4. Tokenización Byte Pair Encoding

Para que el modelo sea capaz de identificar patrones y llegar a comprender un texto, es necesario su conversión a valores numéricos. Para ello, el texto, ya limpio, se descompone en unidades mínimas denominadas tokens. Un token se puede entender como un fragmento de un texto, por ejemplo, un carácter, una palabra o una subpalabra dentro de un vocabulario. Estos tokens se transforman a un valor numérico, que corresponde con el índice de ese fragmento en el vocabulario obtenido. Sin embargo, esto no es suficiente para que el modelo comprenda el texto, para ello cada token se representa como un vector que captura el contexto de las palabras y otra información importante, son los llamados *embeddings*, que se explicarán más adelante.

Existen diferentes técnicas de tokenización, pudiendo clasificarse como tokenización en caracteres, en palabras y en subpalabras. La primera tiene la ventaja de disminuir la complejidad, sin embargo, el modelo está más limitado a la hora de aprender representaciones significativas. Puede ser útil para idiomas como el chino, donde la morfología es compleja y no existen los espacios. La segunda es la tokenización por espacio y puntuación, es decir, la frase *"Mi perro come mucho."* no se descompone en los tokens: *"Mi"*, *"perro"*, *come"*, *"mucho."*, que es a lo que equivaldría hacer un split por espacios, sino que se tokenizaría como: *"Mi"*, *"perro"*, *come"*, *"mucho"*, *."*. El problema de este método es la complejidad en memoria y tiempo, pues genera un vocabulario muy grande y con ellos una matriz de embeddings también demasiado grande. A esto hay que sumar que aparece el problema de palabras no vistas o *UNK*.

La tokenización de subpalabras resuelve estos problemas. Se basa en la idea de que las palabras que se usan con frecuencia no deberían dividirse, mientras que las más raras sí, manteniendo así palabras más frecuentes. Por ejemplo, en un texto en inglés, la palabra *."noyingly"* no es una palabra común, mientras que las subpalabras *."noying"* y *"ly"* sí. Permite, con ello, ver más vocabulario, sin perder significado de las palabras menos frecuentes (LMPO, 2020)

En este proyecto, dado que son textos en inglés y las limitaciones técnicas que existen, se ha decidido utilizar un tokenizador de subpalabras. Entre estos, destacan dos: Byte Pair Encoding (BPE) y WordPiece (HuggingFace, 2021).

Ambos funcionan de forma similar, la principal diferencia radica en cómo realizan las uniones.^a partir del vocabulario base. BPE lo que hace es primeramente una pre-tokenización de los datos de entrenamiento

por espacios o basada en reglas (por ejemplo, Moses [A.2](#)). Con ello, obtiene un conjunto de palabras únicas y su frecuencia. Tras ello, crea un vocabulario base con los distintos símbolos que aparecen en las palabras únicas. La idea ahora es aprender reglas de unión para crear nuevos símbolos a partir de otros dos en base a la frecuencia. El proceso se repite hasta alcanzar el tamaño deseado, que se configura como hiperparámetro. En GPT-3 el tamaño es de 50000 tokens aproximadamente (Paul, [2025, Mar 2](#)). Para evitar un vocabulario base muy extenso si se quiere tener todos los caracteres con el fin de evitar palabras o tokens no vistos, se pueden usar bytes, así se tiene un tamaño fijo de 256, utilizando reglas para los caracteres de puntuación que no se puedan representar.

WordPiece, en lugar de escoger simplemente los pares de símbolos más frecuentes, utiliza probabilidades. Para cada par de símbolos, calcula la probabilidad combinada entre la individual de cada uno. El par con el valor de $\text{score}(A, B)$ más alto es el que se añade al vocabulario, ya que indica que ambos símbolos aparecen juntos con más probabilidad de la esperada si fueran independientes.

$$\text{puntuación}(A, B) = \frac{P(AB)}{P(A) \cdot P(B)}$$

Donde:

- $P(AB)$ es la probabilidad estimada de que aparezca el token combinado AB
- $P(A)$ y $P(B)$ son las probabilidades de cada símbolo por separado.

Dado que los *datasets* que utilizo no son muy grandes, voy a emplear BPE con SentencePiece (Google, [2018](#)). Además, este método ha sido utilizado en modelos como GPT (Reddit, [2021](#)).

SentencePiece es un framework de tokenización y detokenización desarrollado por Google que facilita la implementación de BPE. Entre sus características principales destaca el hecho de que no depende del idioma, ya que trabaja directamente con caracteres crudos, permite entrenar sin preprocesamiento previo y sin precisar librerías externas como NLTK o SpaCy, y trata los espacios en blanco como caracteres independientes, lo que resuelve problemas en lenguas sin segmentación explícita como el chino. El resultado es un modelo portable y eficiente.

Asimismo, SentencePiece incluye mecanismos avanzados que mejoran la robustez de los modelos:

- **Byte fallback**, hace que si aparece cualquier caracter desconocido no genere un UNK , sino que lo descompone en bytes.
- **BPE-dropout**, que introduce aleatoriedad en el proceso de segmentación durante el entrenamiento para mejorar la generalización, en sí lo que hace es omitir uniones aleatoriamente durante la tokenización.
- **Regularización por subpalabras**, que permite generar múltiples segmentaciones posibles de una misma secuencia, actuando como técnica de *data augmentation*.

3.5. Embeddings

La tokenización permite convertir palabras o fragmentos de texto en números enteros, pero estos números no capturan el significado o contexto de las palabras. Hay técnicas como *One-Hot Encoding* que representan cada palabra como un vector binario en el que solo una posición es 1 y el resto son ceros. Esto presenta varias desventajas: genera vectores de alta dimensionalidad, dispersos y no capturan ningún tipo de información sobre el significado, el contexto o la similitud entre palabras.

Es por ello que se utilizan ampliamente los *embeddings*. Se trata de vectores densos, de números reales, capaces de capturar el significado de las palabras, sus relaciones semánticas y su contexto dentro del lenguaje. Los embeddings se obtienen mediante entrenamiento, de manera que palabras con contextos similares tienden a tener vectores cercanos entre sí en el espacio vectorial (GeeksforGeeks, [2021](#)).

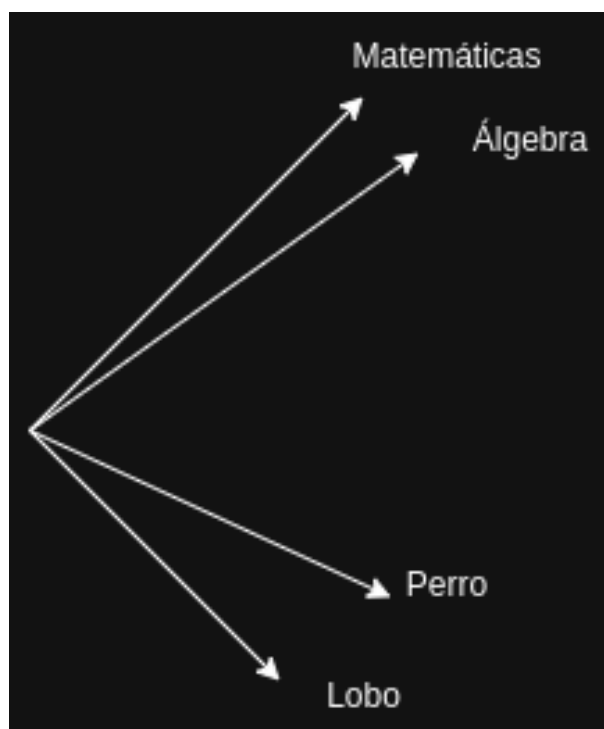


Figura 3.1: Ejemplo similitud vectorial embeddings

Internamente, un embedding se representa como una matriz de pesos $E \in \mathbb{R}^{V \times d}$, donde V es el tamaño del vocabulario y d la dimensión del embedding. Cada fila de E contiene el vector asociado a un token. Durante el entrenamiento, solo se actualizan los vectores correspondientes a los tokens presentes en cada lote, lo que permite un aprendizaje eficiente.

Otra ventaja de estos vectores multidimensionales es la posibilidad de operar con ellos, así un ejemplo clásico a la hora de hablar de embeddings, demuestra cómo capturan el contexto:

$$\text{vec}(\text{"rey"}) - \text{vec}(\text{"hombre"}) + \text{vec}(\text{"mujer"}) \approx \text{vec}(\text{"reina"})$$

En este tipo de arquitecturas, los embeddings se combinan con codificaciones posicionales para incorporar la información del orden de las palabras en la secuencia. Con ello se consigue que el modelo tenga acceso tanto al significado individual de las palabras como a su posición relativa dentro de la frase.

Así puede capturar diferencias según dónde aparezca la palabra, no es lo mismo "un banco de peces" que "dinero en el banco" (Pérez, 2020).

3.6. Arquitectura Transformer

En este proyecto, como ya se ha explicado, se va a implementar una arquitectura Transformer para la generación de texto en un contexto reducido, por lo que se ha decidido desarrollar una arquitectura del tipo Decoder Only, ya que no es necesario un proceso de codificación previo, como sería el caso de un traductor, el cual requeriría de un encoder.

A continuación, se describen los distintos bloques que componen esta arquitectura.

3.6.1. Positional encoding

Un transformer, a diferencia de otros modelos, procesa los datos en paralelo, lo que hace que desconozcan en que orden aparecen los distintos tokens. Por este motivo se realiza una codificación posicional, que asigna una posición relativa a cada token o palabra en la secuencia (Phillips, 2019).

Este aspecto es fundamental, ya que en una frase el significado de cada palabra depende del contexto y la posición en la que aparece. Por ejemplo, "Me senté en el banco.^{es} distinto a .^{El} banco de peces". En el artículo original "Attention is All You Need", se propone la utilización de funciones seno y coseno para dar a cada posición una representación única, ya que cada palabra se representa con un vector numérico. Esto permite que la salida esté normalizada, pues estos valores se encuentran en el rango $[-1, 1]$ y no requiere de entrenamiento adicional ya que los valores posicionales son determinados de forma fija y única.

$$PE_{(pos, 2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d}}}\right), \quad PE_{(pos, 2i+1)} = \cos\left(\frac{pos}{10000^{\frac{2i}{d}}}\right)$$

En este proyecto, se ha optado por utilizar una codificación posicional aprendida, donde el modelo aprende durante el entrenamiento una representación vectorial para cada posición dentro del contexto definido. Este método ofrece mayor flexibilidad y capacidad de adaptación a los datos, aunque limita la generalización a secuencias de longitud distinta a la utilizada durante el entrenamiento.

3.6.2. Masked Multi-head Attention

El componente principal de esta arquitectura son los llamados mecanismos de atención. Estos sopesan la importancia de distintos tokens en la secuencia de entrada. En sí, se puede decir que cada palabra mira al resto para ver cuáles tienen mayor importancia a la hora de entender el contexto. Por ejemplo, ".^{El} perro ladra en el prado", la palabra "ladra" tendrá un peso mayor en "perro" que "prado".

En la práctica, se emplean múltiples mecanismos de atención en paralelo, es lo que se conoce como multi-head attention. Cada una de estas cabezas aprende una proyección distinta del espacio de representaciones, lo que permite capturar diferentes tipos de dependencias (sintácticas, semánticas, etc.) entre las palabras. Finalmente, las salidas de todas las cabezas se concatenan y combinan, proporcionando

una representación más rica y completa del contexto.

El proceso de calcular esos valores de importancia requiere de 3 elementos: los vectores Q,K y V. Donde cada W asociada a esos elementos son una matriz de pesos entrenables (Analytics Vidhya, 2020).

Query (Q)

Refieren a los embeddings de los tokens de la secuencia de entrada y puede entenderse como lo que se está buscando. Siguiendo con el ejemplo de la frase anterior, para el vector correspondiente con el token ".El", este valor se calcula:

$$W_{EI} * W_Q$$

Key (K)

Se entiende como lo que ofrece cada token. Por ejemplo, el token "perro" puede ofrecer: "sustantivo, sujeto, animal, etc.". Siguiendo con el ejemplo de la frase anterior, para el vector correspondiente con el token ".El", este valor se calcula:

$$W_{EI} * W_K$$

Value (V)

La información real que se transmite, si es relevante. Siguiendo con el ejemplo de la frase anterior, para el vector correspondiente con el token ".El", este valor se calcula:

$$W_{EI} * W_V$$

El proceso de atención se puede comparar con la búsqueda de un video en YouTube. Esta plataforma almacena sus videos en un diccionario clave-valor, siendo la clave el nombre. Cuando se realiza una búsqueda (Query) se calcula la similitud con esas claves (Key) para devolver el resultado.

Como punto de partida se usa el vector de embeddings calculado previamente, y en base a este se calcularían los vectores anteriores, que se pueden entender como 3 versiones de dicho embedding. Para su cálculo, se aplican transformaciones lineales que encuentran la mejor combinación de pesos y tras esto se calcula la atención en base a la siguiente expresión:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) V$$

$$QK^T$$

Matriz de similitudes. Con *dot product*, el valor es grande si los vectores son similares y pequeño si apuntan en distinta dirección.

$$\text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) V$$

Como el *dot product* puede dar valores grandes, que hagan que softmax se concentre en ellos, se normaliza con $\sqrt{d_k}$ para estabilizar. La función *softmax* produce una distribución de probabilidades y, al multiplicar por V , se obtiene una combinación ponderada que añade contexto a los embeddings.

En estos mecanismos de atención se incluye habitualmente el término *masked*, ya que se emplean máscaras con el fin de restringir la información a la que tiene acceso el modelo durante el cálculo de la atención. Estas máscaras permiten, por ejemplo, impedir que el modelo considere tokens futuros que aún no deberían ser visibles en una tarea de generación autoregresiva, o bien ignorar tokens que no contienen información semántica relevante.

Entre los tipos de máscaras más comunes se encuentra la *Padding Mask*, que se aplica para ignorar los tokens de relleno, que no contienen información semántica, añadidos a las secuencias con el fin de que todas tengan la misma longitud dentro de un lote de entrenamiento. También se utiliza la *Sequence Mask*, que sirve para ocultar determinadas partes de la secuencia de acuerdo con un criterio específico. Por último, la *Look-ahead Mask* (o *Causal mask*) se emplea en modelos autoregresivos para evitar que la predicción de un token en la posición t dependa de información futura, garantizando así que las predicciones en una posición concreta solo tengan en cuenta los tokens anteriores o la misma posición (Swarms, 2021).

3.6.3. Add & Norm (Sharma, 2024)

En la arquitectura Transformer aparece reiteradamente una capa denominada *Add & Norm*. La principal finalidad de esta capa es estabilizar y mejorar el entrenamiento y se divide en los dos procesos siguientes:

Conexión Residual (*Add*)

En redes profundas con arquitecturas complejas los pesos pueden tomar valores muy pequeños, hasta el punto de desaparecer (*vanishing gradients*) o demasiado grandes (*exploding gradients*) al retropropagar hacia atrás, lo que conlleva que la red deje de aprender o sea muy inestable. Para solucionar estos problemas, en 2015 nace ResNet una arquitectura en la que la entrada de una capa, pasa a su salida directamente, saltándose las operaciones intermedias. Es decir, a nivel simplificado se tendría:

$$y = F(x) + x$$

Donde x es la entrada de la capa anterior, y la salida de la capa actual y $F(x)$ la función de activación correspondiente a la capa actual.

Con ello, la capa ya no tiene que aprender la función completa $F(x)$, sino solo la diferencia respecto a la identidad. Esto asegura que siempre exista al menos una señal que retroceda hasta las capas iniciales durante el entrenamiento.

Se entiende mejor al calcular las derivadas parciales para actualizar los pesos en *backpropagation*. Como se observa en la siguiente expresión, el gradiente nunca se anula completamente gracias al término de identidad:

$$\frac{\partial \mathcal{E}}{\partial x} = \frac{\partial \mathcal{E}}{\partial y} \frac{\partial y}{\partial x} = \frac{\partial \mathcal{E}}{\partial y} \left(\frac{dF(x)}{dx} + 1 \right).$$

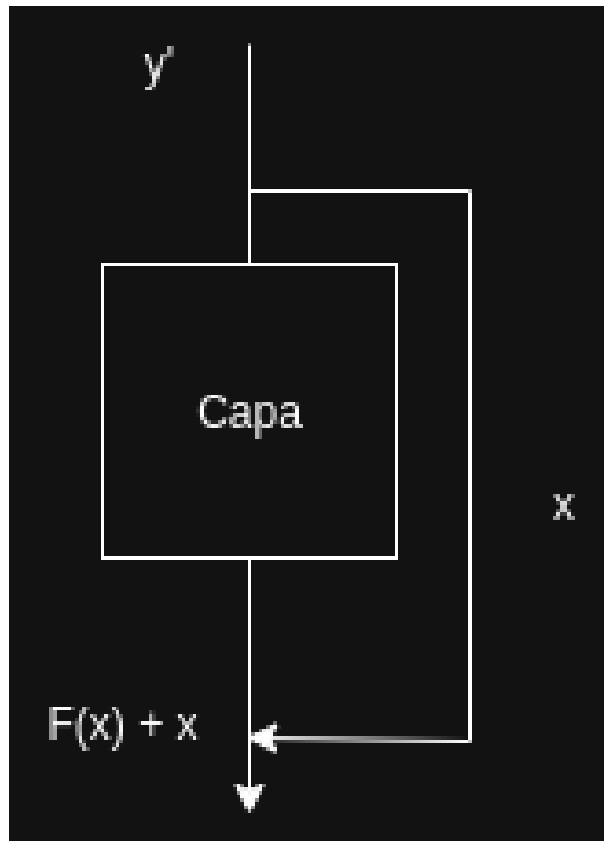


Figura 3.2: Ejemplo gráfico conexión residual

De esta manera, incluso cuando $\frac{dF(x)}{dx}$ tiende a cero, el gradiente conserva la contribución de la identidad (+1). Así, la red es capaz de mantener un flujo estable de información hacia atrás, evitando que los gradientes mueran. Además, puede aprender a ignorar aquellas transformaciones que no sean beneficiosas en el entrenamiento, llevando sus pesos hacia cero y dejando pasar directamente la señal original x (ApXML, 2025).

Normalización (*Layer Norm*)

Tras la conexión residual, se aplica una capa de normalización. A diferencia de otras capas de normalización, como la normalización por lotes (*Batch Norm*), que normaliza los datos a lo largo del lote de entrenamiento, la normalización por capas (*Layer Norm*) lo hace para cada muestra individual en sus características internas. Esto es especialmente importante en NLP, donde las secuencias de entrada pueden variar en longitud y contenido y no interesa que cada una se vea afectada por estadísticas de otra. Además, si se usase la normalización por lotes, se haría *padding*, y los valores de relleno afectarían a las estadísticas de la normalización, especialmente en frases cortas. A continuación, se muestra un ejemplo que plasma claramente la diferencia entre ambas técnicas:

Dadas las siguiente secuencias:

$$\text{secuencia}_1 = [2, 3], \quad \text{secuencia}_2 = [4, 5]$$

La normalización por lotes se aplica a lo largo del batch:

$$\mu_1 = \frac{2+4}{2} = 3, \quad \sigma_1 = \sqrt{\frac{(2-3)^2 + (4-3)^2}{2}} = 1$$

Mientras que la normalización por capa se aplica dentro de cada secuencia considerando todas sus características, así la posición 1:

$$\mu_{\text{seq1}} = \frac{2+3}{2} = 2,5, \quad \sigma_{\text{seq1}} = \sqrt{\frac{(2-2,5)^2 + (3-2,5)^2}{2}} = 0,5$$

$$\mu_{\text{seq2}} = \frac{4+5}{2} = 4,5, \quad \sigma_{\text{seq2}} = \sqrt{\frac{(4-4,5)^2 + (5-4,5)^2}{2}} = 0,5$$

Con ello, en base a estas dos operaciones, de media y varianza, se realiza la normalización de los valores de entrada en cada capa aplicando la siguiente expresión:

$$\hat{x}_i = \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}} \cdot \gamma + \beta \quad (3.1)$$

En dicha expresión se observan dos parámetros entrenables, γ y β , que permiten al modelo aprender la escala y el desplazamiento óptimos para cada característica. Sin estos parámetros, la normalización es demasiado restrictiva, pues los resultados se aplanan siempre a un mismo rango. El término ϵ es un valor pequeño añadido para evitar divisiones por cero y garantizar la estabilidad numérica.

Otra de las ventajas de normalizar es que mitiga el *internal covariate shift* (Sharma, 2024). Durante el entrenamiento las distribuciones de las activaciones cambian, dado que los pesos cambian, lo que ralentiza la convergencia (ApXML, 2025), haciendo el entrenamiento más inestable y el optimizador pierde tiempo corrigiendo desajustes. Gracias a la normalización, las variables se mantienen en una escala controlada.

3.6.4. Feed Forward

En términos generales se trata del proceso por el que la información pasa a través de una red, únicamente en sentido hacia delante, desde su entrada hasta su salida, pasando por todas las capas ocultas que hubiese.

Se define como una capa o bloque que toma la salida de la capa de atención y la procesa a través de una red neuronal densa, esto es, totalmente conectada. Esta red consta de dos capas lineales con una función de activación no lineal entre ellas, que en el paper original es ReLU. La primera capa expande la dimensionalidad del embedding, mientras que la segunda la reduce de nuevo a su tamaño original. Esto permite al modelo aprender representaciones más complejas y captar interacciones no lineales, lo cual es un factor importante pues permite, por ejemplo, que el modelo entienda el contexto y significado de una palabra en función de donde aparece en la frase (Gomez, 2023).

Pese a que originalmente se usaban 2 capas lineales con ReLU, en la actualidad se ha demostrado una mejora significativa al aumentar el número de capas, obteniendo un entrenamiento más estable, pues se vió que el aprendizaje al inicio era mayor, y reduciendo el problema por la desaparición del gradiente.

Además, se ha demostrado ampliamente que es un bloque fundamental, pues en varios experimentos, al eliminar esta capa, el rendimiento del modelo disminuye drásticamente (Gerber, [2025](#)).

3.6.5. Linear y softmax

Las últimas capas del modelo son una capa lineal y softmax. A través de la primera se obtienen los denominados *logits*, que son puntuaciones sin normalizar sobre que tan probable es cada token para el modelo. La función de esta capa es transformar los embeddings de alta dimensión en una representación que pueda ser interpretada como una distribución de probabilidad sobre el vocabulario.

A través de *softmax*, se transforman estas puntuaciones en una distribución de probabilidad, donde cada valor representa la probabilidad de que un token específico sea el siguiente en la secuencia generada.

Capítulo 4

Resultados

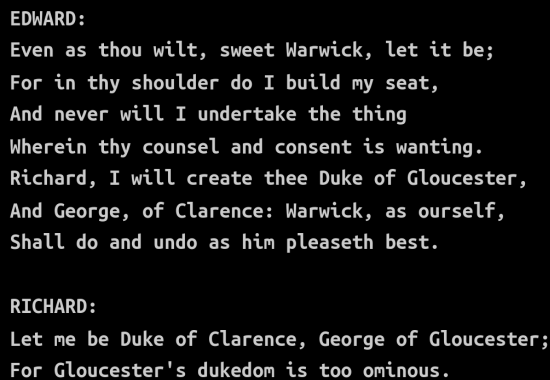
4.1. Adquisición y análisis de las fuentes de datos

En este proyecto se han seleccionado dos datasets distintos: Tiny Shakespeare y WikiText2.

4.1.1. Tiny Shakespeare

Se trata de un dataset creado por Andrej Karpathy compuesto por 40,000 líneas con las obras de Shakespeare. El dataset está disponible en HuggingFace (HuggingFace, [2019a](#)) y puede obtenerse con el comando *wget* de la siguiente forma:

```
wget https://raw.githubusercontent.com/karpathy/char-rnn/master/data/tinyshakespeare/input
-0 tinyshakespeare.txt
```

 A continuación se adjunta una porción del dataset original:

```
EDWARD:
Even as thou wilt, sweet Warwick, let it be;
For in thy shoulder do I build my seat,
And never will I undertake the thing
Wherein thy counsel and consent is wanting.
Richard, I will create thee Duke of Gloucester,
And George, of Clarence: Warwick, as ourself,
Shall do and undo as him pleaseth best.

RICHARD:
Let me be Duke of Clarence, George of Gloucester;
For Gloucester's dukedom is too ominous.
```

Figura 4.1: Fragmento del dataset Tiny Shakespeare

EDA

4.1.2. WikiText2

Este dataset contiene una colección de textos seleccionados de Wikipedia (HuggingFace, 2019b) (AutoNLP, 2020). Se puede descargar de Salesforce en Huggingface y consta de más de 2 millones de tokens de datos, divididos en 4358 ejemplos para test, 36718 para entrenamiento y 3760 para validación.

```
wikitext2 = load_dataset("Salesforce/wikitext", "wikitext-2-raw-v1")
```

EDA

4.2. Limpieza y tokenización

En lo referido a la limpieza del texto existen diversas técnicas, como limpiar etiquetas HTML, caracteres especiales, stop words, duplicados, convertir a minúsculas, solucionar problemas de encoding, lematización o corrección de palabras (librerías de Python como SpellChecker permiten realizar esta tarea) (Shabbir, 2021). En este proyecto, dado que los datasets empleados ya están correctamente procesados y se va a utilizar un tokenizador Byte-Pair encoding (BPE) a través de SentencePiece, no es necesario un pre-procesamiento demasiado exhaustivo, se limita a corrección de posibles problemas de codificación utilizando `ftfy`, espacios en blanco y normalización de comillas y guiones usando `unicode`.

```
def light_clean_fn(example):
    t = example["text"]
    t = ftfy.fix_text(t)
    t = unicodedata.normalize("NFKC", t)
    t = (t.replace("'", "'").replace('"', '"').replace(" ", " ")
        .replace("-", "-").replace("_", "_"))
    t = re.sub(r"[ \t]+", " ", t)
    t = re.sub(r"\s*\n\s*", "\n", t)
    t = t.strip("\n")
    return {"text": t}
```

Figura 4.2: Limpieza antes de tokenizar

4.2.1. Implementación del tokenizador BPE

Una vez el texto está preparado se inicia la implementación del proceso de tokenización. El primer paso es entrenar el tokenizador, de forma que aprenda el vocabulario y a dividir en subpalabras. Se establece el tipo de modelo, "bpe".^{en} este caso, el tamaño del vocabulario final, la porción de caracteres del corpus que debe cubrir el modelo (1 para cubrir todos) y el número de hilos de CPU a usar durante el entrenamiento. Además, se utiliza el parámetro `byte_fallback` para controlar los caracteres desconocidos, tal y como se ha explicado previamente. Se aplica una normalización NFKC (descompone los caracteres en formas básicas y los recompone siguiendo una forma estándar) antes de entrenar donde se estandarizan caracteres y eliminan espacios innecesarios o múltiples. Esto es algo que conviene usar con lenguajes como el japonés que utilizan caracteres desnormalizados o textos sucios y desnormalizados, sin embargo,

en este caso, los datasets empleados son en inglés y ya están listos para usarse y se podría evitar. Por último, se asignan los IDs especiales:

- pad: token de padding para rellenar secuencias de distinta longitud.
- unk: Tokens desconocidos, pese a estar controlados con el parámetro mencionado.
- bos: Token de inicio de secuencia, para dar consistencia.
- eos: Token de fin de secuencia. Permite al modelo a aprender a parar.

```
def tokenizador(dataset):
    if not os.path.exists(TOK_MODEL_PATH):
        print("Entrenamiento modelo BPE")
        # Aprende el vocabulario y como dividir en subpalabras
        spm.SentencePieceTrainer.Train(
            input=train_path, # Corpus de train, se pueden pasar varios
            model_prefix="./resources/models/bpe_model_shakespeare",
            vocab_size= 8000,
            model_type="bpe",
            character_coverage=1.0, # inglés
            byte_fallback=True, # evita UNK en caracteres raros
            normalization_rule_name="nfkc", # Normalización previa
            remove_extra_whitespace=True, # colapsa espacios extra
            num_threads=os.cpu_count(),
            pad_id=0, unk_id=1, bos_id=2, eos_id=3
        )
    sp = spm.SentencePieceProcessor(model_file=TOK_MODEL_PATH)
    tok_ids = sp.encode(dataset, out_type=int)
    print("Número total de tokens:", len(tok_ids))
    return tok_ids, sp
```

Figura 4.3: Enter Caption

El entrenamiento genera dos archivos que contienen el modelo de segmentación y el vocabulario aprendido. Posteriormente, se inicializa un objeto `SentencePieceProcessor` a partir del modelo entrenado, y se procede a tokenizar los conjuntos de datos (entrenamiento, validación y prueba) mediante el método `sp.encode`, que devuelve una lista de identificadores numéricos correspondientes a las subpalabras detectadas en el texto.

Finalmente, la función imprime el número total de tokens generados y devuelve tanto la secuencia de identificadores (`tok_ids`) como el objeto `SentencePieceProcessor`, que permitirá revertir la tokenización o aplicarla en futuros conjuntos de datos.

```
pre_clean_dataset(tinishakespeare)

train_text, valid_text, test_text = read_datasets()

print(f"Longitud corpus train: {len(train_text)} caracteres")
print(f"Longitud corpus valid: {len(valid_text)} caracteres")
print(f"Longitud corpus test: {len(test_text)} caracteres")

train_ids, sp = tokenizador(train_text)
val_ids, _ = tokenizador(valid_text)

vocab_size = sp.vocab_size()
```

Figura 4.4: Enter Caption

4.3. Embeddings y Positional Encodding

Cualquier modelo basado en redes neuronales requiere que sus entradas sean valores numéricos, no arbitrarios, para poder trabajar con el lenguaje humano. Es cierto que los tokens del texto ya se han convertido a índices enteros usando técnicas de tokenización, no obstante, estos valores no son suficientes para que el modelo sea capaz de inferir relaciones semánticas entre las palabras y llegar a entender su significado, es por ello que cada token se representa mediante un vector denso de características, conocido como *embedding*.

Para mapear los índices resultantes de la tokenización a una matriz de *embeddings*, se implementa una capa de embedding utilizando PyTorch. Esta capa actúa como una tabla de búsqueda que asocia cada índice con un vector de características. Internamente, se trata de una matriz de pesos que se optimiza durante el proceso de entrenamiento del modelo.

```
# Token embeddings
embedding_layer = nn.Embedding(
    num_embeddings=vocab_size,
    embedding_dim=embedding_dim
).to(device)

# Positional embeddings aprendidos
pos_embedding_layer = nn.Embedding(
    num_embeddings=context_len,
    embedding_dim=embedding_dim
).to(device)
```

Figura 4.5: Embeddings

Cabe destacar que inicialmente la matriz de embeddings es aleatoria. A medida que se entrena el modelo, estos vectores se ajustan mediante retropropagación, lo que reduce la función de pérdida y permite que los embeddings capturen mejor la semántica, el contexto y otras relaciones relevantes entre los tokens. El resultado final es un tensor en el que cada fila representa un vector denso asociado a un token del vocabulario, adaptado a la tarea específica (Bao, 2022). Se ha decidido inicializar los pesos siguiendo una inicialización Normal (ver sección A.5 del Anexo), lo que permite que inicialmente sean valores cercanos a 0, evitando que algunos tokens tengan valores demasiado grandes.

```
nn.init.normal_(m.weight, mean=0.0, std=0.02)
```

Figura 4.6: Inicialización Normal

4.4. Mecanismos de atención

La base de la arquitectura Transformer es la atención. A través de la atención, cada palabra mira simultáneamente al resto y calcula una métrica de relevancia a través de la cual, se determina que fracción de la información real es relevante para cada token y cpor tanto, para el modelo.

Como se ha mencionado previamente, se basa en 3 matrices entrenables, inicializadas originalmente con valores aleatorios, y una operación, softmax, que calcula una distribución de probabilidad para cada token (ver section 3.6.2).

Usando PyTorch es posible trabajar de forma sencilla con tensores y replicar toda la lógica operacional. En primer lugar, se implementa un mecanismo de atención aplicando una máscara causal, a continuación se desarrolla la lógica de múltiples cabezas para capturar distintas relaciones entre los tokens.

El primer punto es crear las matrices de pesos:

$$W_Q, W_K, W_V$$

Aquí, PyTorch ofrece el módulo `nn.Linear`, que lo que hace es aplicar una transformación lineal a los datos de entrada (Popovic, 2023), inicializando aleatoriamente una matriz de pesos y un vector de sesgos, el cual, en este caso, no se va a inicializar pues no interesa que los vectores se desplacen en el espacio. Este módulo, toma dos parámetros: `in_features` y `out_features`, y la matriz resultante tiene un tamaño: `in_features * out_features`.

De este modo, a partir de la misma entrada $x \in \mathbb{R}^{B \times T \times d_{model}}$ (secuencia de embeddings), se obtienen tres vistas distintas que serán usadas posteriormente en el cálculo de la atención.

```
Q = self.Wq(x) # (B, T, d_model)
K = self.Wk(x) # (B, T, d_model)
V = self.Wv(x) # (B, T, d_model)
```

Figura 4.7: Proyección matrices Q, K, V

Los siguientes pasos son crear la métrica de atención, normalizar con softmax y aplicar los *scores* a la información real. Con PyTorch es muy sencillo de realizar a través de las funciones `matmul`, que permite multiplicar tensores, y `softmax`.

```
metrica_atencion = torch.matmul(Q, K.transpose(-2, -1)) / math.sqrt(self.d_k)
atencion = torch.softmax(metrica_atencion, dim=-1)
res_atencion = torch.matmul(atencion, V)
```

Figura 4.8: Implementación Self Attention

Por último, queda aplicar la máscara de atención. Dado que el modelo implementado es de carácter

autorregresivo y emplea ventanas de tamaño fijo sin añadir *padding*, únicamente es necesaria la máscara causal. Su implementación resulta sencilla: se construye una matriz triangular superior de unos, cuyos valores se reemplazan por $-\infty$. Esta operación se aplica antes de la normalización con la función *softmax*, de manera que las posiciones enmascaradas reciben una probabilidad prácticamente nula y, en la práctica, el modelo las ignora.

Por ejemplo, para una secuencia de longitud $T = 4$, la matriz de *metrica_atencion* S se transforma en:

$$metrica_atencion' = \begin{bmatrix} s_{11} & -\infty & -\infty & -\infty \\ s_{21} & s_{22} & -\infty & -\infty \\ s_{31} & s_{32} & s_{33} & -\infty \\ s_{41} & s_{42} & s_{43} & s_{44} \end{bmatrix}$$

De este modo, cada token únicamente puede atenderse a sí mismo y a los anteriores, lo que garantiza la naturaleza autorregresiva del modelo.

```
mask = torch.triu(torch.ones(T, T, device=x.device), diagonal=1).bool()
metrica_atencion = metrica_atencion.masked_fill(mask, float('-inf'))
```

Figura 4.9: Implementación máscara causal

4.4.1. Implementación de las múltiples cabezas de atención

Para poder aplicar la atención múltiple las matrices originales Q , K , V se reorganizan en múltiples cabezas. Para ello, se emplea la función *view* de PyTorch, que permite reorganizar el tensor sin modificar sus datos en memoria. En este caso, se pasa de una estructura de tamaño (B, T, d_{model}) a otra de tamaño (B, num_heads, T, d_k) , donde $d_k = d_{model}/num_heads$ es la dimensión de cada cabeza. Posteriormente, se aplica una transposición para situar la dimensión de las cabezas en la segunda posición:

```
Q = Q.view(B, T, self.num_heads, self.d_k).transpose(1, 2)
K = K.view(B, T, self.num_heads, self.d_k).transpose(1, 2)
V = V.view(B, T, self.num_heads, self.d_k).transpose(1, 2)
```

Figura 4.10: Implementación Multi-Head Attention

De este modo, cada cabeza de atención opera de manera independiente sobre un subespacio de dimensión d_k , calculando sus propios valores de similitud (QK^T) y generando una salida parcial. Finalmente, las salidas de todas las cabezas se concatenan de nuevo y se proyectan mediante una capa lineal adicional, como se puede ver en la imagen 4.11.

Por último, mencionar que la inicialización de los pesos se realiza mediante el método *Xavier Uniform* (ver sección A.4 del Anexo), que estabiliza el entrenamiento en capas profundas.


```

self.Wo = nn.Linear(d_model, d_model, bias=False)

out = out.transpose(1, 2).contiguous().view(B, T, self.d_model)
out = self.Wo(out)

```

Figura 4.11: Salida Multi-Head Attention

4.5. Normalización (Add & Norm)

Como se ha explicado en 3.6.3, esta capa consta de dos elementos. Primero, una conexión residual que suma la entrada original a la salida del bloque de atención múltiple. En segundo lugar, una capa de normalización por capas que estabiliza y acelera el entrenamiento. En la imagen 4.12 se visualiza mejor el flujo.

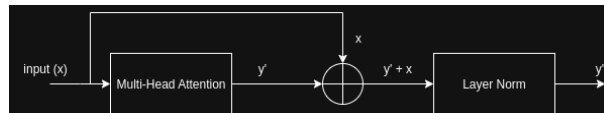


Figura 4.12: Flujo capa Add & Norm

Como se puede ver en la imagen 4.13, la conexión residual se realiza mediante una simple suma de tensores. La capa de normalización se ha implementado de cero, aunque PyTorch ofrece el módulo `nn.LayerNorm`, que realiza la misma función. En la implementación se calcula la media y la varianza por posición y devuelve el resultado de la expresión (3.1).

Hay que señalar aquí el uso de `nn.Parameter` para los parámetros entrenables γ y β . Se tratan como tensores especiales y se registran como parámetros entrenables del modelo, en concreto, se asocian a este módulo.

```

class NormLayer(nn.Module):
    def __init__(self, normalized_shape, eps=1e-5):
        super().__init__()
        self.gamma = nn.Parameter(torch.ones(normalized_shape))
        self.beta = nn.Parameter(torch.zeros(normalized_shape))
        self.eps = eps

    def forward(self, X):
        # X: (batch, seq_len, hidden_dim)
        mean = X.mean(dim=-1, keepdim=True) # media por posición
        var = X.var(dim=-1, keepdim=True, unbiased=False) # varianza
        X_hat = (X - mean) / torch.sqrt(var + self.eps) # normaliza
        return self.gamma * X_hat + self.beta

class AddNorm(nn.Module):
    def __init__(self, norm_shape, dropout):
        super().__init__()
        self.dropout = nn.Dropout(dropout) # Para regularizar
        self.ln = NormLayer(norm_shape) # nn.LayerNorm(norm_shape)

    def forward(self, X, V): # Y es la salida de la subcapa previa y X la entrada a la subcapa
        return self.ln(self.dropout(V) + X) # Aplica add y luego layernorm

```

Figura 4.13: Implementación Add & Norm

4.6. Feed Forward

Tal y como se ha explicado en la sección 3.6.4, el rendimiento del transformer es significativamente mejor al ampliar el número de capas lineales. Siguiendo las recomendaciones del paper (Gerber, 2025), se ha decidido implementar 3 capas utilizando como función de activación *GELU* (ver sección A.3 del Anexo), que ha demostrado ser más efectiva que ReLU en este tipo de arquitecturas. La inicialización de los pesos se realiza mediante el método *Xavier Uniform* al igual que en las capas de atención, para mantener la estabilidad del entrenamiento.

```
nn.init.xavier_uniform_(m.weight)
```

Figura 4.14: Inicialización Xavier Uniform

La implementación es relativamente sencilla, se haría como si se tratara de un perceptrón multicapa (Gomez, 2023), donde se incluyen 3 capas lineales y dos funciones de activación GELU entre ellas. Además, se introduce *dropout* para evitar sobreajuste, apagando aleatoriamente algunas neuronas durante el entrenamiento.

```
class FeedForward(nn.Module):
    def __init__(self, d_model, d_ff, dropout):
        super().__init__()
        # Se usan 3 capas densas, con dropout y activación GELU
        self.linear1 = nn.Linear(d_model, d_ff)
        self.linear2 = nn.Linear(d_ff, d_ff)
        self.linear3 = nn.Linear(d_ff, d_model)
        self.gelu = nn.GELU()
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        x = self.linear1(x)
        x = self.gelu(x)
        x = self.dropout(x)
        x = self.linear2(x)
        x = self.gelu(x)
        x = self.dropout(x)
        x = self.linear3(x)
        return x
```

Figura 4.15: Capa FeedForward

4.7. Entrenamiento

Tras añadir la capa lineal de salida al final del bloque Transformer, se completan los componentes fundamentales de la arquitectura. El siguiente paso consiste en definir los parámetros y establecer la infraestructura necesaria para el entrenamiento del modelo.

```

class MiniLLM(nn.Module):
    def __init__(self, vocab_size, d_model=512, num_heads=8, d_ff=2048, num_layers=6, context_len=256, dropout=0.1):
        super().__init__()
        self.tok_emb = nn.Embedding(vocab_size, d_model)
        self.pos_emb = nn.Embedding(context_len, d_model)

        self.blocks = nn.ModuleList([
            TransformerDecoderOnlyBlock(num_heads, d_model, d_ff, dropout)
            for _ in range(num_layers)
        ])

        self.norm = NormLayer(d_model)
        self.ln_head = nn.Linear(d_model, vocab_size, bias=False)

    def forward(self, idxs):
        B, T = idxs.shape

        pos = torch.arange(T, device=idxs.device).unsqueeze(0).expand(B, T)
        x = self.tok_emb(idxs) * math.sqrt(self.tok_emb.embedding_dim)
        x = x + self.pos_emb(pos)

        # x = self.tok_emb(idxs) + self.pos_emb(pos)
        for block in self.blocks:
            x = block(x)
        x = self.norm(x)
        logits = self.ln_head(x)
        return logits

```

Figura 4.16: Modelo final

4.7.1. Configuración de parámetros de entrenamiento

En lo referido a la configuración de parámetros se han seguido inicialmente las recomendaciones dadas en el *paper: Attention Is Not All You Need* (Gerbner, 2025), teniendo en cuenta las características de este proyecto, tanto a nivel de *hardware* como de la arquitectura implementada. No obstante, como se explicará en la sección 5.1, se han realizado ciertos ajustes para mejorar el entrenamiento.

Para el prototipo inicial, entrenado con el dataset *Tiny Shakespeare*, se ha definido un vocabulario compuesto por 16000 tokens, el tamaño de los embeddings es de 512 tokens y la ventana de contexto de 256. Se han utilizado 8 cabezas de atención, para capturar bien distintos patrones y el número de capas del *transformer* es de 6, un número no muy elevado por las limitaciones técnicas del proyecto. En cuanto al bloque *FeedForward*, se define la dimensión d_{ff} con un valor de 2048, expandiendo cada vector de 512 a 2048 dimensiones. Para no saturar la memoria de la GPU el número de batches se ha mantenido en 64, pese a incluir ciertos elementos regulatorios que se explicarán a continuación. Por último, se ha entrenado para 20 épocas.

4.7.2. Elementos regulatorios

Para el entrenamiento del modelo se ha planteado inicialmente la técnica *Mixed Precision Training*. Esto es porque se ha visto que reduce el uso de memoria y acelera el entrenamiento sin comprometer la precisión del modelo, lo que permite aumentar el tamaño de los batches y no desbordar la memoria de la GPU. En esencia, se utilizan tipos de datos de menor precisión (como `float16`) para las operaciones que no requieren alta precisión, mientras que las operaciones críticas se mantienen en `float32`.

Para una primera implementación se utilizan dos elementos clave. Por un lado `GradScaler`, que escala los valores de la función de pérdida dinámicamente para prevenir el desbordamiento de los gradientes cuando se usan tipos de datos de menor precisión. Lo que hace es multiplicar la pérdida por un factor grande antes de calcular los gradientes. En cada iteración se decide si ese factor fue bueno o malo, en base a si ha habido desbordamiento o no. Si no hubo desbordamiento, se aumenta ligeramente el factor para incrementar la precisión. En caso contrario, se reduce el factor para la próxima iteración (Amit, 2024). Por

otro lado, `autocast`, que permite especificar bloques de código donde se desea utilizar precisión mixta (StackOverflow users, 2022a). Dentro de estos bloques, las operaciones se ejecutan automáticamente en la precisión más adecuada según el hardware y la operación específica. No obstante, hay casos en los que se debe indicar a `autocast` de forma manual que no modifique la precisión de los datos, por ejemplo, en la capa de normalización, donde al no ser nativa de PyTorch, pues se ha implementado desde 0, quizá no lo detecta automáticamente (Amit, 2024). También se ha desactivado en la capa de `Softmax` pues son capas donde escalar puede introducir inestabilidad al modelo.

En combinación con estas técnicas, se ha aplicado `Gradient Clipping` para evitar que los gradientes se vuelvan demasiado grandes y provoquen actualizaciones inestables de los pesos. Lo que hace es reducir su magnitud, sin alterar la dirección.

No obstante, al entrenar el modelo, se han obtenido valores nulos en la función de pérdida, por lo que se ha decidido prescindir de `GradScaler` y `autocast`, haciendo ajustes en los hiperparámetros del optimizador e implementando `CosineAnnealingLR` (PyTorch, 2025a), con ello lo que se logra es reducir el ratio de aprendizaje por época y lote de entrenamiento. Pese a ello, no se ha observado una mejora significativa en los datos de validación, por lo que se ha decidido implementar un proceso de *warm-up* empleando la clase de PyTorch `LambdaLR`. Todo este proceso de ajuste de hiperparámetros se presenta mejor en la sección 5.1.

4.7.3. Función de pérdida

En este tipo de modelos, las métricas tradicionales no capturan adecuadamente los matices que determinan el rendimiento del modelo. Por ello, se ha optado por utilizar la *Cross Entropy Loss* como función de pérdida. Esta métrica mide la diferencia entre la distribución de probabilidad predicha por el modelo y la distribución real de los datos. Matemáticamente, se define como:

$$\mathcal{L}_{CE} = - \sum_{w \in \mathcal{V}} Y_t[w] \log \hat{Y}_t[w],$$

donde \mathcal{V} representa el vocabulario del modelo, $Y_t[w]$ es la distribución real del token correcto en el paso temporal t y $\hat{Y}_t[w]$ es la probabilidad predicha para dicho token tras aplicar la función *softmax*.

Durante la implementación, se emplea la función `nn.CrossEntropyLoss()` de PyTorch, que combina internamente la operación *softmax* y el cálculo de la entropía (ver sección A.5 del Anexo) en un solo paso:

```
loss = loss_fn(
    logits.view(-1, logits.size(-1)),
    batch_y.view(-1)
)
```

Figura 4.17: Función de pérdida del modelo (GeeksforGeeks, 2025b)

Dado que la función `CrossEntropyLoss` recibe como entrada dos tensores, uno con las predicciones del modelo (*logits*) y otro con las etiquetas reales, sus dimensiones deben ajustarse al formato esperado. El primer tensor debe tener forma `[N, C]`, donde `N` representa el número total de tokens que el modelo intenta predecir y `C` corresponde al número de posibles tokens del vocabulario. El segundo tensor debe tener forma `[N]`, conteniendo para cada posición el índice del token correcto. Por ello, los tensores se aplanan mediante `.view(-1, logits.size(-1))` para los *logits* y `.view(-1)` para las etiquetas reales.

A través de esta función de pérdida, se calcula la perplejidad (Keerthanams, 2025, May 8), que mide la incertidumbre al predecir la siguiente palabra. Por ejemplo, una perplejidad de 10 indica el modelo, en promedio, es tan incierto como si tuviera que elegir uniformemente entre 10 opciones. Matemáticamente, se define como:

$$\text{PPL} = e^{\mathcal{L}_{CE}}$$

Una perplejidad baja indica que el modelo es bueno prediciendo la siguiente palabra, mientras que una alta sugiere que el modelo está confundido. Durante el entrenamiento, se monitoriza la perplejidad en los conjuntos de validación para evaluar el progreso del modelo y ajustar hiperparámetros si es necesario.

```
train_ppl = math.exp(avg_loss)
```

Figura 4.18: Calculo de perplejidad en PyTorch (StackOverflow users, 2019)

El problema de la perplejidad (Singh, 2024) es que depende altamente del vocabulario, con lo que puede llegar a darse que un modelo bueno parezca malo debido a palabras poco comunes en vocabularios extensos. En este caso, el vocabulario no es muy extenso y al usar BPE se reduce el número de palabras poco comunes o desconocidas, con lo que se ha considerado una métrica adecuada. Por otro lado, no valida la corrección semántica o contextual, por lo que a la hora de validar el modelo se han planteado otras técnicas como MAUVE (Pillutla & colaboradores, 2025) (ver sección A.7 del Anexo) y Distinct-n (ver sección A.8 del Anexo), además de la validación humana configurando la temperatura, como se explicará en la sección ??.

Por último, se ha implementado un mecanismo del tipo *Early Stopping* por el cual si la perplejidad en los datos de validación no mejora durante un periodo consecutivo se finaliza el entrenamiento. Con esto se pretende evitar sobreajuste y que el modelo se ajuste demasiado bien a los datos de entrenamiento respecto a los de validación.

4.7.4. Optimizadores

Se han probado 3 optimizadores distintos, pero a la vez altamente relacionados. En primer lugar el optimizador Stochastic Gradient Descent (SGD), quien introdujo un cambio respecto al tradicional descenso del gradiente, y es que el gradiente no se calcula sobre todo el conjunto de datos original, sino sobre una muestra aleatoria del conjunto de entrenamiento (GeeksforGeeks, 2025a). Ciertas investigaciones indican que pese a utilizar un ratio de aprendizaje constante, lo que puede ocasionar una convergencia

```

if val_loss < best_val_loss:
    best_val_loss = val_loss
    patience_counter = 0
else:
    patience_counter += 1
    if patience_counter >= patience_limit:
        print(f"Early stopping en epoch: {epoch+1}")
        break

```

Figura 4.19: Calculo de perplexidad en PyTorch (StackOverflow users, 2019)

lenta, generaliza mejor que Adam y es bueno evitando el sobreajuste (Ultralytics, 2025).

Otro de los optimizadores es Root Mean Square Propagation (RMSProp). Este optimizador mejora su rendimiento para funciones de pérdida no convexas, ajustando el ratio de aprendizaje, dado que un solo valor no funciona igual de bien para cada parámetro, haciendo que algunos pesos apenas cambien y otros cambien demasiado. Da estabilidad (Kashyap, 2024, Nov 2).

Por último, se ha utilizado AdamW. Es un optimizador utilizado en GPT por su alta capacidad de generalización. Se diferencia de Adam en que desacopla el decaimiento del peso *weight decay*, una regularización que castiga pesos grandes para evitar sobreajustes, del proceso de actualización del gradiente (DataCamp, 2024). Al igual que Adam, se basa en dos operaciones (Yassin, 2024, Nov 8), por un lado el impulso o *momentum*, y por otro lado el uso de un *learning rate* adaptativo. El primer factor implica que no solo se considere el punto actual en cada paso durante el descenso por gradiente, si no que considera los pasados, así, se logran movimientos más suaves y un descenso más rápido y estable. El segundo factor hace que los ratios de aprendizaje se ajusten en base a los gradientes de los parámetros (RMSProp).

La implementación es muy simple y la dificultad reside más en ajustar los hiperparámetros. En el caso de AdamW se han seguido las recomendaciones dadas en (DataCamp, 2024), que indica partir con un *learning rate* de 10^{-3} y un *weight decay* de 10^{-2} a 10^{-4} , siendo mayor para modelos más complejos.

```

optimizer_sgd = torch.optim.SGD(model.parameters(), lr=1e-3)
optimizer_adamw = torch.optim.AdamW(model.parameters(), lr=1e-3, weight_decay=1e-2)
optimizer_rmsprop = torch.optim.RMSprop(model.parameters(), lr=1e-3, weight_decay=1e-2, momentum=0.9)

```

Figura 4.20: Optimizadores (StackOverflow users, 2022b)

4.7.5. Resumen del modelo para Tiny Shakespeare

En primer lugar se tokeniza el dataset de entrenamiento, que contiene 896135 caracteres. Para ello se utiliza un tokenizador *BPE* a través de *SentencePiece*, que es entrenado con estos datos y aprende una reglas de segmentación y un vocabulario. A través de este tokenizador ya entrenado, se tokenizan los datos de entrenamiento, validación y test. Como el tamaño definido para el vocabulario es de 16000 elementos, se obtiene un conjunto de números enteros que representan valores (índices) dentro de ese vocabulario con valores comprendidos entre 0 y 15999. En total se generan 265566 índices para el dataset de entrenamiento.

La función del modelo es dado cada token, predecir el siguiente. Es por ello que un paso previo a

transformar este vector a embeddings se crea un dataset de PyTorch que, junto con *DataLoader* genera ejemplos de secuencias de entrada y objetivos (se desplaza una posición la de entrada) y se agrupan en batches, que irán alimentando las capas de Embedding del modelo. Aquí cabe resaltar que, como el tamaño del batch es de 64, el modelo recibe como entrada tensores con tamaño (64, 256). Para verlo mejor, es como si el modelo recibe: *[El, perro, ladra, en, el]* y busca predecir *[perro, ladra, en, el,prado]*, para cada token.

Cada uno de estos batch alimentan el modelo durante 20 épocas. En primer lugar entran a las capas de embedding. Aquí los vectores de tamaño (64, 256) se convierten a vectores multidimensionales (embeddings) de tamaño (64, 256, 512), es decir, cada token, se convierte en un vector de 512 elementos, y son sumados al *Positional Embedding* para que el modelo sea consciente de la posición de cada token en la secuencia.

Tras esto, los embeddings avanzan a los bloques principales de la arquitectura, que se replican 6 veces. Estos, son los bloques de atención con máscara causal y 8 cabezas, las capas de normalización y adición y Feed Forward, quien expande la entrada de tamaño (64, 256, 512) para añadir no linealidad y la vuelve a comprimir a su tamaño inicial. Estos bloques se apilan a través de `nn.ModuleList`.

Finalmente, tras salir del bloque principal, la salida pasa de nuevo por una capa de normalización y finalmente se obtienen los *logits* gracias a una capa lineal.

El proceso se repite el número de épocas mencionado y se valida con datos de validación y *CrossEntropy*, utilizando la perplejidad para determinar si el modelo aprende o no.

4.8. LSTM

Se ha implementado un modelo LSTM sencillo utilizando *Tensorflow* y entrenado con los mismos datos que el Transformer para contrastar los resultados.

Capítulo 5

Conclusiones

5.1. Primera aproximación con Tiny Shakespeare

Se ha realizado un primer entrenamiento con 20 épocas y un *Early Stopping* con paciencia igual a 10. El optimizador utilizado es AdamW con un ratio de aprendizaje de $1,5 * 10^{-3}$ y *weight decay* igual a 0,01. Además se ha aplicado CosineAnnealingLR para ir ajustando durante el entrenamiento el ratio de aprendizaje, no solo a nivel de época, sino de batch. El resto de parámetros del modelo son:

- **Tamaño vocabulario:** 10000
- **Tamaño embedding:** 512
- **Tamaño contexto:** 256
- **Tamaño de batches:** 64
- **Tamaño FFNN:** 1024
- **Número de batches por época:** 4146
- **Número de cabezas de atención:** 8
- **Número de capas:** 6
- **Dropout:** 0,1

Con ello, se han entrenado 11 épocas antes del *Early Stopping*, comenzando en la primera época con una pérdida de 6,23 y una perplejidad de 509,94 para datos de entrenamiento y de 586,34, para datos de validación y finalizando con una pérdida de 6,2226 y perplejidad de 504,00 y 598,18 respectivamente. Se ve un descenso en los datos de entrenamiento, frente a un incremento en validación lo que suscita un sobreajuste, además no se aprecia un cambio significativo de los valores a lo largo de las épocas lo que hace intuir que el modelo no está aprendiendo nada significativo.

Para corregirlo se ha reducido el tamaño del vocabulario y el modelo, pues al ser un dataset pequeño, sin gran dispersión, se ha visto que esto da mejores resultados. Además, se ha ajustado el optimizador

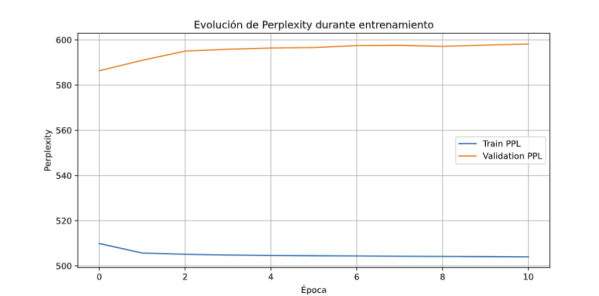


Figura 5.1: Resultado primer entrenamiento

incluyendo como coeficientes de decaimiento exponencial los valores 0,9 y 0,98 (Reddit users, 2023). Para evitar divisiones por cero en el cálculo de los momentos, se ha añadido un valor ϵ igual a 10^{-9} . A esto se ha añadido un *scheduler* basado en la clase LambdaLR de PyTorch (PyTorch, 2025b), el cual permite modificar dinámicamente la tasa de aprendizaje en función del número de pasos de entrenamiento. En este caso, se ha definido una función que implementa una fase de warm-up de 1000 iteraciones, durante la cual la tasa de aprendizaje aumenta linealmente desde 0 hasta su valor máximo inicial. Este procedimiento ayuda a estabilizar el entrenamiento en las primeras etapas, evitando actualizaciones abruptas de los parámetros cuando el modelo todavía no ha alcanzado una representación estable de los gradientes. Por último, otro cambio que se ha llevado a cabo ha sido cambiar el modo de normalización. Se estaba haciendo lo que se conoce como Post-normalización, es decir, la normalización se aplica después de cada subcapa (atención y feed-forward). Sin embargo, ciertos estudios sugieren que la Pre-normalización, donde la normalización se aplica antes de cada subcapa, puede mejorar la estabilidad del entrenamiento (Doherty, 2024, Nov 7).

Los nuevos parámetros del modelo son:

- **Tamaño vocabulario:** 8000
- **Tamaño embedding:** 256
- **Tamaño contexto:** 128
- **Tamaño de batches:** 64
- **Tamaño FFNN:** 1024
- **Número de batches por época:** 4146
- **Número de cabezas de atención:** 8
- **Número de capas:** 3
- **Dropout:** 0,3

Con ello, manteniendo el *Early Stopping* se ha logrado un decaimiento progresivo del modelo en 40 épocas, sin aparecer sobreajuste, obteniendo una pérdida de 3,8240 y una perplejidad de 91,41 en datos de validación y 45,79 en entrenamiento. En la imagen 5.2 se ven mejor los resultados.

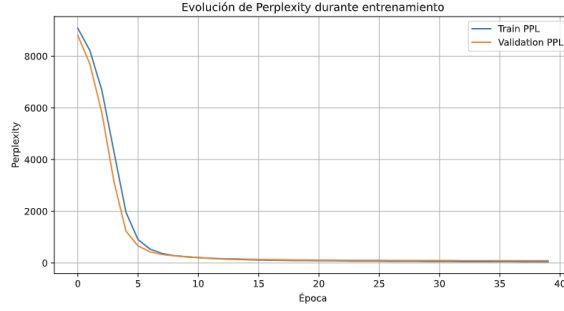


Figura 5.2: Resultado primer entrenamiento

5.2. Validación

Para validar se utiliza el conjunto de datos reservados para tal fin (35327 tokens), los cuales no han sido vistos por el modelo durante el entrenamiento. Se obtiene una pérdida de 4,5039 y una perplejidad de 90,37, de media para un total de 550 lotes. Es buena señal pues indica que el modelo ha aprendido patrones generales y no se ha sobreajustado a los datos de entrenamiento.

Además de una validación con el conjunto de test, se ha implementado un método que consta de distintos hiperparámetros ajustables para generar texto a partir de unos tokens dados, como por ejemplo: «King Charles». Estos hiperparámetros son los siguientes (Vyas, [2025-04-16](#)):

- **Temperatura.** La temperatura ajusta la aleatoriedad en las respuestas del modelo al generar texto. Una temperatura baja hace que el modelo sea más determinista, eligiendo las palabras con mayor probabilidad, pues al dividir por un número menor a 1 se amplifican las diferencias entre los valores, lo que puede resultar en respuestas repetitivas o predecibles. Por otro lado, una temperatura alta introduce más aleatoriedad, permitiendo al modelo explorar opciones menos probables y generando respuestas más variadas y creativas. En sí lo que hace es escalar los logits antes de calcular la distribución de probabilidades con la función *softmax* del siguiente token. Matemáticamente, se expresa como:

$$P(z_i) = \frac{e^{(z_i/T)}}{\sum_j e^{(w_j/T)}}$$

Donde cada z_i es un logit. Aquí a la hora de implementarlo se ha decidido que si la temperatura es 0,0, no se aplica ninguna transformación, pues investigando he visto que se trataría como *argmax*, es decir, siempre se elegiría la palabra con mayor probabilidad (Vishnu Raj, [2025, Jul 27](#)).

- **Top-k.** En lugar de escoger siempre el token con mayor probabilidad, lo que introduce respuestas repetitivas y poco naturales, con este método se seleccionan los k tokens más probables y se normalizan sus probabilidades. Luego, se elige aleatoriamente entre estos k tokens, lo que permite al modelo generar respuestas más variadas y creativas. La implementación se hace a través del método de PyTorch *topk*, que devuelve los k valores más altos de un tensor junto con sus índices. En este caso, se utiliza para filtrar los logits, manteniendo solo los k más altos y estableciendo el resto a $-\infty$, lo que hace que su probabilidad sea prácticamente nula tras aplicar la función *softmax*.
- **Top-p (Núcleo de muestreo).** Se trata de otro método similar a Top-k. En este caso se ordenan los

tokens por probabilidad hasta que la suma acumulada sea mayor o igual a un umbral p . Luego, se normalizan las probabilidades de estos tokens seleccionados y se elige aleatoriamente entre ellos. Para la implementación, al igual que ocurre con el método anterior se ha seguido el ejemplo dado en (Santraigi, 2025). En primer lugar se ordenan los logits de mayor a menor, a continuación, se calcula la suma acumulada de las probabilidades tras aplicar *softmax* mediante la función *cumsum*. Finalmente, se filtran los logits, estableciendo a $-\infty$ aquellos cuya suma acumulada supere el umbral p , lo que hace que su probabilidad sea prácticamente nula tras aplicar la función *softmax*.

- Número máximo de tokens a generar. Se establece un máximo número de tokens a generar para evitar que crezca indefinidamente la respuesta.
- Presence y frequency penalty (Vyas, 2025-04-16). Se trata de dos métodos para reducir o evitar la repetición de frases o tokens previos. El primero penaliza en base a si ha aparecido o no un token previamente y el segundo se basa en el número de veces que ha aparecido. Matemáticamente se expresa como:

$$z_i = z_i(w) - \text{presence_penalty} \cdot \alpha - \text{frequency_penalty} \cdot \beta$$

donde α y β son los coeficientes de penalización para presencia y frecuencia respectivamente. A la hora de implementarlo se calcula esa expresión con tensores, α es un tensor binario que indica si el token ha aparecido o no y β es un tensor con el número de veces que ha aparecido cada token obtenido a través del método *bincount* que cuenta el número de ocurrencias de cada valor en un tensor de enteros no negativo.

5.2.1. Resultados

Se ha probado a generar texto con distintas combinaciones ??, partiendo de unos mismos tokens iniciales: «*KING RICHARD:*» y un número máximo de tokens a generar igual a 100. En la tabla ?? de la sección del anexo A.9 se muestran los resultados.

En primer lugar se han probado distintas temperaturas. Se observa que a medida que la temperatura aumenta, la coherencia se pierde, aunque introduce más variedad. Por ejemplo, para una temperatura de 1,5 introduce al personaje *LOUCESTER*, cuando en realidad es *GLOUCESTER*. Además, aparecen errores de sintaxis y palabras no existentes, como *Ofigeia* o *Theirrah* (parece una composición de *sirrah* u *sirrah*), que se pueden derivar de la tokenización BPE empleada. Con una temperatura baja, se enfatizan los logits más altos, haciendo que el modelo sea más conservador y emplee patrones comunes en el corpus de entrenamiento.

Tras esto se han validado los parámetros *top-k* y *top-p*. Con un valor bajo de K, el texto se mantiene coherente, introduciendo nombres reales de personajes, pero aparecen repeticiones: «*to do not a man, I am not a man*». Al aumentar el valor de dicho parámetro se obtiene más variedad, y a diferencia de la temperatura, se mantiene una coherencia aceptable, sin neologismos, aunque las incongruencias y errores de sintaxis se hacen más notorios, lo que también puede atribuirse a un incremento de la verbosidad. Es cierto que aparecen, por ejemplo, casos como «*PETRUCHIO:SICINIUS:*», pero esto también ocurre en el texto de entrenamiento: «*DUKE VINCENTIO:KING RICHARD III:LEONTES:DUKE VINCENTIO:*». En el caso del parámetro P, la diferencia entre valores bajos y altos llama más la atención. Para un valor

Tabla 5.1: Combinaciones de hiperparámetros para la generación de texto con Tiny Shakespeare

Temp	Top-k	Top-p	Presence Penalty	Frequency Penalty
0.5	None	None	0.0	0.0
1.0	None	None	0.0	0.0
1.5	None	None	0.0	0.0
1.0	10	None	0.0	0.0
1.0	90	None	0.0	0.0
1.0	200	None	0.0	0.0
1.0	None	0.05	0.0	0.0
1.0	None	0.30	0.0	0.0
1.0	None	0.90	0.0	0.0
1.0	None	None	0.2	0.0
1.0	None	None	0.5	0.0
1.0	None	None	1.0	0.0
1.0	None	None	0.0	0.2
1.0	None	None	0.0	0.5
1.0	None	None	0.0	1.0
0.9	100	0.95	0.3	0.2
0.7	80	0.9	0.2	0.3

inferior al 1 % el texto resultante es un bucle de repeticiones de frases carentes de significado como tal: «*And I have been*». Al aumentar el valor aparecen frases con más sentido y variedad, pese a que empiezan a aparecer errores sintácticos como *world'd*. Para un valor del 90 % introduce diálogos y diversidad, no obstante empieza a hacerse menos legible.

El parámetro de penalización de presencia reduce la repetición de palabras y al aumentar su valor hasta un 1,0 se observa que introduce nuevos nombres de personajes, como *ANGELO*, que no había aparecido hasta ahora en el resto de pruebas e introduce signos de exclamación e interrogación, haciendo el texto más natural, aunque los neologismos se mantienen en los tres fragmentos: *goodinate*, *nursetyaish*, *awench* y la coherencia decae.

Finalmente, el parámetro de penalización de frecuencia muestra como al aumentar su valor se pierde el formato del texto (en versos) apareciendo frases mucho más largas, y a diferencia del caso anterior, se aprecia una mejora de coherencia para valores intermedios, aunque para el valor 1,0 se ve como el uso de artículos es penalizado y se eliminan.

Tras comprobar el efecto de cada parámetro se han probado tres combinaciones diferentes. En la primera de ellas prevalecen valores altos de top-k y top-p buscando variedad y creatividad, mientras que la temperatura se mantiene cercana a 1. Las penalizaciones son bajas para no restringir demasiado la generación. Se obtiene un texto con buen cambio de ritmo, expresivo, mezclando personajes y manteniendo un estilo propio de Shakespeare como el texto de entrenamiento, sin neologismos.

En la segunda evaluación se utiliza un valor más bajo de temperatura y disminuye el valor de P y K, lo que lo hace más determinista. Al reducir la penalización de presencia el modelo será más reacio a introducir nuevas palabras y al aumentar la penalización de frecuencia el modelo evitará repetir palabras. El resultado es un texto más coherente, aunque menos variado, sin neologismos o errores sintácticos importantes.

Comparación con LSTM

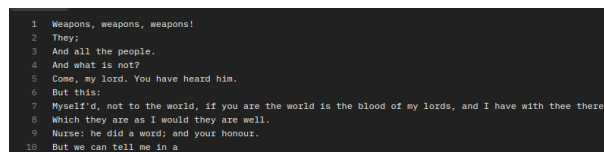
Validación a través de API

Para agilizar la validación del modelo se ha implementado una API RESTful utilizando el framework FastAPI de Python. Esta API permite enviar solicitudes HTTP para generar texto a partir de un prompt inicial y los hiperparámetros definidos anteriormente. Las pruebas se realizan a través de Postman, y facilita ajustar los hiperparámetros dinámicamente y lanzar múltiples pruebas. Se ha expuesto el endpoint localmente a través del puerto 8011. En las imágenes 5.3 y 5.4 se muestra el *curl* generado por Postman y una respuesta de ejemplo respectivamente.



```
cURL v
1 curl --location 'http://127.0.0.1:8011/LLM/api/v1/generar_texto' \
2 --header 'Content-Type: application/json' \
3 --data '{
4   "init_text": "Weapons, weapons, weapons!",
5   "temperatura":0.7,
6   "top_k":80,
7   "top_p":0.9,
8   "max_len":100,
9   "presence_penalty":0.2,
10  "frequency_penalty":0.3
11 }'
```

Figura 5.3: Curl llamada API



```
1 Weapons, weapons, weapons!
2 They;
3 And all the people.
4 And what is not?
5 Come, my lord. You have heard him.
6 But this:
7 Myself'd, not to the world, if you are the world is the blood of my lords, and I have with thee there!
8 Which they are as I would they are well.
9 Nurse: he did a word: and your honour.
10 But we can tell me in a
```

Figura 5.4: Resultado llamada API

Referencias Bibliográficas

- Amit, H. (2024, noviembre). *What Every User Should Know About Mixed Precision Training in PyTorch*. Data Scientist's Diary. <https://medium.com/data-scientists-diary/what-every-user-should-know-about-mixed-precision-training-in-pytorch-63c6544e5a05>
- Analytics Vidhya. (2020). *Understanding Q, K, V in Transformer Self-Attention*. Medium. <https://medium.com/analytics-vidhya/understanding-q-k-v-in-transformer-self-attention-9a5eddaa5960>
- ApXML. (2025). *Add & Norm Layers (Residual Connections)*. ApXML. <https://apxml.com/courses/introduction-to-transformer-models/chapter-3-transformer-encoder-decoder-architecture/add-norm-layers>
- AutoNLP. (2020). *Linked Wikitext-2*. AutoNLP. <https://autonlp.ai/datasets/linked-wikitext-2>
- Bao, G. (2022). *How to use PyTorch's nn.Embedding: A comprehensive guide with examples*. Medium. <https://medium.com/@garybao/how-to-use-pytorchs-nn-embedding-a-comprehensive-guide-with-examples-da00ea42e952>
- DataCamp. (2024). *AdamW Optimizer in PyTorch Tutorial*. DataCamp. <https://www.datacamp.com/tutorial/adamw-optimizer-in-pytorch>
- Doherty, C. (2024, Nov 7). *How Not To Train Your Transformer*. Medium. <https://medium.com/@christopherdoherty14/how-not-to-train-your-transformer-7e63011a16eb>
- Epichka. (2023). *QKV in Transformers*. Epichka Blog. <https://epichka.com/blog/2023/qkv-transformer/>
- GeeksforGeeks. (2021). *Word Embedding in PyTorch*. GeeksforGeeks. <https://www.geeksforgeeks.org/deep-learning/word-embedding-in-pytorch/>
- GeeksforGeeks. (2025a). *ML – Stochastic Gradient Descent (SGD)*. GeeksforGeeks. <https://www.geeksforgeeks.org/machine-learning/ml-stochastic-gradient-descent-sgd/>
- GeeksforGeeks. (2025b). *What Is Cross-Entropy Loss Function?* GeeksforGeeks. <https://www.geeksforgeeks.org/machine-learning/what-is-cross-entropy-loss-function/>
- Gerber, I. (2025). *Attention Is Not All You Need: The Importance of Feedforward Networks in Transformer Models [Submitted 10 May 2025]*. *arXiv preprint arXiv:2505.06633v1*. <https://arxiv.org/html/2505.06633v1>
- Gomez, K. (2023). *The Feedforward Demystified: A Core Operation of Transformers*. Medium. <https://medium.com/@kyeg/the-feedforward-demystified-a-core-operation-of-transformers-afcd3a136c4c>
- Google. (2018). *SentencePiece*. GitHub. <https://github.com/google/sentencepiece>
- HuggingFace. (2019a). *Tiny Shakespeare Dataset*. HuggingFace. https://huggingface.co/datasets/karpathy/tiny_shakespeare

- HuggingFace. (2019b). *WikiText Dataset*. HuggingFace. <https://huggingface.co/datasets/Salesforce/wikitext>
- HuggingFace. (2021). *Tokenizer Summary*. HuggingFace Docs. https://huggingface.co/docs/transformers/es/tokenizer_summary
- Kashyap, P. (2024, Nov 2). *Understanding RMSProp: A Simple Guide to One of Deep Learning's Powerful Optimizers*. Medium. <https://medium.com/@piyushkashyap045/understanding-rmsprop-a-simple-guide-to-one-of-deep-learning's-powerful-optimizers-403baeed9922>
- Keerthanams. (2025, May 8). *Evaluating AI Models — Understanding Entropy, Perplexity, BPB, and BPC*. Medium. <https://medium.com/@keerthanams1208/evaluating-ai-models-understanding-entropy-perplexity-bpb-and-bpc-df816062f21a>
- LMPO. (2020). *From text to tokens: Understanding BPE, WordPiece and SentencePiece in NLP*. Medium. <https://medium.com/@lmpo/from-text-to-tokens-understanding-bpe-wordpiece-and-sentencepiece-in-nlp-1367d9d610af>
- Neuraforge. (2023). *The Ultimate Guide to Preparing Text Data (Embeddings)*. Substack. <https://neuraforge.substack.com/p/the-ultimate-guide-to-preparing-text>
- Paul, R. (2025, Mar 2). *Balancing Vocabulary Size in Modern LLMs (GPT-4, LLaMA, Mistral)*. Rohan Paul Blog. <https://www.rohan-paul.com/p/tutorial-balancing-vocabulary-size>
- Pérez, J. A. (2020). *Embeddings en Transformers*. Universidad de Alicante. <https://www.dlsi.ua.es/~japerez/materials/transformers/embeddings/>
- Phillips, H. J. (2019, agosto). *Positional encoding*. Medium. <https://medium.com/@hunter-j-phillips/positional-encoding-7a93db4109e6>
- Pillutla, K., & colaboradores. (2025). mauve: Package to compute MAUVE, a similarity score between neural text and human text [GitHub repository].
- Plain English AI. (2021). *The 8 mathematical operations in GPT that accidentally created consciousness*. Medium. <https://ai.plainenglish.io/the-8-mathematical-operations-in-gpt-that-accidentally-created-consciousness-ccaee58b241e>
- Popovic, M. (2023). *nn.Linear in PyTorch: Clearly Explained*. Kanaries Docs. <https://docs.kanaries.net/topics/Python/nn-linear>
- PyTorch. (2025a). *CosineAnnealingLR — torch.optim.lr_scheduler*. PyTorch Documentation. https://docs.pytorch.org/docs/stable/generated/torch.optim.lr_scheduler.CosineAnnealingLR.html
- PyTorch. (2025b). *How to adjust learning rate — torch.optim*. PyTorch Documentation. <https://pytorch.org/docs/stable/optim.html%5C#how-to-adjust-learning-rate>
- Reddit. (2021). *SentencePiece, WordPiece, BPE: which tokenizer is best?* Reddit r/MachineLearning. https://www.reddit.com/r/MachineLearning/comments/rprm3/d_sentencepiece_wordpiece_bpe_which_tokenizer_is/?tl=es-es
- Reddit users. (2023). *Why isn't my Transformer model learning? (thread)*. Reddit r/MachineTranslation. https://www.reddit.com/r/machinetranslation/comments/1dmjbrj/why_isnt_my_transformer_model_learning/
- Santraigi, B. (2025). *Batched top-k and top-p / nucleus sampling in PyTorch [Gist con implementación en PyTorch de top-k y top-p filtrado]*. <https://gist.github.com/bsantraigi/5752667525d88d375207f099bd78818b>

- Shabbir, S. (2021). *Text Cleaning in NLP: Libraries, Techniques and Getting Started*. Medium. https://medium.com/@datascientist_SheezaShabbir/text-cleaning-in-nlp-libraries-techniques-and-how-to-get-started-8c7c7e8ba7cf
- Sharma, M. (2024). *What is Add & Norm, as quick as possible?* Medium. <https://molgorithm.medium.com/what-is-add-norm-as-soon-as-possible-178fc0836381>
- Singh, D. (2024, noviembre). *Why Perplexity Matters: A Deep Dive into NLP's Fluency Metric*. Medium. <https://medium.com/ai-enthusiast/why-perplexity-matters-a-deep-dive-into-nlps-fluency-metric-c3872811d598>
- StackOverflow users. (2019). *Calculate perplexity in PyTorch*. StackOverflow. <https://stackoverflow.com/questions/59209086/calculate-perplexity-in-pytorch>
- StackOverflow users. (2022a). *Is GradScaler necessary with Mixed Precision training with PyTorch?* StackOverflow. <https://stackoverflow.com/questions/72534859/is-gradscaler-necessary-with-mixed-precision-training-with-pytorch>
- StackOverflow users. (2022b). *Is SGD optimizer in PyTorch actually does Gradient Descent algorithm?* [Answer explaining the relation between SGD, mini-batch, and full gradient descent in PyTorch]. StackOverflow. <https://stackoverflow.com/questions/72496224/is-sgd-optimizer-in-pytorch-actually-does-gradient-descent-algorithm>
- Swarms. (2021). *Understanding Masking in PyTorch for Attention Mechanisms*. Medium. <https://medium.com/@swarms/understanding-masking-in-pytorch-for-attention-mechanisms-e725059fd49f>
- Ultralytics. (2025). *Optimizador Adam – Glosario Ultralytics*. Ultralytics. <https://www.ultralytics.com/es/glossary/adam-optimizer>
- Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention Is All You Need. *Advances in Neural Information Processing Systems*, 5998-6008. https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf
- Vishnu Raj, M. (2025, Jul 27). *Mathematics Behind the Temperature in LLM*. Medium. <https://medium.com/@madasuvishnuraj/mathematics-behind-the-temperature-in-llm-cfb23120ac62>
- Vyas, P. (2025-04-16). *Transformer: Decoder Only Generative Model (Part-3)*. Medium. https://medium.com/@pratik.vyas%5C_10544/transformer-decoder-only-generative-model-part-3-0b38c10c2ae9
- Wolfe, C. R. (2025). *Scaling Laws for LLMs: From GPT-3 to o3*. Substack. <https://cameronrwolfe.substack.com/p/llm-scaling-laws>
- Yassin, A. (2024, Nov 8). *Adam vs. AdamW: Understanding Weight Decay and Its Impact on Model Performance*. Medium. <https://yassin01.medium.com/adam-vs-adamw-understanding-weight-decay-and-its-impact-on-model-performance-b7414f0af8a1>

Apéndice A

Anexos

A.1. Preparación del entorno

El primer paso es crear el entorno virtual para trabajar con las librerías de Python:

```
python3 -m venv .venv
source .venv/bin/activate
```

El segundo punto consiste en instalar CUDA, Pytorch y Tensorflow y verificar que la GPU es detectada

```
# Verificar CUDA, en mi caso tengo la version 12.9.
nvidia-smi
```

```
#Instalar torch para esa version de CUDA
```

```
pip3 install --pre torch torchvision torchaudio --index-url https://download.pyt
```

```
#Instalar tensorflow
```

```
pip3 install "tensorflow[and-cuda]"
```

```
import torch
print(torch.cuda.is_available())
print(torch.version.cuda)
print(torch.cuda.device_count())
```

Finalmente, se instalan el resto de librerías necesarias con el comando pip3.

A.2. Moses

A.3. GELU

A.4. Xavier Uniform

A.5. Normal uniform

A.6. Entropía

A.7. MAUVE

A.8. Distinct-n

A.9. Resultados obtenidos para Tiny Shakespeare

A continuación se recogen los resultados obtenidos al validar el modelo entrenado con Tiny Shakespeare, partiendo del mismo prompt inicial: «*KING RICHARD*:» y un máximo de 100 tokens a generar.

Tabla A.1: Hiperparámetros y resultados de generación con Tiny Shakespeare

Temp	Top-k	Top-p	Presence Penalty	Frequency Penalty	Resultado
0.5	None	None	0.0	0.0	KING RICHARD: PETRUCHIO: My brother, a thousand crowns, And by the sun of my lord, Iney's the king, And make the king's wife, And from the king, You are sent for your brother. In the day to be not, I am not say 'tis blood, With all the right, and to be, And in the wind of his, Purder to the.

Tabla A.1 – continuación

Temp	Top-k	Top-p	Presence Penalty	Frequency Penalty	Resultado
1.0	None	None	0.0	0.0	<p>KING RICHARD: A issue richer? BRUTUS: ired, poor high a Thou art thou that a better man, girl? 'tis nothing but a squorn's wife, we'll thunder, and Come news? Theirrah! Less'd this outrage, Kate, sweet sir, the fished hat in estimation, Ofigeia There is no other tribute yet What Eished</p>
1.5	None	None	0.0	0.0	<p>KING RICHARD: Never charged thee other mistress repealll prove LOUCESTER: For your own thirst, father conclusion dead my s And, shout: like from his mortal a piece Peter, For night and leave report; Pompey counsels amid treaty in myself lovely an A below me confines nor doth depow tell thee le Imagine pates encothing</p>
1.0	10	None	0.0	0.0	<p>KING RICHARD: Is to be Madam with her: To see your honour, And I know not be a word. That you As my lord. On if we must be gone, to do not a man, I am no MENENIUS: That I will I am I On this land's name, I have I When thou dost thou art a man of you, my father</p>

Tabla A.1 – continuación

Temp	Top-k	Top-p	Presence Penalty	Frequency Penalty	Resultado
1.0	90	None	0.0	0.0	<p>KING RICHARD: Having the cause on'd the state: KING RICHARD II: 'em of our enemies? Their, do not to the same Yea, then 'such; ISABELLA: With bag, PETRUCHIO: SICINIUS: Could be your crown: CORIOLANUS: Fetch thy death hath been accused, my state and I's children should the world to live and good nu</p>
1.0	200	None	0.0	0.0	<p>KING RICHARD: But now must be the law, Thou hast you: Nay not be he were not? Or I's mine enemy in good lady't not in one and Whose repetition: Unations, sir, no So you: I will have no harm is my father's depart As, well; and they joy YORK: I know to his fortunes of the time so I pray</p>
1.0	None	0.05	0.0	0.0	<p>KING RICHARD: And, and I have And I have been And I have been And I have been And I have been And I have been And, I have been And I have been And I have been And I have been And I'd, I have been a man, I have been And I am a man, I have been And I have you, I have done,</p>

Tabla A.1 – continuación

Temp	Top-k	Top-p	Presence Penalty	Frequency Penalty	Resultado
1.0	None	0.30	0.0	0.0	<p>KING RICHARD: If I have you to the way. Which I am I am the poor soul, And all the Capitol, and, Within, sir, And then I will not? Why, the loss of the world'd, To hear me. Which he'd in the best of this, I'd with the field And all the king'd, and so far off. If you</p>
0.9	100	0.95	0.3	0.2	<p>KING RICHARD: And watch's his lands, Within so? With heigh! First Citizen: Against the word, I send you have met to take or But it down. What if I would not so bright too, and Romeo sh To more proud that I'd for ever near with The day! We must be king, here comes too long I thank</p>
0.7	80	0.9	0.2	0.3	<p>KING RICHARD: And make me, sir. So I am I not to be hanged. If what, for our mistress'd at the queen, Nurse: CAMILLO: With all my lord; for his son: Or do you all my daughter is the king's death? And yet thou dost thou art not so; or two, and not rest, FRIAR LAURENCE: H</p>

