UNIVERSIDAD SAN FRANCISCO DE QUITO USFQ

Colegio de Ciencias e Ingenierías

Estructuras Matemáticas en Modelos de Lenguaje

Andrés Nicolás Carrión García

Matemáticas

Trabajo de fin de carrera presentado como requisito

para la obtención del título de

Matemático

Quito, 11 de diciembre de 2024

UNIVERSIDAD SAN FRANCISCO DE QUITO USFQ

Colegio de Ciencias e Ingenierías

HOJA DE CALIFICACIÓN

DE TRABAJO DE FIN DE CARRERA

Estructuras Matemáticas en Modelos de Lenguaje

Andrés Nicolás Carrión García

Julio César Ibarra Fiallo, Ph.D. (c) .....

Antonio Di Teodoro, Ph.D. .....

Quito, 11 de diciembre de 2024

© DERECHOS DE AUTOR

Por medio del presente documento certifico que he leído todas las Políticas y Manuales de la Universidad San Francisco de Quito USFQ, incluyendo la Política de Propiedad Intelectual USFQ, y estoy de acuerdo con su contenido, por lo que los derechos de propiedad intelectual del presente trabajo quedan sujetos a lo dispuesto en esas Políticas.

Asimismo, autorizo a la USFQ para que realice la digitalización y publicación de este trabajo en el repositorio virtual, de conformidad a lo dispuesto en la Ley Orgánica de Educación Superior del Ecuador.

Nombres y apellidos:	Andrés Nicolás Carrión García
Código:	00320261
C.I.:	1727247692
Fecha:	Quito, 11 de diciembre de 2024

ACLARACIÓN PARA PUBLICACIÓN

Nota: El presente trabajo, en su totalidad o cualquiera de sus partes, no debe ser considerado como una publicación, incluso a pesar de estar disponible sin restricciones a través de un repositorio institucional. Esta declaración se alinea con las prácticas y recomendaciones presentadas por el Committee on Publication Ethics COPE descritas por Barbour et al. (2017) Discussion document on best practice for issues around theses publishing, disponible en http://bit.ly/COPETheses

UNPUBLISHED DOCUMENT

Note: The following capstone project is available through Universidad San Francisco de Quito USFQ institutional repository. Nonetheless, this project – in whole or in part – should not be considered a publication. This statement follows the recommendations presented by the Committee on Publication Ethics COPE described by Barbour et al. (2017) Discussion document on best practice for issues around theses publishing available on http://bit.ly/COPETheses

RESUMEN

Este trabajo presenta un análisis detallado del modelo transformer decoder-only. A lo largo del documento, se desglosan los componentes clave del modelo, como los embeddings, la tokenización, la codificación posicional y el mecanismo de atención, desde una perspectiva matemática. Se explica cómo cada uno de estos elementos permite al modelo interpretar y generar lenguaje natural de manera eficiente, preservando tanto la semántica como la estructura del texto.

Además del análisis teórico, se incluye una implementación práctica que consiste en el ajuste fino de un modelo preentrenado para generación de texto relacionado al manual del estudiante de la USFQ. Los resultados obtenidos evidencian la eficacia del modelo en la generación de texto coherente, subrayando la importancia de los transformers en el procesamiento del lenguaje natural y su impacto en diversas áreas. Este estudio contribuye a una mejor comprensión de los fundamentos matemáticos que sustentan el funcionamiento de los transformers, ofreciendo una base sólida para futuros desarrollos y aplicaciones en el ámbito académico y profesional.

***Palabras clave*:** Transformer, ChatGPT, Procesamiento del Lenguaje Natural (NLP), Modelos Generativos, Embeddings, Tokenización, Codificación Posicional, Mecanismo de Atención, Modelos Extensos de Lenguaje (LLM)

ABSTRACT

This document presents a detailed analysis of the decoder-only transformer model. Throughout the paper, the key components of the model, such as embeddings, tokenization, positional encoding and the attention mechanism, are broken down from a mathematical perspective. It explains how each of these elements allows the model to interpret and generate natural language efficiently, preserving both semantics and text structure.

In addition to the theoretical analysis, a practical implementation is included, consisting in the fine tuning of a pre-trained model for text generation related to the USFQ student handbook. The results obtained show the effectiveness of the model in generating coherent text, highlighting the importance of transformers in natural language processing and their impact in several areas. This study contributes to a better understanding of the mathematical foundations that support the operation of transformers, providing a solid basis for future developments and applications in the academic and professional fields.

***Keywords*:** Transformer, ChatGPT, Natural Language Processing (NLP), Generative Models, Embeddings, Tokenization, Positional Encoding, Attention Mechanism, Large Language Models (LLM)

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

ÍNDICE DE CUADROS

DEDICATORIA

A mi familia, cuyo amor, sacrificio y apoyo incondicional han sido la base de este logro. A mis padres, por su entrega y confianza; a mis abuelitos, por su sabiduría y ejemplo de fortaleza; y a mi tía, por estar siempre presente, brindándome su amor y motivación.

AGRADECIMIENTO

Quiero expresar mi más sincero agradecimiento a todos aquellos que han sido parte de este viaje. A mi familia, en especial a mis padres, abuelos y tía, por su cariño incondicional, su apoyo constante y por enseñarme el verdadero significado de la perseverancia. A mi novia, cuyo amor, comprensión y apoyo han sido fundamentales para superar las dificultades de este proceso.

A mis profesores, en especial John Skukalek, quien encendió en mí la pasión por las matemáticas, y a Julio Ibarra, mi tutor de tesis, por haberme introducido en el fascinante mundo de las redes neuronales e inteligencia artificial, lo cual ha sido clave para mi futuro profesional. Gracias por su orientación, paciencia y confianza.

Finalmente, a mis amigos, por su apoyo incondicional, por estar siempre a mi lado y recordarme la importancia de seguir adelante, sin importar las circunstancias.

CAPÍTULO 1

INTRODUCCIÓN

Los avances en el procesamiento del lenguaje natural (NLP) han revolucionado la forma en que interactuamos con la información y las máquinas. Entre estos avances, los modelos de lenguaje basados en arquitecturas transformer han demostrado ser altamente efectivos para tareas que van desde la traducción automática hasta la generación de texto coherente y contextualizado. En particular, los modelos "decoder-only", como ChatGPT o LLaMA, han destacado por su capacidad para generar respuestas precisas y relevantes en una amplia variedad de contextos.

Este trabajo se centra en el estudio detallado del funcionamiento interno de los modelos transformer decoder-only, desglosando sus componentes desde una perspectiva matemática. El análisis incluye el proceso de tokenización, la representación semántica mediante embeddings, la incorporación de información posicional y el mecanismo de atención. Cada uno de estos elementos es fundamental para que el modelo pueda interpretar y generar lenguaje de manera eficiente.

Además, el presente trabajo no solo aborda los aspectos teóricos, sino que también incluye una implementación práctica que consiste en el ajuste fino de un modelo preentrenado para un contexto específico. En este caso, se ha adaptado el modelo para generar texto relacionado con el manual del estudiante de la Universidad San Francisco de Quito, demostrando su aplicabilidad en tareas reales y su capacidad para ser personalizado según necesidades particulares.

A lo largo del documento, se busca no solo explicar el funcionamiento del modelo, sino también resaltar la importancia de cada uno de sus componentes desde un enfoque matemático, permitiendo al lector comprender cómo estas estructuras contribuyen al éxito del modelo en

la generación de texto coherente y contextualizado. Con ello, se pretende ofrecer una visión integral que abarque tanto la teoría como la práctica, subrayando el impacto y el potencial de los modelos de lenguaje en diversos ámbitos académicos y profesionales.

CAPÍTULO 2

DEFINICIONES Y MÉTODOS

En este capítulo se proporcionan definiciones matemáticas clave y conceptos fundamentales necesarios para entender los mecanismos matemáticos que subyacen a los modelos basados en Transformers. Cabe recalcar que, a lo largo de este trabajo, se adopta la convención de indexar los elementos desde 0, en lugar de desde 1. Esto es común en la mayoría de los lenguajes de programación y facilita el manejo de matrices y vectores en diversas implementaciones algorítmicas, como es el caso en los Transformers, donde las secuencias y matrices de atención se gestionan de esta forma. Dicho esto, a continuación se detallan las definiciones y notaciones utilizadas a lo largo de este trabajo.

2.1. Fundamentos de Álgebra Lineal

Primero se recordarán algunas definiciones de álgebra lineal, las cuales permitirán comprender los fundamentos matemáticos necesarios para abordar los modelos basados en Transformers.

Definición 2.1.1. Un **espacio vectorial (real)** es un conjunto V junto con una operación de adición sobre V y una operación de multiplicación por escalares sobre V, de manera que se cumplan las siguientes propiedades:

- **Conmutatividad:** $u + v = v + u$ para todo $u, v \in V$.
- **Asociatividad:** $(u + v) + w = u + (v + w)$ y $(ab)v = a(bv)$ para todo $u, v, w \in V$ y para todo $a, b \in \mathbb{R}$.
- **Elemento neutro aditivo:** Existe un elemento $0 \in V$ tal que $v + 0 = v$ para todo $v \in V$.
- **Inverso aditivo:** Para todo $v \in V$, existe un $w \in V$ tal que $v + w = 0$.
- **Identidad multiplicativa:** $1v = v$ para todo $v \in V$.

- **Propiedades distributivas:** $a(u+v) = au + av$ y $(a+b)v = av + bv$ para todo $a, b \in \mathbb{R}$ y para todo $u, v \in V$.

Los elementos de un espacio vectorial se denominan **vectores**.

Definición 2.1.2. Un **subespacio vectorial** es un subconjunto U de un espacio vectorial V tal que U también es un espacio vectorial bajo la misma operación de adición de V y la misma operación de multiplicación por escalares de V. Para que sea un subespacio se deben satisfacer las siguientes propiedades:

- **Identidad aditiva:** $0 \in U$.
- **Cerrado bajo adición:** $u, w \in U$ implica que $u + w \in U$.
- **Cerrado bajo multiplicación por escalares:** $a \in \mathbb{R}$ y $v \in U$ implican que $au \in U$.

Definición 2.1.3. Una **combinación lineal** de una lista de vectores $v_0, \ldots, v_n \in V$ es un vector de la forma

$$a_1 v_1 + \cdots + a_m v_m,$$

donde $a_1, \ldots, a_m \in \mathbb{R}$.

Definición 2.1.4. El conjunto de todas las combinaciones lineales de una lista de vectores $v_0, \ldots, v_m$ en V se llama el **span** de $v_0, \ldots, v_m$, denotado $\operatorname{span}\{v_0, \ldots, v_m\}$. En otras palabras,

$$\operatorname{span}\{v_0, \ldots, v_m\} = \{a_0 v_0 + \cdots + a_m v_m \mid a_0, \ldots, a_m \in \mathbb{R}\}.$$

El span de la lista vacía $()$ se define como $\{0\}$.

Es importante notar que el span de una lista de vectores es el subespacio más pequeño conteniendo esos vectores.

Definición 2.1.5. Una lista de vectores $v_0, \ldots, v_m$ en V se dice que es **linealmente independiente** si,

$$a_0 v_0 + \cdots + a_m v_m = 0$$

si y solo si $a_0 = a_1 = \cdots = a_m = 0$. Caso contrario se dice que es **linealmente dependiente**.

Definición 2.1.6. Una **base** de V es una lista de vectores en V que es linealmente independiente y genera V (es decir, el span de los vectores de la lista es igual a V).

Definición 2.1.7. La lista de vectores

$$\begin{aligned} e_0 &= (1, 0, 0, \ldots, 0, 0), \\ e_1 &= (0, 1, 0, \ldots, 0, 0), \\ &\vdots \\ e_{n-1} &= (0, 0, 0, \ldots, 0, 1) \end{aligned}$$

es una base de $\mathbb{R}^n$, llamada la **base estándar** o **base canónica** de $\mathbb{R}^n$. En otras palabras:

$$e_i = (a_0, a_1, \ldots, a_{n-1}), \quad a_j = \begin{cases} 1, & j = i \\ 0, & j \neq i \end{cases}$$

Definición 2.1.8. La **dimensión** de un espacio vectorial de dimensión finita es la longitud de cualquier base del espacio vectorial. La dimensión de V (si V es de dimensión finita) se denota por $\dim V$. Por ejemplo, la dimensión de $\mathbb{R}^n$ es n debido a la definición 2.1.7.

Definición 2.1.9. Sean $U_1, U_2, \ldots, U_m$ subconjuntos de V. La **suma** de $U_1, \ldots, U_m$, denotada como $U_1 + \cdots + U_m$, es el conjunto de todas las posibles sumas de elementos de $U_1, \ldots, U_m$. Más precisamente,

$$U_1 + \cdots + U_m = \{u_1 + \cdots + u_m \mid u_1 \in U_1, \ldots, u_m \in U_m\}.$$

Si $U_1, U_2, \ldots, U_m$ son subespacios, entonces $U_1 + \cdots + U_m$ es el subespacio más pequeño de V que los contiene.

Definición 2.1.10. Sean $U_1, \ldots, U_m$ subespacios de V. La suma $U_1 + \cdots + U_m$ se llama **suma directa** si cada elemento de $U_1 + \cdots + U_m$ puede escribirse de manera única como una suma $u_1 + \cdots + u_m$, donde cada $u_j \in U_j$.

Si $U_1 + \cdots + U_m$ es una suma directa, entonces se denota como

$$U_1 \oplus \cdots \oplus U_m$$

para indicar que es una suma directa, con el símbolo $\oplus$ sirviendo como indicación de que esta es una suma directa.

Definición 2.1.11. Una **transformacion lineal** de V a W (espacios vectoriales) es una función $T : V \to W$ que satisface las siguientes propiedades:

- **Aditividad:** $T(u + v) = T(u) + T(v)$ para todos $u, v \in V$.
- **Homogeneidad:** $T(\lambda v) = \lambda T(v)$ para todo $\lambda \in \mathbb{F}$ y $v \in V$.

Denotamos al conjunto de transformaciones lineales de V a W como $\mathcal{L}(V, W)$.

Definición 2.1.12. Una transformación lineal de un espacio vectorial a sí mismo se llama un **operador**. La notación $\mathcal{L}(V)$ denota el conjunto de todos los operadores sobre V. En otras palabras, $\mathcal{L}(V) = \mathcal{L}(V, V)$.

Definición 2.1.13. Supongamos que $T \in \mathcal{L}(V, U)$ para espacios vectoriales V, U y $v_0, \ldots, v_{n-1}$ es una base de V y $u_0, \ldots, u_{m-1}$ es una base de U. La **matriz de T respecto a estas bases** es la matriz $m \times n$ W_T cuyas entradas $w_{j,k}$ están definidas por

$$T(v_k) = w_{1,k}u_1 + \cdots + w_{m,k}u_m \quad \text{para cada} \quad k = 1, \ldots, n.$$

Si una matriz $W \in \mathbb{R}^{m \times n}$ denota una transformación lineal en $\mathcal{L}(V, U)$ para espacios vectoriales reales V, U de dimensiones n y m respectivamente, entonces decimos que $W \in \mathcal{L}(V, U)$.

Definición 2.1.14. Supongamos que A es una matriz de $m \times n$.

- Si $1 \leq j \leq m$, entonces $A_{j,\cdot}$ denota la matriz de $1 \times n$ que consiste en la fila j de A.
- Si $1 \leq k \leq n$, entonces $A_{\cdot,k}$ denota la matriz de $m \times 1$ que consiste en la columna k de A.

Definición 2.1.15. Para $x, y \in \mathbb{R}^n$, el **producto escalar** de x y y, denotado por $x \cdot y$, se define como

$$x \cdot y = x_0 y_0 + \cdots + x_{n-1} y_{n-1},$$

donde $x = (x_0, \ldots, x_{n-1})$ y $y = (y_0, \ldots, y_{n-1})$.

Definición 2.1.16. Denotamos $[N]$ al conjunto de números enteros no negativos menores a N, es decir,

$$[N] = \{n \in \mathbb{Z} : 0 \leq n < N\} = \{0, 1, \ldots, N-1\}$$

2.2. Funciones Matemáticas y Métricas en Modelos de Aprendizaje Automático

Tras haber establecido las definiciones fundamentales de álgebra lineal, es necesario introducir una serie de funciones y métricas que juegan un papel crucial en el entrenamiento y la optimización de modelos. Para ello recordemos lo que es una capa oculta de una red neuronal pero desde un punto de vista matemático.

Definición 2.2.1. Una **capa oculta de una red neuronal** de dimensión k con entrada de dimensión d_{in} es un mapa $f : \mathbb{R}^{d_{in}} \to \mathbb{R}^k$ que consiste en la conjugación de una transformación lineal $W^T \in \mathcal{L}(\mathbb{R}^{d_{in}}, \mathbb{R}^k)$ (W^T denota W transpuesta) y un sesgo $b \in \mathbb{R}^k$ con una función (generalmente no lineal) $\theta : \mathbb{R}^k \to \mathbb{R}^k$, es decir,

$$f(x) = \theta(W^T \mathbf{x} + b).$$

A θ se le denomina **función de activación**. Por otro lado, para procesar una entrada de longitud

τ a la vez, entonces obtenemos

$$f_\tau(x) = \theta_\tau(XW + B),$$

donde $X \in \mathbb{R}^{\tau, d_{in}}$ y para $j \in [k]$, $B_{j,\cdot} = b$ y $\theta_\tau(A)_{j,\cdot} = \theta(A_{j,\cdot})$.

Para abordar la clasificación en modelos de aprendizaje automático así como la obtención de ponderaciones para una media ponderada, una de las funciones más comunes y esenciales es la función **Softmax**. Esta función es particularmente útil ya que convierte un vector de valores reales en un vector de probabilidades, asignando una ponderación o probabilidad a cada componente de modo que la suma de todas las probabilidades sea 1, es decir, se crea una distribución de probabilidad.

Definición 2.2.2. La función **Softmax** es una función matemática que toma un vector $z \in \mathbb{R}^k$ y lo convierte en un vector de probabilidades $p \in \mathbb{R}^k$, tal que los componentes de p forman una distribución de probabilidad. Dada una entrada $z = (z_0, z_2, \ldots, z_{k-1})$, la función Softmax se define como

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=0}^{k-1} e^{z_j}} \quad \text{para} \quad i \in [k].$$

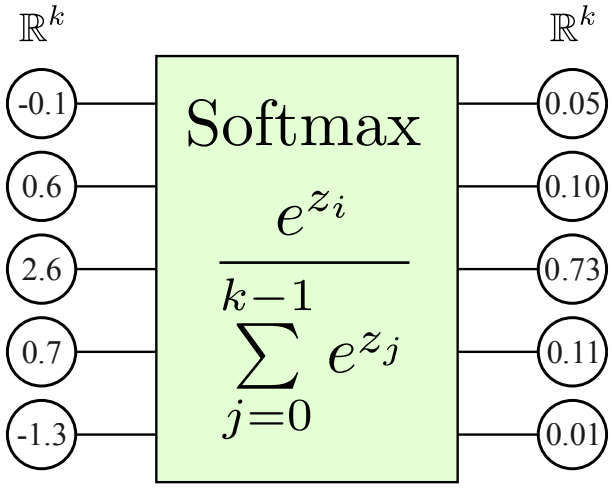

Figura 2.1: Softmax aplicado a un vector en $\mathbb{R}^k$.

Continuando con la descripción de funciones utilizadas en redes neuronales, es importante mencionar unas de las funciones de activación ampliamente utilizadas. La función **ReLU** es una de las funciones de activación más simples y efectivas, especialmente en redes neuronales profundas. Es ampliamente utilizada debido a su eficiencia computacional y a que puede ayudar a mitigar el problema del desvanecimiento del gradiente.

Definición 2.2.3. La función **ReLU** (Rectified Linear Unit) es una función no lineal que se define punto a punto como

$$\text{ReLU}(x) = \text{máx}(0, x),$$

para cada $x \in \mathbb{R}$. En otras palabras, la función devuelve el valor de entrada x si es positivo, y 0 si es negativo.

Por otro lado, **GELU** (Hendrycks & Gimpel, 2023) es una función de activación inspirada en ReLU pero definida de tal manera que es suave y continuamente diferenciable, contrario a ReLU que no es diferenciable en 0. Es una función de activación más reciente que ha mostrado buenos resultados en arquitecturas de redes neuronales modernas, como los modelos de lenguaje basados en transformers.

Definición 2.2.4. La función **GELU** (Gaussian Error Linear Unit) es una función de activación suave que combina los beneficios de una activación lineal con una modulación probabilística basada en la distribución normal. Se define como

$$\text{GELU}(x) = xP(X \leq x) = x\Phi(x),$$

donde $x \in \mathbb{R}$, $X \sim \mathcal{N}(0, 1)$ y Φ es la función de distribución acumulativa de la distribución normal estándar. También se suele usar la siguiente aproximación:

$$\text{GELU}(x) \approx 0.5x \left(1 + \tanh\left(\sqrt{\frac{2}{\pi}}\left(x + 0.044715x^3\right)\right)\right),$$

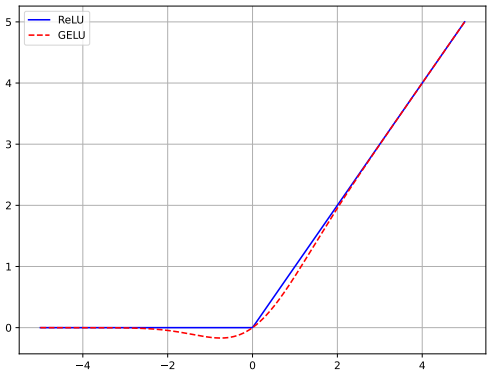

Figura 2.2: ReLU vs GELU.

La **normalización de capa** (Ba et al., 2016) es una técnica importante utilizada en redes neuronales profundas, especialmente en modelos de aprendizaje automático que involucran secuencias, como los modelos basados en transformers. La normalización ayuda a estabilizar el entrenamiento al reducir el problema de la covariancia interna, asegurando que las activaciones de cada capa tengan una media de 0 y una desviación estándar de 1, lo que facilita el proceso de aprendizaje y mejora la convergencia.

Definición 2.2.5. La **normalización de capa** (Layer Normalization) $\hat{h}$ para un vector de activaciones $h^{(t)} = (h_0^{(t)}, h_1^{(t)}, \ldots, h_{k-1}^{(t)}) \in \mathbb{R}^k$ se define como

$$\hat{h}_i^{(t)} = \frac{h_i^{(t)} - \mu_{h^{(t)}}}{\sigma_{h^{(t)}}} \cdot \gamma_i + \beta_i \quad \text{para } i \in [k],$$

donde:

$$\mu_{h^{(t)}} = \frac{1}{k}\sum_{i=1}^{k} h_i, \quad \sigma_{h^{(t)}} = \sqrt{\frac{1}{k}\sum_{i=1}^{k}(h_i^{(t)} - \mu_{\mathbf{h}})^2},$$

y γ y β son vectores en $\mathbb{R}^k$ que son parámetros aprendibles. También se lo puede denotar de la siguiente forma:

$$\text{LayerNorm}(h) = \frac{h - \mu_h}{\sigma_h} \odot \gamma + \beta,$$

donde $\odot$ denota el producto elemento a elemento.

La **entropía cruzada** (Goodfellow et al., 2016) es una medida para determinar qué tan distintas son dos distribuciones de probabilidad. En el caso de aprendizaje automático, sirve para problemas de clasificación donde se debe medir la diferencia entre una salida con una distribución de probabilidad y entre la distribución de probabilidad verdadera de las clases.

Definición 2.2.6. La **Entropía Cruzada** (Cross-entropy) $H(P, Q)$ entre dos distribuciones de probabilidad P y Q se define como

$$H(P, Q) = -\mathbb{E}_{x \sim P} \log Q(x).$$

En el caso de un modelo de aprendizaje automático que busca clasificar, lo que devolverá serán las probabilidades para cada clase c_i dada una entrada $(x_0, \ldots, x_n)$, es decir, $p(c_i|x_0, \ldots, x_n)$. En este caso, la distribución de probabilidad sería

$$p(c_i|x_0, \ldots, x_n) = \begin{cases} 1, & c_i \text{ es la clase real } (c_{\text{real}}) \\ 0, & \text{caso contrario} \end{cases}.$$

Por lo tanto, en este caso se busca maximizar la ecuación

$$-\sum p(c_{\text{real}}|x_0, \ldots, x_n).$$

CAPÍTULO 3

EXPLICACIÓN MATEMÁTICA DE UN TRANSFORMER

El Transformer revolucionó el procesamiento del lenguaje natural (NLP) al introducir mecanismos de atención que optimizan tareas como la generación de texto. Su arquitectura se basa en mecanismos de atención que permiten modelar secuencias de manera eficiente y efectiva. Esta sección se centra en el funcionamiento de un modelo de solo decodificador, comúnmente utilizado en la generación de texto.

Un modelo de solo decodificador (o "decoder-only"), como el que se utiliza en sistemas de generación de texto, está diseñado para producir secuencias de salida basándose en entradas anteriores y en un contexto generado por el propio modelo. La arquitectura consta de múltiples capas apiladas que incluyen mecanismos de atención y redes neuronales MLP. Estas capas se intercalan con mecanismos de normalización y conexiones residuales que garantizan la estabilidad y el aprendizaje eficiente. En la imagen 3.1 se puede apreciar la arquitectura de un modelo de solo decodificador.

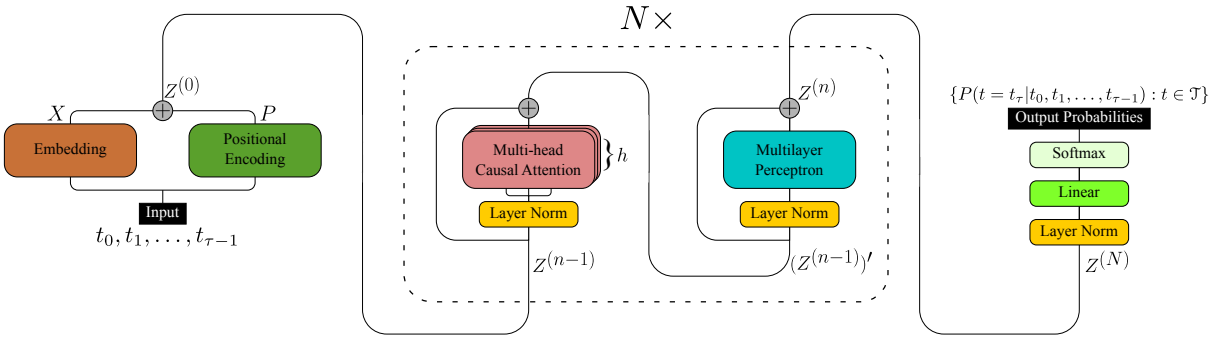

Figura 3.1: Arquitectura del Transformer tipo decoder-only.

3.1. Embedding

Cuando trabajamos con texto en modelos de aprendizaje profundo, surge la necesidad de convertir las palabras o secuencias de caracteres en representaciones numéricas que las

redes neuronales puedan procesar, esto debido a que las palabras en su forma natural no son interpretables directamente por una red neuronal. Podríamos pensar que simplemente asignar un número a cada palabra sería suficiente (por ejemplo, el número natural correspondiente al índice de esa palabra dentro de la lista de todo el vocabulario que maneja el modelo), pero esto no es adecuado porque no captura la relación semántica entre las palabras. Si asignamos números como 1 para *gato*, 2 para *perro* y 3 para *puerta*, el modelo no tendría forma de entender que estas palabras están relacionadas en el lenguaje natural; además, esto implica incorrectamente que relaciones como $2 \cdot x_{\text{gato}} = x_{\text{perro}}$ o $1.5 \cdot x_{\text{perro}} = x_{\text{puerta}}$ tienen sentido semántico, lo cual es absurdo en el lenguaje natural. Las palabras no pueden ser linealmente independientes entre sí (ser múltiplos una de otra), lo cual automáticamente descarta la posibilidad de usar cualquier representación unidimensional como números reales.

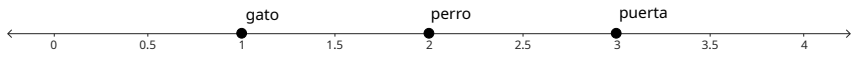

Figura 3.2: Representación de palabras en una sola dimensión.

Otro enfoque inicial es el **one-hot encoding**, donde cada palabra se representa como un vector elemento de la base canónica (2.1.7) de $\mathcal{V}$, donde $\mathcal{V}$ es el conjunto de palabras w en el vocabulario que se está considerando para el modelo. Debido a que $\mathbf{V} = \dim \mathcal{V}$ es finito, podemos considerar un conjunto de índices $\mathcal{T} = [\mathbf{V}]$ que represente nuestro vocabulario, es decir, $\mathcal{V} = \{w_t : t \in \mathcal{T}\}$. Cada índice $t \in \mathcal{T}$ se representa a través de un vector $e_t \in \mathbb{R}^{\mathbf{V}}$. Así, el vector e_t es la representación one-hot del token t, y se asocia de manera única a la palabra w_t en el vocabulario. Un ejemplo simple en el que el one-hot encoding es útil es en la clasificación de colores. Si tenemos tres colores: rojo, verde y azul, podemos representarlos de la siguiente manera con one-hot encoding:

- Rojo: $e_0 = (1, 0, 0) \in \mathbb{R}^3$
- Verde: $e_1 = (0, 1, 0) \in \mathbb{R}^3$

- Azul: $e_2 = (0, 0, 1) \in \mathbb{R}^3$

Este enfoque funciona bien porque las categorías son mutuamente excluyentes y no se requiere ninguna relación semántica entre ellas. Sin embargo, esto representa un problema al momento de trabajar con lenguaje. Aunque usar one-hot encoding garantiza unicidad en las representaciones, la ortogonalidad entre vectores impide que el modelo identifique relaciones semánticas, ya que todas las palabras están igualmente distantes unas de otras. Por ejemplo, los vectores que representan *gato* y *perro* estarían tan separados como *gato* y *puerta*, lo cual no refleja el comportamiento del lenguaje.

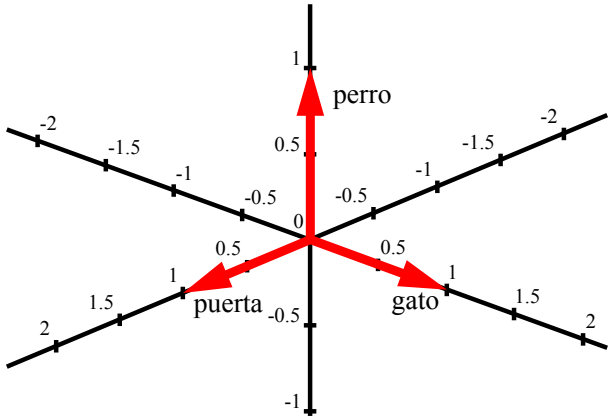

Figura 3.3: Representación de palabras con one-hot encoding.

Para superar estas limitaciones, se usan representaciones vectoriales denominadas **embeddings** que se encuentran en un espacio vectorial de dimensión menor al tamaño del vocabulario. Se escoge usar espacios vectoriales debido a que estos tienen estructura y es posible realizar operaciones algebraicas entre sus elementos. Así mismo, usar una dimensión más pequeña posibilita que las relaciones semánticas puedan capturarse a través de la proximidad en el espacio vectorial. Estos embeddings se pueden entender como representaciones distribuidas de palabras en un espacio vectorial, y ayudan a los algoritmos de aprendizaje a lograr un mejor rendimiento en tareas de procesamiento del lenguaje natural al agrupar palabras similares (Mikolov, Sutskever et al., 2013).

3.1.1. Word2Vec y Operaciones Algebraicas entre Palabras

Un conjunto de técnicas populares para entrenar embeddings pertenecen a la herramienta Word2Vec que surge del paper "Efficient Estimation of Word Representations in Vector Space" (Mikolov, Chen et al., 2013). En estas se utiliza redes neuronales para aprender representaciones vectoriales de palabras basadas en su contexto. Una de las arquitecturas principales de Word2Vec es la de **Skip-gram**, donde se predicen las palabras que rodean a una palabra central en un corpus de texto. Esto permite capturar las relaciones semánticas entre las palabras, basándose en su coocurrencia en diferentes contextos.

Una forma de entrenar este modelo es con una red neuronal con una capa oculta y sin activación. Las capas de entrada y salida son de dimensión $\mathbf{V}$, mientras que la capa oculta es de dimensión d_m que es la dimensión del embedding deseada; para esto es necesario tener todo un vocabulario $\mathcal{V}$ definido y que cada palabra tenga un índice único $t \in \mathcal{T}$. El input del modelo es una palabra w_{t_i}, donde i representa el índice de la palabra dentro del corpus de texto. Esta palabra w_{t_i} se representa como un vector one-hot e_{t_i} que activa únicamente la neurona correspondiente al índice de la palabra (t_i), y los pesos de esta primera capa son los embeddings que se ajustan durante el entrenamiento. En un corpus de texto escogemos una ventana o tamaño de contexto c para obtener una secuencia de palabras de tamaño $2c+1$ $(w_{t_{i-c}}, \ldots, w_{t_{i-1}}, w_{t_i}, w_{t_{i+1}}, \ldots, w_{t_{i+c}})$. Para entrenar el modelo y que aprenda la representación de la palabra w_{t_i}. Debido a que solo la neurona correspondiente al índice de la palabra se activa, solo sus d_m pesos se actualizarán para esta primera capa. Por otro lado, los targets del modelo serán todas las palabras que le rodean a w_{t_i}, es decir $w_{t_{i+j}}$ para $-c \leq j \leq c$, $j \neq 0$. Realizando ese procedimiento por todo el corpus de texto, se obtendrán representaciones vectoriales de dimensión d_m que conservan el contexto de las palabras, las cuales corresponden a los pesos de la capa de entrada de la red neuronal.

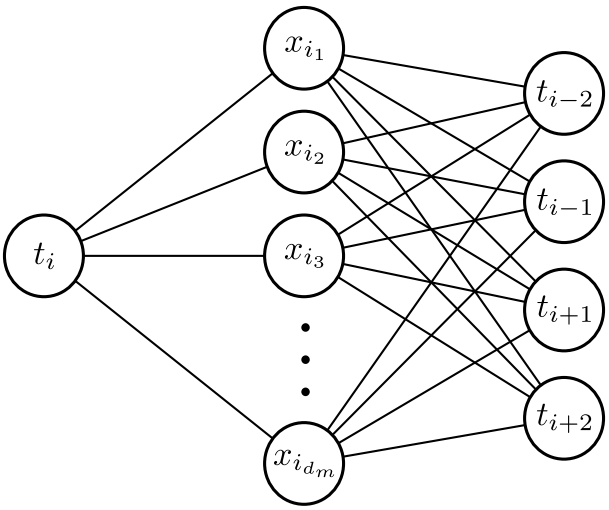

Figura 3.4: Arquitectura Modelo Word2Vec Skip-gram.

En la figura 3.4 se puede apreciar la neurona t_i que se activa en la entrada con la palabra w_{t_i} así como las únicas neuronas de salida a las que debería apuntar en esa iteración. Para las demás palabras del vocabulario debería apuntar a 0, con lo que, usando entropía cruzada (2.2.6), al actualizar los pesos se van a obtener representaciones vectoriales de palabras. En este caso, la entropía cruzada de solamente esa iteración (solo considerando w_{t_i}) viene dada por

$$-\sum_{\substack{-c\leq j\leq c\\ j\neq 0}} \log p(w_{t_{i+j}}|w_{t_i})$$

"Las representaciones de palabras calculadas usando redes neuronales son muy interesantes porque los vectores aprendidos codifican explícitamente muchas regularidades lingüísticas y patrones" (Mikolov, Sutskever et al., 2013, p. 1, *traducción propia*). Estas regularidades lingüísticas pueden ser modeladas como traslaciones lineales. Por ejemplo, se ha observado que:

$$x_{\text{Madrid}} - x_{\text{España}} + x_{\text{Francia}} \approx x_{\text{París}}$$

donde x_{palabra} es la representación vectorial de dicha palabra. Esto significa que la relación entre la representación de *España* con la de *Madrid* se aproxima a la de *Francia* con la de *París*

(Mikolov, Chen et al., 2013). En otras palabras, lo que se está conservando en este caso es una diferencia vectorial (delta) que refleja la relación entre un país y su capital. Este delta, denotado como $\Delta_{\text{capital}} = x_{\text{capital}} - x_{\text{país}}$, puede ser sumado a cualquier vector de características de un país para predecir su capital. De esta manera, si tenemos un vector x_{Francia} representando un país, entonces el vector correspondiente para su capital se puede aproximar como:

$$x_{\text{Francia}} + \Delta_{\text{capital}} \approx x_{\text{París}}$$

Esto implica que la relación semántica entre países y sus capitales se mantiene constante en el espacio vectorial, permitiendo que el modelo generalice la relación de forma efectiva entre otros pares país-capital. Este tipo de patrones revela cómo las relaciones semánticas pueden ser manipuladas algebraicamente en el espacio de los embeddings, permitiéndonos realizar álgebra lineal con palabras.

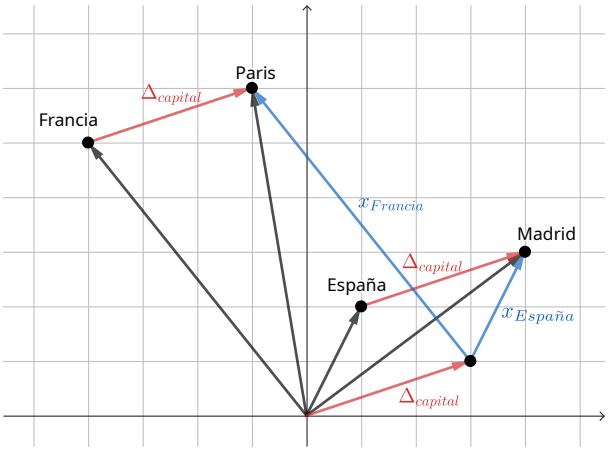

Figura 3.5: Operaciones vectoriales entre embeddings.

3.1.2. Optimización de Embeddings en Transformers

A diferencia de modelos como Word2Vec, que utilizan embeddings preentrenados, el Transformer entrena sus propios embeddings directamente dentro del modelo. Esto permite que las representaciones se optimicen conjuntamente con el resto de los parámetros, asegurando que capturen el contexto y la semántica del texto de manera efectiva para las tareas específicas. Los

embeddings en el Transformer no solo representan las palabras de forma aislada, sino que se adaptan dinámicamente a medida que el modelo entrena, lo que les otorga flexibilidad y riqueza contextual. Además, gracias a técnicas como la segmentación en subpalabras, el modelo puede manejar vocabularios extensos y generalizar mejor entre variaciones morfológicas, lo que mejora su capacidad para lidiar con palabras raras o desconocidas.

3.1.3. Tokenización

Para entrenar adecuadamente un modelo de lenguaje, es esencial definir un vocabulario $\mathcal{V}$ finito que el modelo pueda entender y represente el texto de forma eficiente. Sin embargo, una dificultad inherente es la gran cantidad de palabras y más aún cuando el modelo maneja múltiples idiomas. Entrenar un modelo con todas las palabras posibles sería ineficiente, ya que, incluso si se intentara incluir una gran cantidad de palabras, sería imposible garantizar que todas estén representadas. Además, si el vocabulario se construye a partir de un corpus que no contiene una palabra específica, el modelo no podría interpretarla, lo que generaría problemas cuando esa palabra aparezca en texto nuevo, produciendo errores en la predicción.

Una forma básica de abordar este problema es partir de caracteres en lugar de palabras. Esto tiene la ventaja de que el número de caracteres es finito, y cualquier palabra puede ser representada por una secuencia de caracteres. Por ejemplo, el conjunto de caracteres ASCII tiene 256 elementos (Injosoft AB, s.f.), lo cual es manejable desde un punto de vista computacional. Sin embargo, este enfoque tiene limitaciones importantes, ya que los caracteres, por sí mismos, no tienen semántica. Modelar secuencias de caracteres individuales no captura la estructura ni el significado de las palabras, lo que dificultaría que el modelo comprenda el contexto.

Para abordar las limitaciones de usar caracteres individuales, una técnica común es utilizar **subpalabras** o secuencias de caracteres que aparecen frecuentemente en el texto. Un método popular para identificar estas secuencias es **Byte Pair Encoding** (BPE) como lo describen Sennrich et al. (2016). El BPE comienza con un vocabulario que contiene caracteres individuales y luego combina los pares de caracteres que más se repiten en el texto. Estas combinaciones iterativas crean nuevas secuencias que corresponden a subpalabras, y eventualmente forman el

vocabulario final.

```
Texto: 'tokens en texto tokenizado'
Corpus: ['t', 'o', 'k', 'e', 'n', 's', ' ', 'e', 'n', ' ', 't', 'e', 'x', 't', 'o', ' ', 't', 'o', 'k', 'e', 'n', 'i', 'z', 'a', 'd', 'o']
Par más común: ['to']
New corpus: ['to', 'k', 'e', 'n', 's', ' ', 'e', 'n', ' ', 't', 'e', 'x', 'to', ' ', 'to', 'k', 'e', 'n', 'i', 'z', 'a', 'd', 'o']
Par más común: ['en']
New corpus: ['to', 'k', 'en', 's', ' ', 'en', ' ', 't', 'e', 'x', 'to', ' ', 'to', 'k', 'en', 'i', 'z', 'a', 'd', 'o']
Par más común: ['tok']
New corpus: ['tok', 'en', 's', ' ', 'en', ' ', 't', 'e', 'x', 'to', ' ', 'tok', 'en', 'i', 'z', 'a', 'd', 'o']
Par más común: ['token']
New corpus: ['token', 's', ' ', 'en', ' ', 't', 'e', 'x', 'to', ' ', 'token', 'i', 'z', 'a', 'd', 'o']
```

Figura 3.6: Ejemplo de BPE.

De esta forma, con un corpus de texto largo se aplica el método hasta obtener el tamaño de vocabulario $\mathbf{V}$ deseado. Así se obtiene el vocabulario $\mathcal{V}$ que es el conjunto de todos los tokens construidos mediante BPE.

Este enfoque permite que el modelo represente tanto palabras completas como fragmentos de palabras más pequeños que aún llevan semántica. Por ejemplo, si un modelo no conoce la palabra “automóvil”, podría dividirla en subpalabras como “auto” y “móvil”. Esto es útil porque, incluso si el modelo no ha visto la palabra completa durante el entrenamiento, puede manejarla al dividirla en fragmentos reconocibles. Para comprender mejor este proceso, podemos ver el ejemplo que nos ofrece la propia página de Open AI para observar como funciona la tokenización (OpenAI, s.f.), donde claramente tiene una tokenización óptima en inglés:

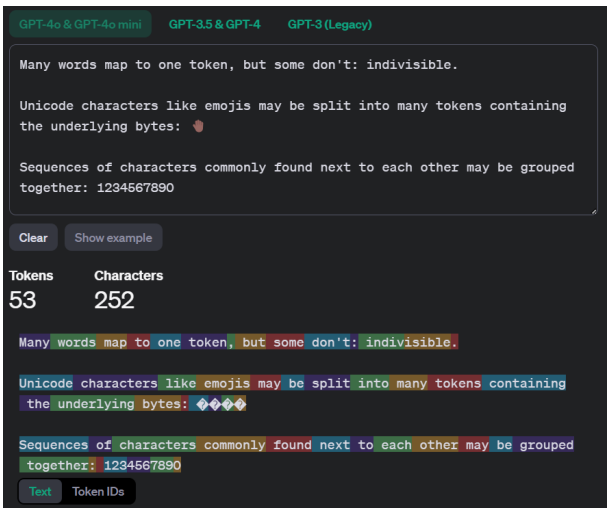

Figura 3.7: Ejemplo de Open AI de texto tokenizado.

Así mismo, en esta página se menciona que un token suele corresponder a unos 4 caracteres, lo que equivale aproximadamente a $3/4$ de palabra. Además de definir el vocabulario, la tokenización implica seguir ciertas reglas para asegurar que las secuencias de tokens sean coherentes. Un ejemplo es la regla que establece que el espacio debe ir solo a la izquierda de un carácter, pero nunca a la derecha, lo que garantiza que los tokens no representen espacios vacíos de manera ambigua. Estas reglas varían según el modelo y el corpus utilizado, pero son esenciales para evitar inconsistencias.

A pesar de sus ventajas, la tokenización también tiene limitaciones. Un problema común es el manejo de palabras o secuencias de texto completamente nuevas para el modelo. Aunque el enfoque basado en subpalabras, como BPE, ayuda a mitigar esto, no siempre garantiza que se conserve el significado original. Si una palabra se divide en demasiadas subpartes, el modelo podría perder el contexto necesario para interpretarla correctamente. Además, la elección del tamaño del vocabulario también influye. Un vocabulario pequeño podría resultar en una excesiva fragmentación de palabras, mientras que un vocabulario grande podría hacer que el modelo sea más costoso computacionalmente y menos eficiente. Por otro lado, la tokenización basada en reglas no siempre maneja adecuadamente las particularidades de todos los idiomas o dominios de texto. Los modelos que fueron entrenados principalmente en inglés, como los modelos GPT de Open AI, podrían tener dificultades al enfrentarse a lenguajes con diferentes estructuras gramaticales o sintácticas, como el chino o el árabe.

3.1.4. Definición Matemática de Embeddings

Consideremos un token $t \in \mathcal{T}$, donde $\mathcal{T}$ es el conjunto de índices correspondientes a los elementos de nuestro vocabulario $\mathcal{V}$. El embedding asigna a cada token un vector en un espacio de dimensión d_m, donde d_m es un hiperparámetro que determina la capacidad de representación del modelo. Definimos un mapa $\epsilon : \mathcal{T} \to \mathbb{R}^{d_m}$ tal que al token t se asocia a un vector x_t en $\mathbb{R}^{d_m}$:

$$\epsilon(t) = W_X^T e_t = x_t, \quad W_X^T \in \mathcal{L}(\mathbb{R}^{\mathbf{V}}, \mathbb{R}^{d_m}), \quad e_t \in \mathbb{R}^{\mathbf{V}}$$

donde W_X es la matriz $\mathbf{V} \times d_m$ de embeddings para todo el vocabulario. Este mapeo permite que el modelo transforme tokens en representaciones continuas aptas para operaciones algebraicas, además de que el utilizar una matriz dada por una transformación lineal nos permite que sus coeficientes sean parámetros aprendibles por el modelo durante el proceso de entrenamiento. Así mismo, podemos definir un mapa $\epsilon_\tau : \mathcal{T}^\tau \to \mathbb{R}^{\tau \times d_m}$ para tomar una secuencia de $\tau \in \mathbb{N}$ tokens $(t_0, \ldots, t_{\tau-1})$ y así obtener la matriz $\tau \times d_m$ de representaciones vectoriales:

$$\epsilon_\tau(t_0, t_1, \ldots, t_{\tau-1}) = \begin{bmatrix} e_{t_0} \\ e_{t_1} \\ \vdots \\ e_{t_{\tau-1}} \end{bmatrix} W_X = \begin{bmatrix} x_{t_0} \\ x_{t_1} \\ \vdots \\ x_{t_{\tau-1}} \end{bmatrix}.$$

3.2. Codificación Posicional

En un Transformer, no basta con saber qué palabras o tokens componen una oración o secuencia, sino que también es crucial conocer el orden en el que aparecen. El orden de las palabras influye directamente en el significado de una frase. Por ejemplo, “María le dio un libro a Juan” y “Juan le dio un libro a María” usan las mismas palabras, pero su significado cambia debido a la diferente disposición de los elementos. Por esta razón, un modelo debe tener información no solo sobre las palabras, sino también sobre su posición en la secuencia. Para incorporar esta información de orden, los Transformers utilizan **codificaciones posicionales** (*positional encodings*), que asignan un vector a cada posición de una secuencia. Esta codificación posicional permite que el modelo capture las relaciones entre palabras no solo por su proximidad en el texto, sino también por su posición relativa en la secuencia.

3.2.1. Formas de Codificación Posicional

Existen diversas formas de implementar las codificaciones posicionales. Una de las más conocidas es la presentada en el artículo “Attention Is All You Need” por Vaswani et al. (2023) que propone el uso de funciones sinusoidales para este propósito, lo que permite que cada posición se asocie con una combinación única de senos y cosenos de diferentes frecuencias. Estas funciones sinusoidales garantizan que el modelo pueda capturar tanto las relaciones locales

como globales entre los tokens y para secuencias de infinita longitud.

Se eligen funciones sinusoidales para las codificaciones posicionales debido a su periodicidad. Además, tener frecuencias distintas garantiza que las codificaciones sean linealmente independientes y permite que el modelo capte relaciones tanto locales como globales. También, su naturaleza periódica y continua facilita la extrapolación a secuencias más largas que las vistas durante el entrenamiento. Finalmente, al ser funciones deterministas, son computacionalmente eficientes y escalables, evitando la necesidad de aprender codificaciones específicas para cada posición.

Otra aproximación, como la utilizada en GPT-2, consiste en aprender las codificaciones posicionales directamente como parte del entrenamiento del modelo. En lugar de utilizar funciones fijas, el modelo entrena vectores posicionales que representan de manera óptima las posiciones de los tokens en la secuencia, ajustándolos de acuerdo a los datos específicos. Una desventaja de este último método es que el modelo solo puede interpretar una secuencia de longitud finita que se define al momento de entrenar el modelo.

Este último tipo de codificaciones posicionales se aprenden como vectores que se inicializan aleatoriamente antes del entrenamiento. Durante el entrenamiento, estos vectores son actualizados mediante retropropagación, es decir, a medida que el modelo ajusta sus parámetros (incluyendo las codificaciones posicionales), estos vectores se afinan para que el modelo pueda capturar de manera eficiente las relaciones entre los tokens en función de su posición en la secuencia. Este tipo de codificación no depende de funciones predefinidas, sino que permite que el modelo aprenda cuál es la representación más útil de cada posición, basándose en los datos con los que se entrena. Se obtienen ciertas ventajas como la adaptabilidad para tareas específicas, así como la flexibilidad de la forma que pueden tomar, permitiéndole al modelo captar estructuras de relaciones posicionales que no podrían modelarse fácilmente mediante funciones sinusoidales. Por otro lado, entre sus desventajas se encuentra la limitación a una longitud finita en las secuencias, al igual que un costo computacional adicional.

3.2.2. Definición Matemática de la Codificación Posicional

Consideremos una secuencia (ordenada) de tokens (t_i) donde la posición de cada token está representada por su respectivo índice $i \in \mathbb{Z}_{\geq 0}$. La codificación posicional se define como un mapa $\phi : \mathbb{N} \to \mathbb{R}^{d_m}$, donde d_m es la dimensión del modelo. Este mapa asocia a cada posición i un vector posicional $p_i \in \mathbb{R}^{d_m}$.

La codificación propuesta en el trabajo de Vaswani et al. (2023) con funciones sinusoidales es

$$(p_i)_{2k} = \sin\left(\frac{i}{10000^{2k/d_m}}\right), \quad (p_i)_{2k+1} = \cos\left(\frac{i}{10000^{2k/d_m}}\right)$$

donde podemos observar que está definida para toda posición $i \in \mathbb{N}$ y el valor en cada coordenada $k \in [d_m]$ dependerá de si es par o impar.

Por otro lado, en caso de utilizar codificaciones posicionales entrenadas, el mapa ϕ estaría definido de manera análoga a ϵ con una matriz W_P en lugar de la matriz W_X así como una distinta dimensionalidad para los vectores e_i; sin embargo, notemos que con las codificaciones posicionales entrenadas surge una restricción debido a cómo se define la matriz W_P. Recordemos que para W_X definimos la cantidad de filas de acuerdo a la cantidad de tokens en el vocabulario. En este caso necesitamos escoger un $M \in \mathbb{N}$ que representa la longitud máxima de las secuencias que admitirá el modelo. De esta manera, el dominio de ϕ se restringe a $[M]$, W_P es una matriz $M \times d_m$ tal que $W_P^T \in \mathcal{L}(\mathbb{R}^M, \mathbb{R}^{d_m})$, al igual que los vectores e_i usados en la definición de ϕ son elementos de $\mathbb{R}^M$. Así mismo, se obtiene la restricción $\tau \leq M$ para las secuencias $(t_0, \ldots, t_{\tau-1})$ que podremos procesar. Por ende, el mapa ϕ quedaría definido de la siguiente manera:

$$\phi(i) = W_P^T e_i = p_i, \quad W_P^T \in \mathcal{L}(\mathbb{R}^M, \mathbb{R}^{d_m}), \quad e_t \in \mathbb{R}^M$$

y así mismo podemos definir un mapa $\phi_k : [M]^k \to \mathbb{R}^{k \times d_m}$

$$\phi_k(i_0, i_1, \ldots, i_k) = \begin{bmatrix} e_{i_0} \\ e_{i_1} \\ \vdots \\ e_{i_k} \end{bmatrix} W_P = \begin{bmatrix} p_{i_0} \\ p_{i_1} \\ \vdots \\ p_{i_k} \end{bmatrix}.$$

De acuerdo a lo mencionado por Vaswani et al. (2023), no hay una diferencia significativa en los resultados al usar codificaciones sinusoidales y codificaciones aprendidas, debido a que se obtuvieron resultados casi idénticos tras experimentar con ambos métodos.

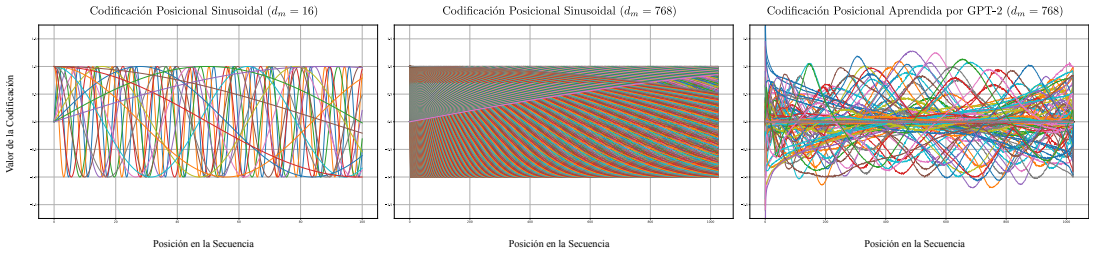

Figura 3.8: Comparación entre codificaciones con funciones sinusoidales y codificaciones aprendidas.

Hay diversas maneras de implementar la información posicional del token en una secuencia dentro de la representación vectorial que interpretará el Transformer, ya sea sumando ambos vectores de la misma dimensión d_m, o concatenando dos vectores de dimensión menor que juntos formen un vector de dimensión d_m. Si bien concatenarlos podría ofrecer mejores resultados asegurando que ambos subespacios vectoriales (embeddings y codificaciones posicionales) sean ortogonales, también aumentaría la complejidad del modelo y sería más costoso computacionalmente. Por otro lado, al sumar ambos vectores se han obtenido buenos resultados, indicando que el modelo aprende a representar los tokens como las posiciones de manera que se complementen entre sí al sumarlos.

En el caso que se analiza en este trabajo, se sumarán los vectores de embedding y codificación posicional, es decir, la representación vectorial $z_{t,i}$ de un token $t \in \mathcal{T}$ en la posición $i \in [M]$

está dada por un mapa $\psi : \mathcal{T} \times [M] \to \mathbb{R}^{d_m}$ definido como:

$$\psi(t, i) = \epsilon(t) + \phi(i) = x_t + p_i = z_{t,i}.$$

De esta manera, sea una secuencia de tokens $(t_0, \ldots, t_{\tau-1}) \in \mathcal{T}^\tau$, $\tau \leq M$, podemos definir un mapa $\psi_\tau : \mathcal{T}^\tau \to \mathbb{R}^{\tau \times d_m}$ dado por

$$\psi_\tau(t_0, \ldots, t_{\tau-1}) = \epsilon_\tau(t_0, \ldots, t_{\tau-1}) + \phi_\tau(0, \ldots, \tau-1) = \begin{bmatrix} z_{t_0,0} \\ z_{t_1,1} \\ \vdots \\ z_{t_{\tau-1},\tau-1} \end{bmatrix}.$$

Consideremos ahora la secuencia de tokens “el perro persigue al gato”. A cada palabra le asignamos un vector de embedding x_t y una codificación posicional p_i, como se muestra a continuación:

$$\begin{array}{cc} \text{el} & x_{t_0} \\ \text{perro} & x_{t_1} \\ \text{persigue} & x_{t_2} \\ \text{al} & x_{t_3} \\ \text{gato} & x_{t_4} \end{array} \begin{bmatrix} 1.3 & -0.2 & 0.7 \\ 4.1 & 4.7 & 4.9 \\ 0.9 & 5.3 & 6.7 \\ 2.3 & -5.4 & -0.8 \\ 3.5 & 6.9 & 4.2 \end{bmatrix}, \quad \begin{array}{c} p_0 \\ p_1 \\ p_2 \\ p_3 \\ p_4 \end{array} \begin{bmatrix} 1.2 & -0.9 & -1.1 \\ 0.1 & 0.8 & 0.1 \\ 0.4 & -3.0 & 1.4 \\ 2.0 & 0.9 & -0.2 \\ -0.4 & 0.2 & -0.7 \end{bmatrix}.$$

El siguiente paso es sumar las representaciones de los tokens y las posiciones, tal como se describe en la fórmula original:

$$z_{t,i} = x_t + p_i.$$

Al realizar esta suma, obtenemos la representación final de cada token en la secuencia considerando tanto su significado (embeddings) como su posición (codificación posicional). El resultado

es la siguiente representación vectorial:

$$\begin{matrix} z_{t_{0,0}} \\ z_{t_{1,1}} \\ z_{t_{2,2}} \\ z_{t_{3,3}} \\ z_{t_{4,4}} \end{matrix} \begin{bmatrix} 2.5 & -1.1 & -0.4 \\ 4.2 & 5.5 & 5.0 \\ 1.3 & 2.3 & 8.1 \\ 4.3 & -4.5 & -1.0 \\ 3.1 & 7.1 & 3.5 \end{bmatrix}.$$

De esta manera, la suma de los vectores de embedding y codificación posicional permite que el modelo obtenga una representación más completa de cada token. En lugar de tratar el significado de las palabras y su posición como conceptos separados, ambos se combinan en un solo vector que el modelo puede procesar de manera eficiente. En la figura 3.9 se puede apreciar como se combinan los embeddings contextuales (vectores rojos) con las codificaciones posicionales (vectores azules) para formar las representaciones vectoriales finales (vectores negros).

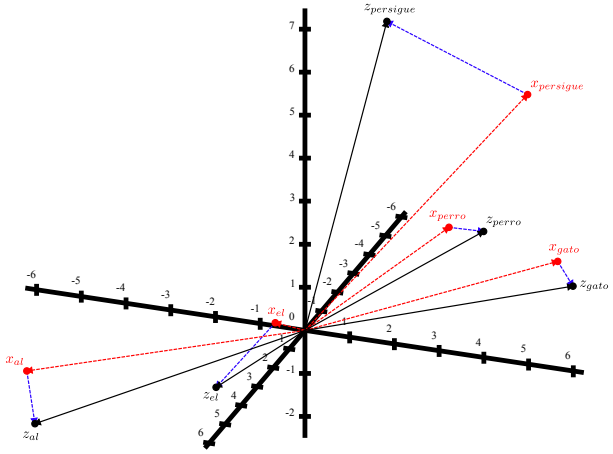

Figura 3.9: Combinación de embeddings y codificaciones posicionales.

3.2.3. Complementariedad de Representaciones Vectoriales

Uno de los desafíos clave de combinar embeddings y codificaciones posicionales es asegurarse de que estos vectores no se sobrescriban entre sí cuando se elige sumarlos para obtener la representación vectorial final que interpretará el modelo. Idealmente, los embeddings de los tokens y las codificaciones posicionales deben ser "casi" linealmente independientes, de modo que la información semántica del embedding y la información posicional se complementen sin

interferir. Es decir, sean $\mathcal{Z}, \mathcal{X}, \mathcal{P} \subseteq \mathbb{R}^{d_m}$ subespacios vectoriales tales que

$$\mathcal{X} = \text{span}\{x_t : t \in \mathcal{T}\}, \quad \mathcal{P} = \text{span}\{p_i : i \in [M]\},$$

$$\mathcal{Z} = \text{span}\{z_{t,i} : t \in \mathcal{T}, i \in [M]\},$$

idealmente se debería obtener que $\mathcal{Z} = \mathcal{X} \oplus \mathcal{P}$; sin embargo, en la realidad no se obtiene un resultado tan perfecto que independice completamente la semántica de la palabra con la posición de la misma, aunque en ciertos casos si se obtiene una independencia aproximada que permite recuperar mayor parte de la información semántica o posicional partiendo de un vector $z \in \mathcal{Z}$.

3.3. Mecanismo de Atención

Cuando procesamos una secuencia de palabras, como un texto o una oración, no basta con considerar solo las palabras individuales, sino también las relaciones que existen entre ellas. En muchos casos, las palabras dependen de otras palabras en diferentes posiciones para dar sentido completo al mensaje. Así como cuando leemos un libro, no recordamos todos los detalles a la vez, sino que prestamos más atención a ciertos pasajes clave para entender mejor la historia. El mecanismo que se presentará en esta sección permite a los modelos enfocar su "atención" en partes relevantes de la entrada, ayudándoles a capturar mejor estas relaciones en la secuencia y mejorando así su capacidad de generar o entender texto de manera coherente.

El mecanismo de atención en los transformadores se basa en tres componentes: **queries** (*consultas*), **keys** (*claves*) y **values** (*valores*). Estos tres componentes forman la base de cómo el modelo selecciona y pondera la información relevante de la secuencia de entrada. Los términos provienen del concepto de búsqueda en bases de datos. Imagina que estás buscando información en una base de datos. El query es tu pregunta o búsqueda, los keys son identificadores que ayudan a encontrar lo que buscas, y los values son los datos que finalmente recuperas. En el contexto del Transformer, sucede algo similar, pero aplicado a palabras y contexto dentro de una secuencia de texto.

Para comprender mejor esto, veamos un ejemplo de queries en bases de datos. Supongamos que tenemos una pequeña base de datos con información sobre frutas. La base de datos tiene tres columnas: **Fruta**, **Color**, y **Precio**.

ID	Fruta	Color	Precio
1	Manzana	Roja	1.00
2	Plátano	Amarillo	0.50
3	Naranja	Naranja	0.75

Cuadro 3.1: Ejemplo de query y keys en bases de datos.

Imagina que quieres saber el precio de una fruta específica, por ejemplo, "Plátano". Este sería tu query, es decir, lo que estás buscando. Los keys son las frutas que están en la base de datos. Estos se usarán para hacer coincidir tu búsqueda. En este caso, los keys son los valores en la columna "Fruta", es decir, "Manzana", "Plátano", y "Naranja". Los values son los datos que te interesa recuperar. En este caso, los values son los precios de las frutas, que están en la columna "Precio".

En un Transformer, el query es un vector que representa lo que el modelo está "buscando" en relación a un token particular, mientras que los keys son vectores que describen lo que cada token "ofrece" en términos de información. El modelo calcula la similitud entre el query y todos los keys utilizando un producto escalar, para determinar qué tokens en la secuencia de entrada contienen la información más relevante para el query. Una vez que se determina esta relevancia, se ponderan los values correspondientes, que son los vectores que contienen la información real de los tokens, y se combinan para producir la salida final.

En este trabajo se está analizando la arquitectura de un Transformer de solo decodificador para generación de texto, cuyo objetivo es, a partir de una entrada de una secuencia de $\tau \leq M$ tokens $(t_0, \ldots, t_{\tau-1})$, predecir el siguiente token en la secuencia, t_τ. Lo que hace el modelo

internamente es tomar el último token debido a que se intenta predecir el siguiente y se genera un query que pregunta "¿qué palabras anteriores me pueden dar contexto?". Los keys de todos los tokens anteriores (incluyendo el mismo) se comparan con este query mediante producto escalar para cuantificar la similitud entre los vectores. De esta manera, aquellos tokens cuyos keys más similares (es decir, que tengan una alta relevancia con respecto al query) tendrán mayor peso en la predicción. Finalmente, los values de esos tokens seleccionados (que contienen la información que aportan al contexto) se combinan mediante suma ponderada para ayudar a predecir el siguiente token en la secuencia.

3.3.1. Definición Matemática de Queries, Keys y Values

Dados los tokens de entrada $(t_0, t_1, \ldots, t_{\tau-1})$, $\tau \leq M$, y sus representaciones vectoriales $z_{t_i,i} \in \mathcal{Z} = \mathbb{R}^{d_m}$ (para abreviar la notación, nos referiremos a $z_{\tau_i,i}$ como solamente z_i a partir de este punto), los queries, keys y values se obtienen aplicando transformaciones (generalmente lineales) sobre las representaciones de los tokens. Asumiremos el uso de transformaciones lineales por simplicidad, ya que estas facilitan una mejor comprensión matemática del modelo y proporcionan una forma natural de generalizar las relaciones entre los tokens, sin introducir la complejidad adicional de funciones no lineales que podrían dificultar el análisis teórico. Sean $Z \in \mathbb{R}^{\tau \times d_m}$ la matriz que contiene las representaciones z_i de todos los tokens de la secuencia:

$$Z = \begin{bmatrix} z_0 \\ z_1 \\ \vdots \\ z_{\tau-1} \end{bmatrix} = \psi_\tau(t_0, \ldots, t_{\tau-1}),$$

$X \in \mathbb{R}^{\tau \times d_m}$ la matriz de embeddings x_{t_i}:

$$X = \begin{bmatrix} x_{t_0} \\ x_{t_1} \\ \vdots \\ x_{t_{\tau-1}} \end{bmatrix} = \epsilon_\tau(t_0, \ldots, t_{\tau-1}),$$

y $P \in \mathbb{R}^{\tau \times d_m}$ la matriz de codificaciones posicionales p_i:

$$P = \begin{bmatrix} p_0 \\ p_1 \\ \vdots \\ p_{\tau-1} \end{bmatrix} = \phi_\tau(0, \ldots, \tau - 1).$$

Para obtener las matrices Q, K y V correspondientes a queries, keys y values, se aplican las siguientes transformaciones lineales con matrices de pesos W_Q, W_K y W_V respectivamente:

$$Q = ZW_Q \in \mathbb{R}^{\tau \times d_k},$$
$$K = ZW_K \in \mathbb{R}^{\tau \times d_k},$$
$$V = ZW_V \in \mathbb{R}^{\tau \times d_k},$$

donde W_Q y $W_K \in \mathbb{R}^{d_m \times d_k}$ son las matrices de pesos que mapean los vectores $z_i \in \mathcal{Z}$ a un nuevo espacio vectorial de atención $\mathcal{A}$ de dimensión d_k. Así mismo, $W_V \in \mathbb{R}^{d_m \times d_v}$ mapea los vectores $z_i \in \mathcal{Z}$ a otro espacio $\mathcal{V}$ de dimensión d_v en el cual los values contienen la información relevante que será ponderada y combinada en el siguiente paso (más adelante se darán detalles sobre qué dimensiones d_k y d_v se deberían escoger junto con su propósito). En otras palabras, $W_Q^T, W_K^T \in \mathcal{L}(\mathcal{Z}, \mathcal{A})$ y $W_V^T \in \mathcal{L}(\mathcal{Z}, \mathcal{V})$. Esto significa que:

$$q_i = (W_Q)^T z_i \in \mathcal{A} \subseteq \mathbb{R}^{d_k},$$
$$k_i = (W_K)^T z_i \in \mathcal{A} \subseteq \mathbb{R}^{d_k},$$
$$v_i = (W_V)^T z_i \in \mathcal{V} \subseteq \mathbb{R}^{d_v},$$

donde q_i, k_i y v_i son los vectores que representan al token t_i en los espacios de queries, keys y values, respectivamente. El objetivo es comparar cada query q_i con todos los keys k_j para determinar cuánta atención debe prestar el modelo a cada token en el contexto de la secuencia.

3.3.2. Producto Escalar y Similitud entre Queries y Keys

El nivel de atención que se le otorga a un token se mide mediante el producto escalar entre los queries y los keys. Este producto escalar indica la similitud entre un query q_i y un key k_j, es

decir, cuánta relevancia tiene el token t_j en el contexto del token t_i. Formalmente:

$$q_i^T k_j = (W_Q)^T z_i \cdot (W_K)^T z_j = z_i^T (W_Q W_K^T) z_j = z_i^T (W_A z_j),$$

donde $W_A = W_Q W_K^T \in \mathcal{L}(\mathcal{Z})$ es un operador en $\mathcal{Z}$. Este producto escalar proporciona un valor que indica cuánta atención debe prestarse al token t_j al procesar el token t_i. De esta manera podemos obtener una matriz de atención

$$A = Z W_A Z^T = (X + P) W_A (X + P)^T \in \mathbb{R}^{\tau \times \tau},$$

donde el vector de puntajes de atención de un token de query t_i sería

$$A_{i,\cdot} = z_i^T W_A Z^T = (x_{t_i}^T + p_i^T) W_A (X + P)^T \in \mathbb{R}^{\tau},$$

y el puntaje de un token de query t_i con un token de key t_j sería

$$A_{i,j} = z_i^T (W_A z_j) = (x_{t_i}^T + p_i^T) W_A (x_{t_j} + p_j) \in \mathbb{R}.$$

Para evitar que los valores del puntaje de atención crezcan demasiado a medida que aumenta la dimensión de los vectores d_k, se escala este valor dividiéndolo por $\sqrt{d_k}$. Luego, se aplica la función **Softmax** (σ) a las filas para convertir estas similitudes escaladas en probabilidades o ponderaciones que sumen 1, las cuales representan los pesos de atención que se asignarán a los values para cada query:

$$\sigma\left(\frac{1}{\sqrt{d_k}} A_{i,\cdot}\right) = \frac{\exp\left(\frac{1}{\sqrt{d_k}} A_{i,\cdot}\right)}{\sum_{t=0}^{\tau-1} \exp\left(\frac{1}{\sqrt{d_k}} A_{i,t}\right)},$$

$$\sigma\left(\frac{1}{\sqrt{d_k}} A_{i,\cdot}\right)_j = \frac{\exp\left(\frac{q_i^T k_j}{\sqrt{d_k}}\right)}{\sum_{t=0}^{\tau-1} \exp\left(\frac{q_i^T k_t}{\sqrt{d_k}}\right)}.$$

Estos pesos de atención determinan la contribución de cada value v_j a la salida final del token t_i. La salida del mecanismo de atención para el token t_i es entonces una suma ponderada de los values v_j correspondientes a cada token t_j:

$$\text{Self-Attention}(Z)_{i,\cdot} = \sigma\left(\frac{1}{\sqrt{d_k}} A_{i,\cdot}\right) V = \sigma\left(\frac{Z_{i,\cdot} W_A Z^T}{\sqrt{d_k}}\right) Z W_V \in \mathcal{V}.$$

Se le denomina "Autoatención" (o Self-Attention) debido a que los queries Q y los keys K provienen de una misma entrada Z, es decir, los tokens se prestan atención a sí mismos en lugar de atender a otro contexto. Además, originalmente para el decoder se menciona una "Atención Enmascarada" (Masked Attention) en la que se anulan ciertos valores de la matriz A para evitar prestar atención a ciertos tokens con respecto a alguna condición dada. En este ejemplo, debido a que se procura predecir la siguiente palabra, se utiliza la llamada "Atención Causal" (Causal Attention, que es un tipo de Masked Attention) en la cual se aplica una máscara M a la atención A para evitar que el modelo pueda "ver" tokens futuros y, de esta manera, al momento de entrenar, no acceda a tokens que no han sido generados. Esta máscara se representa mediante una matriz triangular inferior de unos:

$$M_{ij} = \begin{cases} 1, & j \leq i, \\ -\infty, & j > i. \end{cases}$$

La matriz M se aplica mediante el producto elemento a elemento $\odot$:

$$\text{Causal-Attention}(Z)_{i,\cdot} = \sigma\left(\frac{1}{\sqrt{d_k}} (A \odot M)_{i,\cdot}\right) V = \sigma\left(\frac{(Z_{i,\cdot} W_A Z^T) \odot M}{\sqrt{d_k}}\right) Z W_V.$$

3.3.3. Ejemplo de Atención Causal

Veamos un ejemplo simple para facilitar la comprensión de este mecanismo. Supongamos que estamos evaluando la frase “el perro persigue al gato”, en este caso vamos a suponer que tenemos las siguientes representaciones vectoriales:

- el: $z_{\text{el}}(0.25, -0.11, -0.04)$.
- perro: $z_{\text{perro}}(-0.42, 0.55, 0.50)$.
- persigue: $z_{\text{persigue}}(-0.13, 0.23, 0.81)$.
- al: $z_{\text{al}}(0.43, -0.45, -0.10)$.
- gato: $z_{\text{gato}}(-0.31, 0.71, 0.35)$.

$$Z = \begin{bmatrix} 0.25 & -0.11 & -0.04 \\ -0.42 & 0.55 & 0.50 \\ -0.13 & 0.23 & 0.81 \\ 0.43 & -0.45 & -0.10 \\ -0.31 & 0.71 & 0.35 \end{bmatrix}$$

Así mismo, supongamos que tenemos las siguientes matrices correspondientes a las transformaciones lineales de queries, keys y values:

$$W_Q = \begin{bmatrix} 0.64 & -0.44 & 0.29 \\ 0.14 & 0.39 & -0.63 \\ -0.26 & 0.02 & -0.15 \end{bmatrix}, \quad W_K = \begin{bmatrix} 0.03 & -2.04 & 0.41 \\ 0.49 & 0.10 & -1.60 \\ -0.41 & -0.31 & -0.12 \end{bmatrix},$$

$$W_V = \begin{bmatrix} 0.21 & 0.53 & -0.02 \\ -0.24 & 0.16 & 0.87 \\ 0.05 & -0.37 & 0.45 \end{bmatrix}.$$

De esta manera, obtenemos las matrices de queries, keys y values:

$$Q = ZW_Q = \begin{bmatrix} 0.1550 & -0.1537 & 0.1478 \\ -0.3218 & 0.4093 & -0.5433 \\ -0.2616 & 0.1631 & -0.3041 \\ 0.2382 & -0.3667 & 0.4232 \\ -0.1900 & 0.4203 & -0.5897 \end{bmatrix},$$

$$K = ZW_K = \begin{bmatrix} -0.0300 & -0.5086 & 0.2833 \\ 0.0519 & 0.7568 & -1.1122 \\ -0.2233 & 0.0371 & -0.5185 \\ -0.1666 & -0.8912 & 0.9083 \\ 0.1951 & 0.5949 & -1.3051 \end{bmatrix},$$

$$V = ZW_V = \begin{bmatrix} 0.0769 & 0.1297 & -0.1187 \\ -0.1952 & -0.3196 & 0.7119 \\ -0.0420 & -0.3318 & 0.5672 \\ 0.1933 & 0.1929 & -0.4451 \\ -0.2180 & -0.1802 & 0.7814 \end{bmatrix}.$$

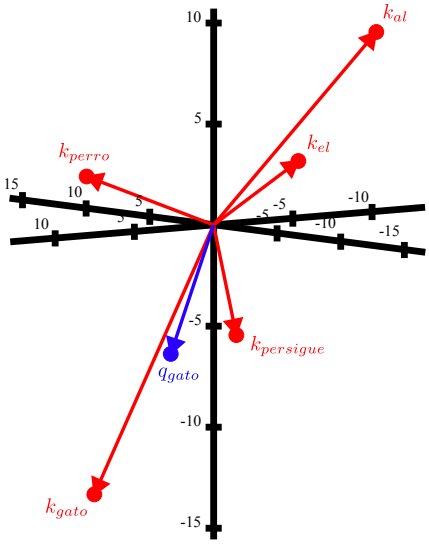

Figura 3.10: Vector de query para la palabra gato comparada con los vectores key del resto de palabras.

Ahora ya podemos calcular la atención y el resultado de Causal-Attention(Z). Primero obtenemos la matriz de atención (por esta ocasión no escalaremos por el factor $(1/\sqrt{d_k})$ para simplificar cálculos, así mismo aproximamos a 4 decimales):

$$A = QK^T$$

$$= \begin{bmatrix} 0.1550 & -0.1537 & 0.1478 \\ -0.3218 & 0.4093 & -0.5433 \\ -0.2616 & 0.1631 & -0.3041 \\ 0.2382 & -0.3667 & 0.4232 \\ -0.1900 & 0.4203 & -0.5897 \end{bmatrix} \begin{bmatrix} -0.0300 & 0.0519 & -0.2233 & -0.1666 & 0.1951 \\ -0.5086 & 0.7568 & 0.0371 & -0.8912 & 0.5949 \\ 0.2833 & -1.1122 & -0.5185 & 0.9083 & -1.3051 \end{bmatrix}$$

$$= \begin{array}{c|ccccc} query \backslash key & \text{el} & \text{perro} & \text{persigue} & \text{al} & \text{gato} \\ \hline \text{el} & 0.1154 & -0.2727 & -0.1169 & 0.2454 & -0.2541 \\ \text{perro} & -0.3524 & 0.8973 & 0.3687 & -0.8046 & 0.8898 \\ \text{persigue} & -0.1613 & 0.4481 & 0.2221 & -0.3780 & 0.4429 \\ \text{al} & 0.2993 & -0.7358 & -0.2862 & 0.6715 & -0.7240 \\ \text{gato} & -0.3751 & 0.9641 & 0.3638 & -0.8785 & 0.9826 \end{array}$$

Una vez tenemos calculada la matriz de atención, debemos aplicar la máscara para convertirla en una matriz triangular inferior $A \odot M$. De esta manera evitamos que palabras como "perro" presten atención a palabras futuras como "al" o como "gato" que van después en la oración:

$query \backslash key$	el	perro	persigue	al	gato
el	0.1154	$-\infty$	$-\infty$	$-\infty$	$-\infty$
perro	-0.3524	0.8973	$-\infty$	$-\infty$	$-\infty$
persigue	-0.1613	0.4481	0.2221	$-\infty$	$-\infty$
al	0.2993	-0.7358	-0.2862	0.6715	$-\infty$
gato	-0.3751	0.9641	0.3638	-0.8785	0.9826

Al momento de aplicar la función Softmax $\sigma(A \odot M)$, los $-\infty$ se convierten en 0, resultando

en lo siguiente:

$query \backslash key$	el	perro	persigue	al	gato
el	1.0000	0.0000	0.0000	0.0000	0.0000
perro	0.2227	0.7773	0.0000	0.0000	0.0000
persigue	0.2322	0.4271	0.3407	0.0000	0.0000
al	0.2973	0.1056	0.1656	0.4315	0.0000
gato	0.0877	0.3347	0.1836	0.0530	0.3409

Ahora que ya sabemos las ponderaciones de cada key con respecto con cada query, realizamos la suma ponderada de values:

$$\text{Causal-Attention}(Z) = \sigma(A \odot M)V = \begin{bmatrix} 0.0769 & 0.1297 & -0.1187 \\ -0.1346 & -0.2195 & 0.5269 \\ -0.0798 & -0.2194 & 0.4697 \\ 0.0787 & 0.0331 & -0.0582 \\ -0.1304 & -0.2077 & 0.5748 \end{bmatrix}$$

3.3.4. Atención Multi-Cabeza

La Atención Multi-Cabeza (Multi-Head Attention) es una extensión del mecanismo de autoatención que permite al modelo enfocarse en diferentes subespacios de atención simultáneamente. En lugar de calcular un único puntaje de atención para cada par de tokens, la atención multi-cabeza divide los vectores de query, key y value en múltiples subespacios de menor dimensión. Cada subespacio realiza su propia atención de forma independiente, y los resultados de todas las cabezas de atención se combinan posteriormente. Este enfoque mejora la capacidad del modelo para capturar relaciones complejas entre los tokens en diferentes subespacios de características.

Supongamos que tenemos h cabezas de atención. En cada cabeza m, los vectores de query, key y value se proyectan linealmente a un espacio de dimensión reducida d_k mediante matrices

de proyección W_{Q_m}, W_{K_m} y W_{V_m}, respectivamente. La atención H_m para la cabeza m se calcula de manera similar al mecanismo de atención descrito previamente:

$$A_m = \frac{Q_m (K_m)^T}{\sqrt{d_k}}, \quad H_m = \sigma\left(\frac{A_m}{\sqrt{d_k}}\right) V_m,$$

$$\implies H_m = \sigma\left(\frac{Q_m (K_m)^T}{\sqrt{d_k}}\right) V_m$$

donde $Q_m = ZW_{Q_m}$, $K_m = ZW_{K_m}$ y $V_m = ZW_{V_m}$ son las proyecciones lineales de los vectores de entrada Z. Los resultados de todas las cabezas de atención se concatenan y luego se proyectan nuevamente a la dimensión original del espacio de entrada mediante una matriz de proyección $W_O \in \mathcal{L}(\mathcal{V}, \mathcal{Z})$ y se agrega un sesgo $b_O \in \mathcal{Z}$:

$$\text{MHSA}(Z) = \text{Concat}\left(H_1, H_2, \ldots, H_h\right) W_O + b_O,$$

Así mismo, podemos procesarlo con la máscara causal de la siguiente manera:

$$\text{MHCA}(Z) = \text{Concat}(C_1, C_2, \ldots, C_h) W_O + b_O,$$

donde $C_m = \text{Causal-Attention}_m(Z)$. Este proceso permite que el modelo aprenda a representar las relaciones entre tokens desde diferentes perspectivas, ya que cada cabeza puede enfocarse en un aspecto particular de la información. Si bien cada cabeza de atención opera de manera independiente, la información de todas ellas se integra para generar una salida que combina las distintas representaciones de los tokens. La ventaja de utilizar múltiples cabezas es que cada una de ellas puede capturar patrones o dependencias diferentes, lo que aumenta la capacidad del modelo para aprender de forma más rica y compleja los contextos entre los tokens.

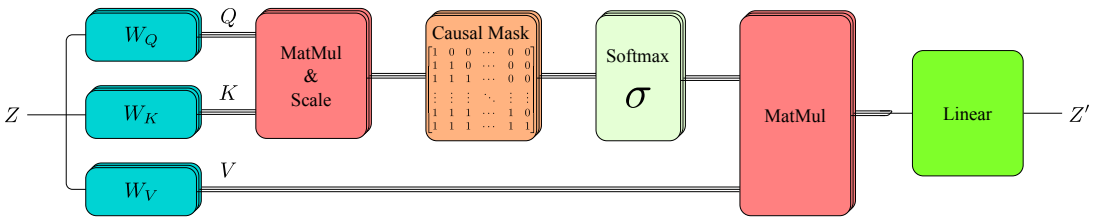

Figura 3.11: Arquitectura de atención causal Multi-Cabeza.

Para que este proceso sea eficiente, generalmente se escoge una dimensión d_m que sea una potencia de 2 y así mismo una cantidad de cabezas h tal que d_m sea divisible para h. De esta forma definimos las dimensiones $d_k = d_v = \frac{d_m}{h}$. El uso de potencias de 2 en las dimensiones de los datos no es algo arbitrario sino por optimización. En computadoras modernas, especialmente en arquitecturas paralelas como GPUs, las operaciones que involucran potencias de 2 son mucho más rápidas. Por ejemplo, la división y el cálculo del módulo de un número por una potencia de 2 pueden ser reemplazados por operaciones de desplazamiento y enmascarado de bits, lo que reduce significativamente el costo computacional. Estas optimizaciones son manejadas automáticamente por los compiladores cuando n es una constante literal, aprovechando las características del sistema binario y mejorando la eficiencia de la ejecución en hardware como las GPUs (NVIDIA Corporation, s.f.).

3.4. Arquitectura del Transformer: Atención, Conexiones Residuales y Perceptrones Multicapa

Aunque el mecanismo de atención permite capturar dependencias entre los elementos de una secuencia, en muchos casos no es suficiente procesar la información solo una vez. Las representaciones obtenidas mediante atención pueden ser mejoradas y refinadas para capturar características más complejas y abstractas de los datos. Es aquí donde entra en juego la necesidad de añadir capas adicionales, como las redes MLP y las conexiones residuales, que no solo permiten realizar transformaciones no lineales, sino que también estabilizan y facilitan el aprendizaje en redes profundas. Además, se utiliza una arquitectura de bloques apilados que permite al modelo aprender representaciones cada vez más abstractas y complejas. Al igual que en una red neuronal profunda, cada bloque sucesivo refina y transforma la información de los bloques anteriores, lo que permite al Transformer captar estructuras más sofisticadas y dependencias a largo plazo en los datos. En esta sección, exploraremos cómo estos bloques de atención y MLP se combinan, cómo interactúan con las normalizaciones de capa y cómo el apilamiento de estos bloques permite al Transformer aprender representaciones cada vez más complejas, mejorando su capacidad para modelar dependencias de largo alcance y realizar tareas de predicción de manera eficiente.

3.4.1. Normalización de Capas (Layer Normalization)

La Normalización de Capas o Layer Normalization (Ba et al., 2016) es una operación clave para estabilizar y acelerar el entrenamiento de redes neuronales profundas. Se utiliza para estabilizar el entrenamiento y mejorar la eficiencia del aprendizaje. Durante el proceso de atención, los valores pueden variar significativamente e incluso se pueden volver demasiado grandes o pequeños, dificultando la convergencia del modelo. La normalización de capa ayuda a mitigar este problema al ajustar las activaciones dentro de cada capa para que tengan una media cercana a cero y una desviación estándar unitaria. Recordemos la definición de Layer Norm (2.2.5), en este caso, adaptada para aceptar como entrada a nuestra matriz de representaciones Z:

$$\text{LayerNorm}(Z) = \frac{Z - \mu_Z}{\sigma_Z} \odot \gamma + \beta$$

donde:

- $\mu_Z = \frac{1}{d_m} \sum_{i=1}^{d} Z_{\cdot,i}$ es el vector de medias de cada z_i, o el promedio de las columnas de Z,
- $\sigma_Z = \sqrt{\frac{1}{d_m} \sum_{i=1}^{d} (Z_{\cdot,i} - \mu)^2}$ es la desviación estándar de cada z_i, o de las columnas de Z,
- $\gamma, \beta \in \mathcal{Z} \cong \mathbb{R}^{d_m}$ son parámetros aprendibles de escala y desplazamiento, respectivamente.

Esta operación asegura que cada representación vectorial de los tokens sea normalizada y tenga la misma escala y desplazamiento que el resto para que el funcionamiento del modelo sea lo más eficiente y estable posible. En el paper de "Attention Is All You Need", la normalización se realiza después de cada capa del Transformer; sin embargo, varios modelos más recientes como en GPT-2 (Radford et al., 2019) se lo usa antes, en la entrada de cada capa. Por lo tanto, con la normalización, una capa de atención multi-cabeza tendría la siguiente estructura:

$$\text{MHCA}(\text{LayerNorm}(Z))$$

3.4.2. Conexiones Residuales

En redes neuronales profundas, uno de los problemas más comunes es el desvanecimiento del gradiente, que ocurre cuando los gradientes de la función de pérdida se vuelven extremadamente pequeños a medida que se retropropagan a través de las capas. Este fenómeno dificulta el ajuste de los pesos, especialmente en capas más cercanas a la entrada, lo que ralentiza o incluso detiene el entrenamiento de la red. Una solución clave para este problema en la arquitectura Transformer es la introducción de conexiones residuales, que permiten que la entrada de una capa se sume directamente a su salida. En el caso de una capa de atención multi-cabeza, su salida se expresa como:

$$Z' = Z + \text{MHCA}(\text{LayerNorm}(Z)).$$

La suma de la entrada Z con la salida de la atención multi-cabeza garantiza que la información original fluya directamente a través de las capas, sin ser completamente modificada por la atención. Esto facilita la retropropagación de gradientes, mitigando el desvanecimiento al permitir que los gradientes se propaguen más fácilmente a través de las capas, mejorando la estabilidad y la velocidad del entrenamiento.

3.4.3. Perceptrón Multicapa (MLP)

Una de las claves para aprender representaciones complejas y abstractas es introducir no linealidades en el modelo. Las capas MLP (Perceptrones Multicapa o Multilayer Perceptron, también conocidas como red neuronal Feed Forward) juegan un papel fundamental en este sentido, ya que permiten que el modelo realice transformaciones no lineales en las representaciones de los datos. Sin la presencia de estas capas no lineales, el modelo se limitaría a aprender combinaciones lineales de las entradas, lo que restringiría su capacidad para capturar relaciones complejas en los datos.

Desde una perspectiva matemática, un MLP consiste típicamente en una secuencia de transformaciones lineales seguidas de una función de activación no lineal. Para una entrada x, la

operación en una capa MLP se puede expresar como:

$$y = f(Wx + b),$$

donde W es la matriz de pesos, b es el sesgo y f es la función de activación no lineal. Comúnmente para la función de activación se utiliza ReLU (2.2.3), o, en el caso de ciertos modelos como los GPT (Radford & Narasimhan, 2018), se utiliza GELU (2.2.4). Esta última tiene la ventaja de ser continuamente diferenciable en todo su dominio, mientras que ReLU no lo es en 0. Además, el usar esta función de activación iguala o incluso supera los resultados de los modelos con ReLU (Hendrycks & Gimpel, 2023, p. 1).

En un Transformer, el MLP se encuentra típicamente después de la capa de atención y está compuesto por dos capas lineales con una función de activación no lineal en medio. Específicamente, si la salida de la capa de atención es Z', entonces el MLP se aplica de la siguiente manera:

$$\text{MLP}(Z') = \text{GELU}(Z'W_1 + b_1)W_2 + b_2,$$

donde $W_1 \in \mathbb{R}^{d_m \times 4d_m}$, $W_2 \in \mathbb{R}^{4d_m \times d_m}$, $b_1 \in \mathbb{R}^{4d_m}$ y $b_2 \in \mathbb{R}^{d_m}$. Esta estructura permite que las representaciones de los tokens se transformen de manera más expresiva y no lineal, mejorando la capacidad del Transformer para aprender características complejas. Junto con las conexiones residuales, que ayudan a evitar la pérdida de información, el MLP refina aún más las representaciones aprendidas, permitiendo que el Transformer capture patrones más sofisticados y realice predicciones más precisas. La combinación de la atención, las conexiones residuales y los MLP asegura que el modelo no solo capture dependencias locales y globales en los datos, sino que también pueda aprender representaciones no lineales de alta dimensión que son esenciales para tareas complejas.

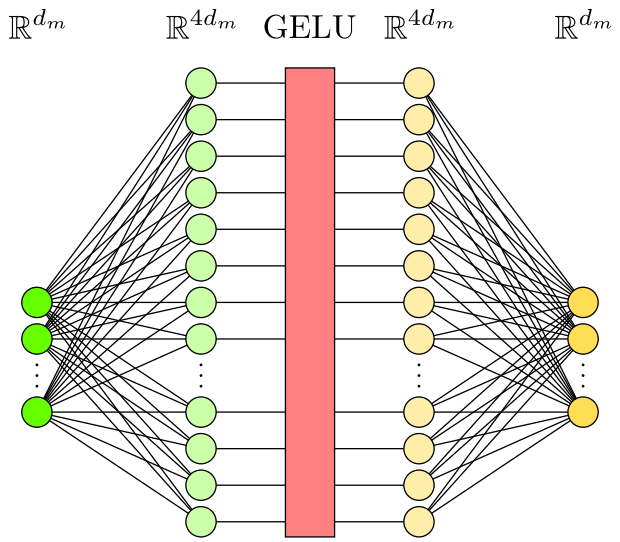

Figura 3.12: Arquitectura del Perceptrón Multicapa (MLP).

3.4.4. Bloque Transformer

Ahora que ya conocemos todas las herramientas necesarias para el Transformer, podemos construir un bloque de Transformer. El bloque Transformer se construye mediante una secuencia de operaciones que integran atención, normalización de capas, conexiones residuales y una capa MLP. Para un bloque tipo solo decoder tendríamos la siguiente arquitectura con atención causal:

$$\mathcal{T}(Z) = Z' + \text{MLP}(\text{LayerNorm}(Z + \text{MHCA}(\text{LayerNorm}(Z))),$$

debido a que en un bloque decoder se procura generar una secuencia de salida de manera autoregresiva, un token a la vez y sin la posibilidad de ver tokens futuros (causal). Por otro lado, un bloque encoder simplemente codifica o procesa una secuencia de entrada para generar una interpretación interna de esa información, procesando todos los tokens a la vez y cada uno prestando atención al resto. Para un bloque solo encoder tendríamos la misma estructura con autoatención, es decir, sin máscara causal:

$$\mathcal{T}_e(Z) = Z' + \text{MLP}(\text{LayerNorm}(Z + \text{MHSA}(\text{LayerNorm}(Z))).$$

En un modelo encoder-decoder, existe una capa adicional en el bloque decoder de atención cruzada (Cross-Attention), donde los queries se generan a partir de la salida procesada en la capa de atención causal, mientras que los keys y values se generan a partir de la salida del encoder.

Pese a la complejidad de un bloque Transformer, un solo bloque no suele ser suficiente para interpretar correctamente el lenguaje y aprender ciertas propiedades del mismo. Por este motivo, se utilizan múltiples capas apiladas de bloques Transformer durante el entrenamiento, lo que permite que el modelo refine progresivamente las representaciones de los datos a medida que avanza a través de las capas. Para realizar el entrenamiento en una red profunda, N bloques se apilan en serie. Es importante definir el input original como $Z^{(0)}$, y las salidas de cada bloque sucesivo como $Z^{(n)}$ para $n = 1, \ldots, N$, donde cada $Z^{(n)}$ es la salida del bloque n, que se convierte en la entrada del siguiente bloque $n + 1$. De esta manera, la información pasa a través de varias capas de transformaciones no lineales, atención y normalización, refinándose progresivamente a medida que se apilan más bloques. Este apilamiento de bloques permite que el modelo capture representaciones cada vez más abstractas y complejas, lo que lo hace capaz de aprender relaciones a largo alcance dentro de los datos de entrada.

$$Z^{(n)} = \mathcal{T}(Z^{(n-1)}) \in \mathbb{R}^{\tau \times d_m}, \quad 1 \leq n \leq N$$

3.5. Generación de Texto

Una vez que los bloques de decoder han procesado la secuencia de entrada, el modelo está listo para predecir el siguiente token. Nuevamente se pasa por una capa de normalización (Layer Norm) para estabilizar el entrenamiento. A continuación, se aplica una capa denominada **Language Modelling Head** (LM-Head) que dependerá del tipo de modelo. Para este caso en el que se busca generar texto, LM-Head será una capa lineal que proyecta la representación del decoder al espacio del vocabulario. Esta proyección está definida de la siguiente manera:

$$L_{i,\cdot} = \text{LayerNorm}(Z^{(N)})_{i,\cdot} W_{proy} \in \mathbb{R}^{\mathbf{V}},$$

donde $W_{proy} \in \mathcal{L}(\mathcal{Z}, \mathbb{R}^{\mathbf{V}})$. A los resultados $L_{i,\cdot}$ de cada query los denominamos "logits", los cuales se utilizarán como criterio para obtener una distribución de probabilidad para el siguiente token, mientras que L es la matriz de logits después de procesar toda la secuencia de tokens.

Ahora, notemos que tenemos un vector de logits $L_{i,\cdot}$ para cada token de query t_i, donde $i = 0, \ldots, \tau - 1$. El vector de logits $L_{i,\cdot}$ tiene los valores para predecir el token t_{i+1} en la secuencia. Por lo tanto, para predecir el token t_τ, solamente necesitamos trabajar con $L_{\tau-1,\cdot}$, es decir, el correspondiente al último token de la secuencia, lo cual nos deja con la siguiente expresión para los logits de salida:

$$l_t = L_{\tau-1,t} = \text{LayerNorm}(Z^{(N)})_{\tau-1,\cdot} W_{proy_{\cdot,t}} = (z_{out})^T w_{proy_t}, \quad t \in \mathfrak{T},$$

donde l_t es el logit del token $w_t \in \mathcal{V}$ y donde z_{out} denota el vector $z_{\tau-1}^{(N)}$ normalizado con la capa de LayerNorm correspondiente. Por otro lado, analizar el resto de logits solamente es útil al momento de entrenar para poder comparar lo predicho vs lo real.

3.5.1. Vinculación de Pesos

La matriz de proyección W_{proy} puede ser aprendida durante el entrenamiento del Transformer; sin embargo, es posible aplicar un truco que mejora la eficiencia del modelo y reduce la cantidad de parámetros. Observamos que W_{proy}, también conocida como "embedding de salida" (output embedding), tiene la misma dimensión que la transposición de la matriz de embeddings, W_X^T, debido a que ambas proyectan las representaciones de dimensión d_m a un espacio de tamaño igual al del vocabulario $\mathbf{V}$.

El truco que se emplea es la **vinculación de pesos** (weight tying) propuesto por Press y Wolf (2017), una técnica en la que se comparten los pesos entre la capa de embedding de entrada W_X y la capa de proyección W_{proy}. De esta forma, en lugar de aprender dos matrices independientes, se fuerza a que ambas compartan los mismos parámetros, reduciendo significativamente la cantidad de parámetros del modelo.

Intuitivamente, si tenemos palabras semánticamente similares, esperaríamos que la probabilidad de que sean la siguiente palabra en la secuencia sea parecida y, por ende, que sus logits

sean cercanos. Al atar los pesos, el logit del token $t \in \mathcal{T}$ está dado por

$$l_t = (z_{out})^T x_t = \langle z_{out}, x_t \rangle .$$

Sean palabras $p, q \in \mathcal{V}$ tales que sus embeddings son cercanos, es decir $\|x_p - x_q\| \to 0$, entonces podemos verificar que sus logits también serán bastante parecidos:

$$\begin{aligned} \|x_p - x_q\| &\to 0 \\ x_p - x_q &\to \vec{0} \\ \langle z_{out}, x_p - x_q \rangle &\to 0 \\ \langle z_{out}, x_p \rangle - \langle z_{out}, x_q \rangle &\to 0 \\ l_p - l_p &\to 0 \end{aligned}$$

por lo tanto, esto sugiere que vincular los pesos dará buenos resultados. De acuerdo a Press y Wolf (2017), la vinculación de los embeddings de entrada y salida mejora la perplejidad de varios modelos lingüísticos; esto se traduce a una mayor capacidad predictiva y una mejor generalización del modelo.

3.5.2. Distribución de Probabilidad

La salida de la capa lineal nos da los logits con los cuales podremos obtener una distribución de probabilidad para el siguiente token en una secuencia. Para convertir estos logits en una distribución de probabilidad sobre los posibles tokens, se aplica nuevamente la función *Softmax*. Esta función toma los logits y los normaliza de manera que la suma de las probabilidades de todos los tokens posibles sea igual a 1, obteniendo la siguiente función de probabilidad:

$$P(t = t_\tau | t_0, t_1, \ldots, t_{\tau-1}) = \frac{\exp(l_t)}{\sum_{t' \in \mathcal{T}} \exp(l_{t'})}$$

Ahora que ya tenemos la probabilidad para cada token, el siguiente paso es escoger un elemento mediante muestreo. Lo más simple es seleccionar el token con la probabilidad más alta directamente; sin embargo, esto provocaría que siempre se obtenga el mismo resultado y

eso no simularía adecuadamente la manera de expresarse de un humano. Otra opción es escoger el siguiente token de acuerdo con la distribución de probabilidad obtenida, aunque existe la posibilidad de que tokens con una probabilidad significativamente baja sean seleccionados aleatoriamente, dando un resultado erróneo.

Una solución para esto es usar **top-k sampling**. Esta técnica consiste en escoger solo los k tokens con los logits más altos y muestrear a partir de ellos, donde $k \leq \mathbf{V} \in \mathbb{N}$ es un hiperparámetro. Para ello definimos el conjunto de los top-k tokens de la siguiente manera

$$\mathcal{T}_k = \{t \in \mathcal{T} : |\{t' \in \mathcal{T} : l_{t'} \geq l_t\}| \leq k\},$$

es decir, escogemos todos los tokens t tales que el conjunto de tokens con un logit mayor o igual al suyo contiene a lo sumo k elementos. A partir de este conjunto nuevamente aplicamos Softmax:

$$P_k(t = t_\tau | t_0, t_1, \ldots, t_{\tau-1}) = \frac{\exp(l_t)}{\sum\limits_{t' \in \mathcal{T}_k} \exp(l_{t'})}.$$

Notemos que esto es equivalente a redistribuir proporcionalmente las probabilidades originales, es decir, sea $p_t = P(t = t_\tau | t_0, t_1, \ldots, t_{\tau-1})$, entonces

$$\begin{aligned} P_k(t = t_\tau | t_0, t_1, \ldots, t_{\tau-1}) &= \frac{\exp(l_t)}{\sum\limits_{t' \in \mathcal{T}_k} \exp(l_{t'})} \\ &= \frac{\frac{1}{\sum_{t^* \in \mathcal{T}} \exp(l_{t^*})} \exp(l_t)}{\frac{1}{\sum_{t^* \in \mathcal{T}} \exp(l_{t^*})} \sum\limits_{t' \in \mathcal{T}_k} \exp(l_{t'})} \\ &= \frac{\frac{\exp(l_t)}{\sum_{t^* \in \mathcal{T}} \exp(l_{t^*})}}{\sum\limits_{t' \in \mathcal{T}_k} \frac{\exp(l_{t'})}{\sum_{t^* \in \mathcal{T}} \exp(l_{t^*})}} \\ &= \frac{p_t}{\sum\limits_{t' \in \mathcal{T}_k} p_{t'}} \end{aligned}$$

En la figura 3.13 podemos observar un ejemplo de top-k con $k = 5$ para las probabilidades de la

siguiente palabra para la frase “el perro persigue al”.

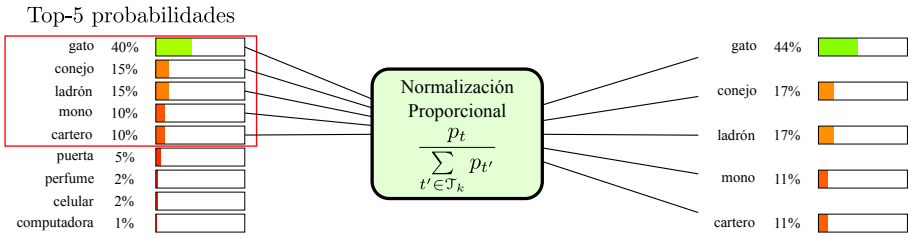

Figura 3.13: Ejemplo de top-k con $k = 5$.

Las predicciones top-k son bastante útiles para tareas como la generación de lenguaje natural o los sistemas de diálogo, mejorando el rendimiento de estas tareas al proporcionar una gama más amplia de posibles candidatos (Mao et al., 2024).

Por otro lado, no basta con solo muestrear con las probabilidades dadas por el modelo, sino que en ciertos casos es útil poder controlar qué tan aleatorio es el output del modelo. Para esto se usa el concepto de **temperatura** donde a cada logit se le divide para una temperatura λ, obteniendo la siguiente función de probabilidad:

$$P(t = t_\tau | t_0, t_1, \ldots, t_{\tau-1}) = \frac{\exp(l_t/\lambda)}{\sum_{t' \in \mathcal{T}'} \exp(l_{t'}/\lambda)}$$

donde $\mathcal{T}'$ es el conjunto de índices de los tokens sobre los cuales se está muestreando (por ejemplo, el conjunto de todos los tokens, el conjunto de los top-k tokens con mayor logit, o cualquier subconjunto de $\mathcal{T}$ excluyendo tokens que no se deseen predecir). De acuerdo con Renze y Guven (2024), una temperatura más baja hace que la salida del modelo sea más determinista, favoreciendo así las predicciones más probables, mientras que una temperatura más alta hace que el resultado sea más aleatorio, lo que favorece las predicciones más “creativas”. Esta creatividad se refleja en la disposición del modelo a explorar resultados menos convencionales y menos probables. Las temperaturas más altas pueden dar lugar a textos novedosos, ideas diversas y soluciones creativas a los problemas.

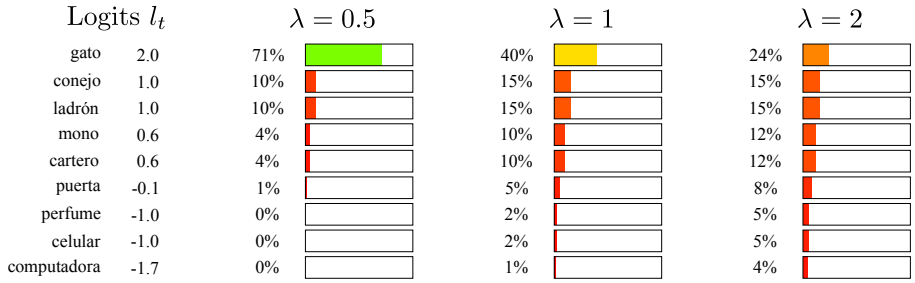

Figura 3.14: Ejemplo de muestreo con temperaturas $\lambda = 0.5, 1, 2$.

De esta forma, simplemente nos queda entrenar el modelo mediante entropía cruzada, buscando minimizar la ecuación

$$-\sum_{1\leq\tau\leq M}\sum_{j=0}^{J-\tau-1}\log P(t_{\tau+j}|t_j,\ldots,t_{\tau-1+j}),$$

donde J es la cantidad de tokens que contiene el corpus de texto para entrenamiento.

CAPÍTULO 4

IMPLEMENTACIÓN DE UN LLM CON EL MANUAL DEL ESTUDIANTE DE LA USFQ

Para poner en práctica lo visto en este trabajo, se implementaron modelos transformer sobre el texto del manual del estudiante de la Universidad San Francisco de Quito. A modo de experimentación, se implementaron 3 modelos para su respectiva comparación. El primer modelo parte de un transformer preentrenado y se hace un entrenamiento corto sobre el manual del estudiante para evitar sobreajuste. El segundo modelo parte de pesos inicializados aleatoriamente buscando demostrar que se necesita un corpus sumamente extenso para entrenar un transformer. Por último, el tercer modelo parte del primer modelo y se le entrena por varias épocas sobre el manual del estudiante para buscar sobreajustarlo y observar el comportamiento del modelo.

Para poner en práctica lo analizado en este trabajo, se implementaron tres modelos basados en arquitecturas Transformer aplicados al texto del manual del estudiante de la Universidad San Francisco de Quito. Estos modelos fueron diseñados con fines experimentales para evaluar su rendimiento en este contexto específico.

El primer modelo utiliza un Transformer preentrenado como punto de partida, al que se le realiza un entrenamiento breve sobre el manual, con el objetivo de evitar el sobreajuste y evaluar cómo el modelo adapta sus conocimientos al dominio del manual con una cantidad limitada de datos.

El segundo modelo se inicializa con pesos aleatorios y se entrena desde cero, con el propósito de investigar la necesidad de un corpus extenso para entrenar un modelo Transformer de manera efectiva. Este enfoque permite observar el impacto de la falta de un preentrenamiento sobre el

rendimiento del modelo en tareas relacionadas con el manual del estudiante.

Finalmente, el tercer modelo parte del primer modelo preentrenado y se somete a un entrenamiento prolongado sobre el manual durante varias épocas, con el objetivo de inducir un posible sobreajuste y examinar cómo este afecta la capacidad del modelo para generalizar en tareas de generación de texto dentro del dominio específico.

Para los modelos se utilizó una arquitectura tipo GPT-2-small (Radford et al., 2019) con los siguientes hiperparámetros:

Hiperparámetro	**Notación**	**Valor**
Longitud máxima de secuencia	M	1024
Tamaño del vocabulario	**V**	50257
Dimensión del modelo	d_m	768
Número de bloques	N	12
Número de cabezas de atención	h	12

Cuadro 4.1: Hiperparámetros utilizados para la implementación de los modelos Transformer.

La arquitectura del modelo tipo GPT-2-small incluye varias características clave. Se utilizan embeddings y codificaciones posicionales que son entrenados conjuntamente dentro del mismo modelo. Además, se incorpora un sesgo aprendible para todas las capas lineales, excepto para la capa lineal final, ya que esta está vinculada a la matriz de embeddings de entrada. Los sesgos se aplican en las transformaciones de los queries, keys y values, así como en la proyección final de las capas de atención multi-cabeza y en los perceptrones multicapa. Por otro lado, en la capa oculta de los perceptrones multicapa, sube la dimensionalidad a 4 veces, es decir, $4d_m = 3072$. Como resultado, el modelo cuenta con un total de 123.65 millones de parámetros.

Por otro lado, se utilizó el optimizador AdamW (Loshchilov & Hutter, 2019), con un decaimiento de pesos aplicado solo a los parámetros con dos o más dimensiones tensoriales, excluyendo los sesgos y los parámetros de LayerNorm. Los hiperparámetros del optimizador se indican en la tabla 4.2: Las tasas de aprendizaje variaron con respecto a cada modelo. Para el primer y tercer modelo se utilizó una tasa de aprendizaje constante de 1×10^{-5}. Para el segundo

Hiperparámetro	Notación	Valor
Tasa de decaimiento exponencial (primer momento)	β_1	0.9
Tasa de decaimiento exponencial (segundo momento)	β_2	0.95
Epsilon	ϵ	1×10^{-8}
Decaimiento de pesos	λ	0.1

Cuadro 4.2: Hiperparámetros del optimizador AdamW.

modelo se aplicó decaimiento del coseno (Becerra, 2024) durante 32 épocas sin periodo de calentamiento, iniciando con una tasa de aprendizaje de 1×10^{-4} y finalizando con una tasa de 1×10^{-6}.

El modelo preentrenado que se utilizó fue el modelo GPT2-Spanish de Oñate Latorre y Ortiz Fuentes (2021), el cual está disponible en Hugging Face. Todos los modelos se entrenaron en una laptop ASUS ROG Zephyrus G14 equipada con una GPU NVIDIA GeForce RTX 2060 y 16GB de RAM. Para el primer modelo se cargaron los pesos de GPT2-Spanish y se entrenó sobre el texto del manual del estudiante de la USFQ, tomando secuencias aleatorias de longitud $M = 1024$ en lotes de tamaño 16. Dado que la capacidad de la computadora era limitada, se optó por utilizar sublotes de tamaño 2 y se acumularon los gradientes por 8 iteraciones de modo que el resultado por época fuera equivalente al procesamiento de lotes de tamaño 16. Este modelo fue entrenado por 64 épocas, alcanzando un error medio de 3.2053 en las últimas 10 épocas, lo que indica que el modelo logró adaptarse de manera efectiva al dominio específico del manual del estudiante. A pesar de que no se utilizó un conjunto de validación, el comportamiento observado en el conjunto de entrenamiento (4.1) sugiere que el modelo no experimentó un sobreajuste significativo, ya que la tasa de descenso del error no se redujo a un ritmo anómalo en las últimas épocas. Esto indica que el preentrenamiento permitió que el modelo transfiriera conocimientos generales al dominio del manual, sin caer en la memorización de los datos específicos.

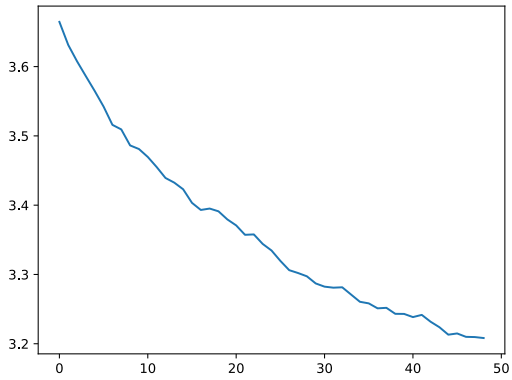

Figura 4.1: Resultado de entrenamiento del primer modelo (Pérdida en función de época).

En el segundo modelo, los pesos se inicializaron de manera aleatoria. Debido a que el modelo se entrena desde cero, se consideró un tamaño de lote significativamente mayor (256), acumulando el gradiente en lotes de tamaño 2 durante 128 iteraciones. Tras entrenar durante 64 épocas se observó que el error comenzó a estancarse, alcanzando un error medio de 8.9921 en las últimas 10 épocas. En la figura 4.2 se observa que mayor parte de la reducción del error ocurrió en las primeras épocas, mientras que en las siguientes el progreso fue mucho más lento o incluso nulo. Además, al generar texto con este modelo, se observó que solo predecía tokens comúnes en el corpus como “de” y “la”. Esto puede atribuirse a la falta de un preentrenamiento adecuado, que proporcionara una base sólida de conocimiento. Este comportamiento refuerza la hipótesis de que, para entrenar un modelo Transformer desde cero, se requiere un corpus considerablemente más grande y tiempo de entrenamiento prolongado, lo cual no fue posible debido a los recursos computacionales limitados.

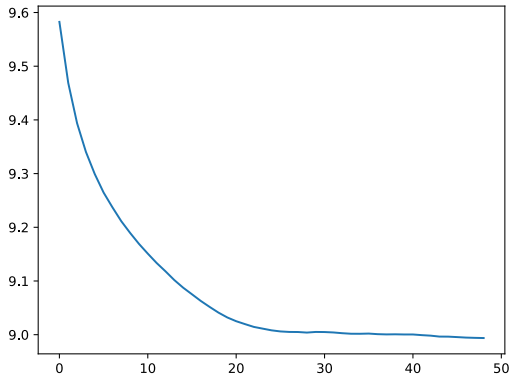

Figura 4.2: Resultado de entrenamiento del segundo modelo (Pérdida en función de época).

En el tercer modelo, que comenzó con los pesos del primer modelo y luego se entrenó durante varias épocas adicionales, todos los hiperparámetros se mantuvieron iguales con excepción del número de épocas que se elevó a 2048. Este modelo puede considerarse como una continuación del entrenamiento del primer modelo. En el entrenamiento se observó un comportamiento interesante; a partir de las primeras 5 épocas el ritmo de descenso en la pérdida se volvió notablemente mayor, indicando un rápido sobreajuste al conjunto de entrenamiento. Es probable que el modelo haya comenzado a memorizar patrones específicos del manual debido al entrenamiento prolongado sobre un corpus pequeño, lo que potencialmente afectó su capacidad de generalizar en situaciones fuera de este contexto; sin embargo, es útil al momento de trabajar directamente con texto relacionado al manual del estudiante. En este último modelo se obtuvo un error medio de 0.0867 en las últimas 10 épocas.

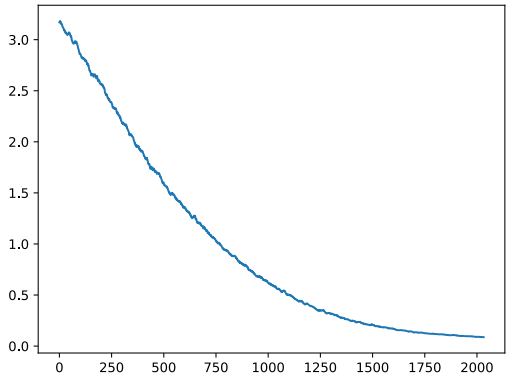

Figura 4.3: Resultado de entrenamiento del tercer modelo (Pérdida en función de época).

A continuación, se presentan ejemplos de generación de texto con los tres modelos entrenados, utilizando dos frases para completar: una relacionada con el manual del estudiante y otra fuera de este contexto. Los resultados muestran las diferencias clave en el comportamiento de cada modelo.

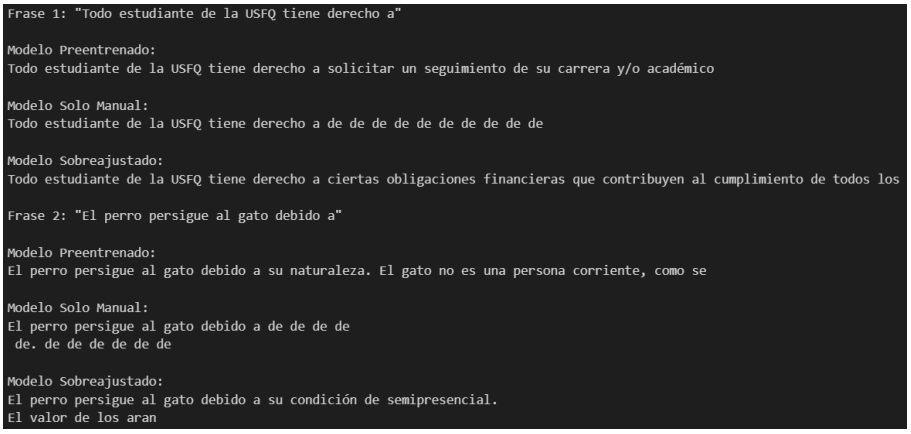

Figura 4.4: Resultados de generación de texto con los tres modelos.

En el primer modelo, que fue preentrenado y luego ajustado con el manual del estudiante, se observa que el modelo se adapta bien tanto a contextos relacionados con el manual como ajenos a este. Por ejemplo, al completar la primera frase, el modelo genera una continuación coherente: “Todo estudiante de la USFQ tiene derecho a solicitar un seguimiento de su carrera y/o académico”. Este resultado indica que el modelo ha aprendido a generar texto relevante para el contexto de la USFQ, sin perder fluidez.

El segundo modelo, entrenado desde cero, presenta dificultades en cuanto al contexto, como lo demuestra la continuación de ambas frases, que se convierte en una repetición constante de los tokens "de". Esto refleja la falta de conocimiento contextual y semántico debido a la falta de preentrenamiento en un corpus más extenso y diverso.

El tercer modelo, sobreajustado al manual del estudiante, muestra un comportamiento distinto al generar texto fuera de contexto. Al completar la primera frase, el modelo genera un texto muy relacionado con el contexto del manual: “Todo estudiante de la USFQ tiene derecho a ciertas

obligaciones financieras que contribuyen al cumplimiento de todos los". Este comportamiento revela que el modelo ha memorizado patrones específicos del manual; sin embargo, se puedeb obtener respuestas inesperadas al momento de salirse de este contexto.

Al analizar la segunda frase, el primer modelo continúa con "El perro persigue al gato debido a su naturaleza", lo que muestra una respuesta fluida y coherente en un contexto ajeno al manual. Por el contrario, el modelo sobreajustado genera una continuación como "El perro persigue al gato debido a su condición de semipresencial", lo que refleja que el modelo ha absorbido contenido muy específico y tiende a redirigir las respuestas hacia el dominio de este, aplicándolo de manera innecesaria fuera del contexto original.

CAPÍTULO 5

CONCLUSIONES Y TRABAJO FUTURO

Este trabajo ha proporcionado un análisis exhaustivo y riguroso de los modelos tipo transformer decoder-only, explorando tanto su base teórica como su aplicabilidad práctica. Se ha desglosado en detalle la estructura matemática de los modelos, abordando componentes clave como los embeddings, la tokenización, la codificación posicional y el mecanismo de atención. Estos elementos se integran de manera eficiente para permitir la generación de texto coherente y contextualizado. Los embeddings no solo representan palabras en un espacio vectorial, sino que también capturan relaciones semánticas complejas, mejorando la capacidad del modelo para entender y generar texto. La tokenización, a través de técnicas como Byte Pair Encoding (BPE), facilita el procesamiento de vocabularios amplios y la gestión de palabras desconocidas. Las codificaciones posicionales juegan un papel crucial al asegurar que el orden de las palabras se preserve sin interferir con su significado, mientras que el mecanismo de atención permite que el modelo se enfoque en las partes más relevantes del texto según el contexto. La formulación matemática de queries, keys y values permite seleccionar información pertinente de manera eficiente, lo que dota al modelo de flexibilidad y precisión en sus respuestas.

Un aporte significativo del trabajo ha sido la implementación práctica de los conceptos teóricos mediante el ajuste fino de un modelo preentrenado, adaptado para la generación de texto en un contexto específico. Este proceso ha demostrado cómo la teoría subyacente puede traducirse en aplicaciones prácticas, reafirmando la capacidad de los transformers para personalizarse según las necesidades del usuario. En particular, se ha validado que los mecanismos analizados matemáticamente, como las codificaciones posicionales y el mecanismo de atención, desempeñan un papel crucial en el rendimiento del modelo, asegurando que las palabras sean procesadas con un enfoque contextualizado y ordenado.

A pesar de los logros alcanzados, este estudio presenta algunas limitaciones. Por ejemplo, el tamaño del corpus utilizado para el ajuste fino podría restringir la capacidad del modelo para generalizar a contextos más amplios. Además, los recursos computacionales disponibles han condicionado la escala de los experimentos realizados, lo que podría limitar el alcance de las conclusiones obtenidas. Estas limitaciones ofrecen oportunidades para futuras investigaciones.

En este sentido, los trabajos futuros podrían centrarse en explorar el uso de modelos más grandes o con arquitecturas más avanzadas, lo que permitiría evaluar cómo la escala impacta el rendimiento en tareas de generación de texto. Asimismo, sería interesante realizar experimentos en diferentes dominios temáticos para analizar la adaptabilidad de los transformers a contextos especializados. Por último, investigar técnicas de ajuste fino más eficientes podría abrir nuevas posibilidades para optimizar modelos en entornos con recursos computacionales limitados.

En conclusión, este trabajo no solo ha aportado un análisis detallado del funcionamiento de los transformers, sino que también ha demostrado su capacidad práctica en la generación de texto. Al integrar teoría y práctica, se han sentado las bases para futuras exploraciones que amplíen el impacto de estos modelos en diversos ámbitos académicos y profesionales.

BIBLIOGRAFÍA

Axler, S. (2014, 5 de noviembre). *Linear algebra done right* (3.ª ed.). Springer. https://doi.org/10.1007/978-3-319-11080-6

Ba, J. L., Kiros, J. R., & Hinton, G. E. (2016). Layer Normalization. https://arxiv.org/abs/1607.06450

Becerra, K. (2024, abril). Learning rate: Cosine decay with warmup and hold period. https://www.linkedin.com/pulse/learning-rate-cosine-decay-warmup-hold-period-karel-becerra-fppye

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning* [http://www.deeplearningbook.org]. MIT Press.

Hendrycks, D., & Gimpel, K. (2023). Gaussian Error Linear Units (GELUs). https://arxiv.org/abs/1606.08415

Injosoft AB. (s.f.). *ASCII table - Table of ASCII codes, characters and symbols*. Consultado el 30 de noviembre de 2024, desde https://www.ascii-code.com/

Loshchilov, I., & Hutter, F. (2019). Decoupled Weight Decay Regularization. https://arxiv.org/abs/1711.05101

Mao, A., Mohri, M., & Zhong, Y. (2024). Top-k Classification and Cardinality-Aware Prediction. https://arxiv.org/abs/2403.19625

Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient Estimation of Word Representations in Vector Space. https://arxiv.org/abs/1301.3781

Mikolov, T., Sutskever, I., Chen, K., Corrado, G., & Dean, J. (2013). Distributed Representations of Words and Phrases and their Compositionality. https://arxiv.org/abs/1310.4546

NVIDIA Corporation. (s.f.). *CUDA C Programming Guide*. Consultado el 30 de noviembre de 2024, desde https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

Oñate Latorre, A., & Ortiz Fuentes, J. (2021). DeepESP/gpt2-spanish. https://huggingface.co/DeepESP/gpt2-spanish

OpenAI. (s.f.). *Tokenizer*. https://platform.openai.com/tokenizer

Press, O., & Wolf, L. (2017). Using the Output Embedding to Improve Language Models. https://arxiv.org/abs/1608.05859

Radford, A., & Narasimhan, K. (2018). Improving Language Understanding by Generative Pre-Training. https://api.semanticscholar.org/CorpusID:49313245

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., & Sutskever, I. (2019). Language Models are Unsupervised Multitask Learners. https://api.semanticscholar.org/CorpusID:160025533

Renze, M., & Guven, E. (2024). The Effect of Sampling Temperature on Problem Solving in Large Language Models. https://arxiv.org/abs/2402.05201

Sennrich, R., Haddow, B., & Birch, A. (2016). Neural Machine Translation of Rare Words with Subword Units. https://arxiv.org/abs/1508.07909

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2023). Attention Is All You Need. https://arxiv.org/abs/1706.03762