IMF Smart Education

MÁSTER EN

Big Data e Inteligencia Artificial

ONLINE

Implementación de una arquitectura Transformer educativa (NLP).

TFM elaborado por:

Diego García Muro

Tutor/a de TFM:

Daniel Rubio Yagüe

- Soria, 16 Septiembre del 2025 -

Resumen

Explicación de la arquitectura Transformer, importancia y lo que contiene el documento

Abstract

Lo mismo pero en inglés

Índice general

Capítulo 1

Introducción y antecedentes

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

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

1.1. De modelos secuenciales a mecanismos de atención

Antes de la aparición de esta arquitectura, los modelos existentes procesaban las palabras de forma secuencial con el fin de entender el lenguaje. Esto implica una complejidad [O(n²)] de forma que para procesar 3 palabras se tenían que ejecutar 6 operaciones secuenciales (para la tercera palabra necesita recordar la primera y segunda y procesar la tercera, para la segunda tendría que recordar la primera y procesar la segunda y para la primera se debía procesar esa palabra), esto, para 10 palabras aumenta a 45 operaciones, y para 100, 4950. Computacionalmente es muy ineficiente.

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

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

Todo este proceso no se ejecuta una única vez, sino que se apila en capas, de forma que cada capa o

cabeza de atención se encarga de una tarea distinta. Así, las capas más superficiales capturan relaciones sintácticas del tipo sujeto-verbo-objeto, mientras que las capas más profundas abordan el razonamiento abstracto y el procesamiento meta-cognitivo, de hecho, se cree que en las capas 70 a 80 de los modelos GPT la representación se asemeja al comportamiento humano (Plain English AI, 2021). En este punto, un elemento clave es el *flujo residual*. Gracias a él, toda la red comparte y acumula información y cada capa comparte sus aportaciones al resto.

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

Pese a los grandes avances y la repercusión que están teniendo los modelos LLM hoy en día, basados en este tipo de arquitectura, hay que decir que presentan varios problemas. Por un lado la atención es [$O(n^2)$], pues para 10.000 palabras hay 10.000 x 10.000 relaciones posibles, por otro lado, las infraestructuras son fijas y, además, todos los parámetros se activasn siempre, lo que es ineficiente. Es por ello que están surgiendo soluciones como la atención dispersa (el modelo no atiende a todas las posiciones), arquitecturas capaces de modificarse a sí mismas de forma autónoma y técnicas como *mixture of experts* que permiten usar una fracción de los parámetros en cada paso (Plain English AI, 2021).

Capítulo 2

Objetivos del proyecto

Implementar una arquitectura Transformer funcional que no busque competir con los modelo SOTA, sino comprenderla a bajo nivel y probarla con un corpus reducido.

Lo que se persigue en este proyecto es desarrollar un pequeño transformer capaz de generar texto en un contexto reducido como son las obras de Shakespeare. Con ello se consigue comprender la arquitectura, técnicas de tokenización y embedding (procesamiento de texto en general) y algoritmos de redes neuronales (Linear) y modelos NLP (LSTM).

Capítulo 3

Material y métodos

3.1. Arquitectura Transformer

Se va a implementar una arquitectura Transformer para la generación de texto en un contexto reducido; por ello, se desarrollará únicamente el Decoder, ya que no se necesita una entrada previa que transformar, como sería el caso de un traductor.

Este elemento consta de los siguientes módulos en su composición fundamental:

3.1.1. Positional encoding

Un transformer, a diferencia de otros modelos como los LSTM, no son secuenciales y procesan los datos en paralelo, esto hace que desconozcan en que orden aparecen los distintos tokens. Es por ello que se realiza esta codificación, que provee una posición relativa a cada token o palabra en la secuencia (Phillips, 2019). Es un factor importante pues en una frase, cada palabra depende del resto y según la posición en la que aparezcan pueden tener distintos significados: "Me senté en el banco.^{es} distinto a .^{El} banco de peces". En el paper .^{A}ttention is All You Need", se utiliza el seno o coseno para dar a cada posición una representación única, ya que cada palabra se representa con un vector numérico. Esto es debido a que la salida está normalizada, pues estos valores comprenden entre [-1 ,1] y no requiere de entrenamiento adicional pues son valores únicos.

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

3.1.2. Masked Multi-head Attention

En esta arquitectura, como se ha comentado, no se procesan secuencialmente los datos, sino que se hace en paralelo. Para ello se utilizan los llamados **mecanismos de atención**. Estos sopesan la importancia de distintos tokens en la secuencia de entrada. En sí, se puede decir que cada palabra mira al resto para ver cuáles tienen mayor importancia a la hora de entender el contexto. Por ejemplo, .^{El} perro ladra en el prado", la palabra "ladra"tendrá un peso mayor en "perro"que "prado". Lo que se hace es calcular múltiples de estos mecanismos en paralelo, de forma que cada uno aprende una proyección diferente (dependencias sintácticas, semánticas,etc.) y a continuación, se concatenan. En sí, el proceso de calcular esos valores de importancia requiere de 3 elementos: los vectores Q,K y V. Donde cada W asociada a esos elementos son una matriz de pesos entrenables (Analytics Vidhya, 2020).

Query (Q)

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

."El", este valor se calcula:

$$W_{El} * W_Q$$

Key (K)

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

$$W_{El} * W_K$$

Value (V)

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

$$W_{El} * W_V$$

El proceso de atención se puede comparar con la búsqueda de un video en YouTube. Esta plataforma almacena sus videos en un diccionario Key-Value, siendo la clave el nombre. Cuando se realiza una búsqueda (Query) se calcula la similitud con esas claves (Key) para devolver el resultado.
Como punto de partida se usa el vector de embeddings calculado previamente, y en base a este se calcularían los vectores anteriores, que se pueden entender como 3 versiones de dicho embedding. Para su cálculo, se aplican transformaciones lineales que encuentran la mejor combinación de pesos y tras esto se aplica la siguiente fórmula:

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

$$QK^T$$

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

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

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

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

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

3.1.3. Add & Norm

3.1.4. Feed Forward

3.1.5. Linear

3.1.6. Softmax

Capítulo 4

Resultados

4.1. Preparación del entorno

El proyecto se implementa íntegramente en Python 3.12. El primer punto es montar un entorno virtual donde se instalarán las librerías necesarias.

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

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

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

#Instalar torch para esa version de CUDA
pip install --pre torch torchvision torchaudio --index-url https://download.pyto

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

En este caso, la GPU utilizada es: NVIDIA GeForce RTX 4090 Laptop GPU con memoria total (MB): 15943

4.2. Análisis de los dataset

4.2.1. Tiny Shakespeare

Se trata de un dataset creado por Andrej Karpathy compuesto por 40.000 líneas con las obras de Shakespeare. El dataset está disponible en HuggingFace (HuggingFace, 2019a).

```
wget https://raw.githubusercontent.com/karpathy/char-rnn/master/data/tinyshakesp
```

4.2.2. WikiText2

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

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

```
First Citizen:
Before we proceed any further, hear me speak.

All:
Speak, speak.

First Citizen:
You are all resolved rather to die than to famish?

All:
Resolved. resolved.

First Citizen:
First, you know Caius Marcius is chief enemy to the people.
```

Figura 4.1: Fragmento del dataset Tiny Shakespeare

4.3. Limpieza y tokenización

Para que el modelo sea capaz de encontrar patrones y llegar a çomprenderün texto, es necesario estructurarlo previamente. En sí, el proceso de estructuración consiste en aplicar una serie de reglas al texto para normalizarlo, limpiarlo y extraer los llamados "tokens". Un token se puede entender como un fragmento de un texto; puede ser, por ejemplo, un carácter, una palabra o una subpalabra. Estos tokens se transforman a un valor numérico, con lo que ya se tendría una representación numérica del texto, pero para que el modelo lo comprenda, cada token se representa como un vector o tensor, son los llamados embeddings, que capturan el contexto de las palabras y otra información importante. (LMPO, 2020)

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

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

Figura 4.2: Limpieza antes de tokenizar

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

embeddings también demasiado grande. A esto hay que sumar que aparece el problema de palabras no vistas o ÜNK". Estos problemas se resuelven con la tokenización de subpalabras. Este método se basa en la idea de que las palabras que se usan con frecuencia no deberían dividirse, mientras que las más raras sí, manteniendo así palabras más frecuentes. Por ejemplo, en un texto en inglés, la palabra *.ªnnoyingly"* no es una palabra común, mientras que las subpalabras *.ªnnoying"* y *"ly"* sí. Permite, con ello, ver más vocabulario, sin perder significado de las palabras menos frecuentes (LMPO, 2020)

En este proyecto, dado que son textos en inglés y las limitaciones técnicas que existen, se ha decidido utilizar un tokenizador de subpalabras. Entre estos, destacan dos: BPE y WordPiece (HuggingFace, 2021). Ambos funcionan de forma similar, la principal diferencia radica en cómo realizan las üniones.ª partir del vocabulario base. BPE lo que hace es primeramente una pre-tokenización de los datos de entrenamiento por espacios o basada en reglas (por ejemplo, **Moses**). Con ello, obtiene un conjunto de palabras únicas y su frecuencia. Tras ello, crea un vocabulario base con los distintos símbolos que aparecen en las palabras únicas. La idea ahora es aprender reglas de unión para crear nuevos símbolos a partir de otros dos en base a la frecuencia. El proceso se repite hasta alcanzar el tamaño deseado, que se configura como hiperparámetro. **En GPT el tamaño es de 40478 tokens.** Para evitar un vocabulario base muy extenso si se quiere tener todos los caracteres con el fin de evitar palabras o tokens no vistos, se pueden usar bytes, así se tiene un tamaño fijo de 256, utilizando reglas para los caracteres de puntuación que no se puedan representar.

METER EJEMPLO

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

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

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

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

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

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

donde:

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

##################################

4.3.1. Implementación del tokenizador BPE

El primer paso es entrenar el tokenizador, de forma que aprenda el vocabulario y a dividir en subpalabras. Se establece el tipo de modelo, "bpe.[en] este caso, el tamaño del vocabulario final, la porción de caracteres del corpus que debe cubrir el modelo (1 para cubrir todos) y el número de hilos de CPU a usar durante el entrenamiento. Además, se utiliza el parámetro *byte_fallback* para controlar los caracteres desconocidos, tal y como se ha explicado previamente.Se aplica una normalización NFKC (descompone los caracteres en formas básicas y los recompone siguiendo una forma estándar) antes de entrenar donde se estandarizan caracteres y eliminan espacios innecesarios o múltiples. Esto es algo que conviene usar con lenguajes como el japonés que utilizan caracteres desnormalizados o textos sucios y desnormalizados; sin embargo, en este caso, los datasets empleados ya están listos para usarse y se podría evitar. Por último, se asignan los IDs especiales:

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

```
spm.SentencePieceTrainer.Train(
    input="./project/resources/datasets/shakespeare_clean_train.txt",
    model_prefix="./project/resources/models/bpe_model_shakespeare",
    vocab_size=16000,
    model_type="bpe",
    character_coverage=1.0,
    byte_fallback=True,
    normalization_rule_name="nfkc",
    remove_extra_whitespaces=True,
    num_threads=os.cpu_count(),
    pad_id=0, unk_id=1, bos_id=2, eos_id=3

)
```

Figura 4.3: Enter Caption

Una vez entrenado el tokenizador, se procede a tokenizar los conjuntos de datos (*train*, *validation* y *test*) utilizando dicho modelo. Para ello, se definen las siguientes funciones:

- `sp_encode_batch_train`: utilizada para tokenizar los datos de entrenamiento. Esta función habilita el muestreo probabilístico de segmentaciones (*subword regularization*) mediante los parámetros `enable_sampling=True`, `nbest_size=-1` y `alpha=0.1`. Esta técnica introduce pequeñas variaciones en las secuencias tokenizadas para un mismo texto, lo que actúa como regularización durante el entrenamiento del modelo de lenguaje.

```
def sp_encode_batch_train(batch):
    ids = [
        [sp.bos_id()] +
        sp.encode(t, out_type=int, enable_sampling=True, nbest_size=-1, alpha=0.1) +
        [sp.eos_id()]
        for t in batch["text"]
    ]
    attn = [[1]*len(x) for x in ids]
    return {"input_ids": ids, "attention_mask": attn}
```

Figura 4.4: Enter Caption

- `sp_encode_batch_eval`: usada para tokenizar los conjuntos de validación y prueba. Aquí se desactiva el muestreo para que la tokenización sea determinista y reproducible.

```
def sp_encode_batch_eval(batch):
    ids = [
        [sp.bos_id()] + sp.encode(t, out_type=int) + [sp.eos_id()]
        for t in batch["text"]
    ]
    attn = [[1]*len(x) for x in ids]
    return {"input_ids": ids, "attention_mask": attn}
```

Figura 4.5: Enter Caption

En ambos casos, a cada secuencia tokenizada se le añaden explícitamente los tokens especiales de inicio (<bos>) y fin (<eos>) de secuencia, cuyos IDs fueron definidos durante el entrenamiento del modelo SentencePiece. Además, se genera una máscara de atención (`attention_mask`) compuesta inicialmente por unos, ya que en este punto todavía no se ha aplicado *padding*.

Ejemplo: Considerando la frase:

`El perro ladra en el prado`

La función `sp.encode(...)` devuelve una secuencia de IDs correspondiente a las subpalabras identificadas por SentencePiece. Por ejemplo:

`[120, 457, 98, 14, 120, 892]`

donde los números representan los IDs de tokens correspondientes a subpalabras como:

`["_El", "_perro", "_ladra", "_en", "_el", "_prado"]`

Si los IDs de los tokens especiales son <bos>= 1 y <eos>= 2, entonces la secuencia final de entrada (`input_ids`) será:

`[1, 120, 457, 98, 14, 120, 892, 2]`

La correspondiente máscara de atención generada será:

`[1, 1, 1, 1, 1, 1, 1, 1]`

Agrupación en lotes y padding: Supongamos ahora que otra frase más corta, como:

`El perro ladra`

es tokenizada como:

`[1, 120, 457, 98, 2]`

con la máscara:

`[1, 1, 1, 1, 1]`

Al agrupar ambas secuencias en un mismo lote (*batch*), se aplica *padding* con un token especial (por ejemplo, <pad>= 3) para igualar su longitud. Las secuencias quedarían así:

- `input_ids`: `[1, 120, 457, 98, 14, 120, 892, 2]`
  `[1, 120, 457, 98, 2, 3, 3, 3]`

- `attention_mask`: `[1, 1, 1, 1, 1, 1, 1, 1]`
  `[1, 1, 1, 1, 1, 0, 0, 0]`

Esta máscara de atención es utilizada por el modelo Transformer para ignorar las posiciones de padding durante el cálculo de las atenciones.
##################################

4.4. Embeddings y Positional Encodding

Cualquier modelo basado en redes neuronales requiere que sus entradas sean valores numéricos, no arbitrarios, para poder trabajar con el lenguaje humano. Es cierto que los tokens del texto ya se han convertido a índices enteros usando técnicas de tokenización; no obstante, estos valores no son suficientes para que el modelo sea capaz de inferir relaciones semánticas entre las palabras y llegar a entender su significado.

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

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

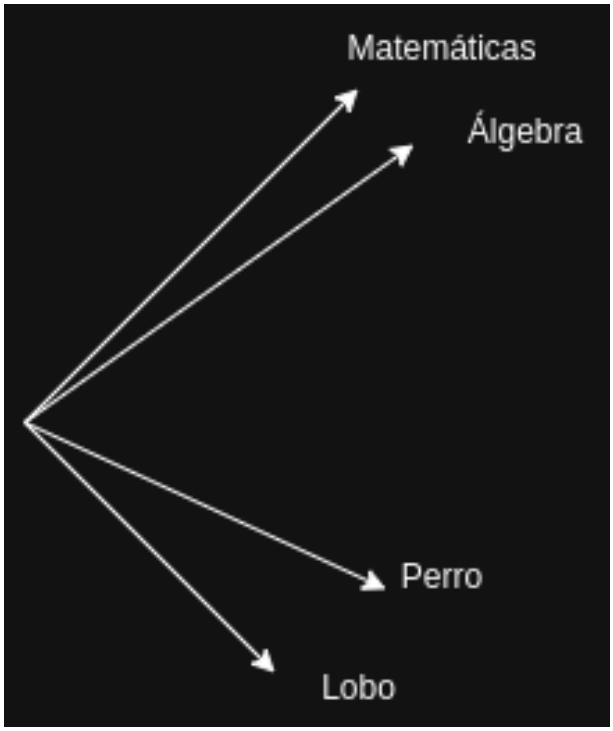

Figura 4.6: Ejemplo similitud vectorial embeddings

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

lote, lo que permite un aprendizaje eficiente.

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

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

En modelos avanzados como los Transformers, los embeddings se combinan con codificaciones posicionales para incorporar la información del orden de las palabras en la secuencia. Esta suma permite que el modelo tenga acceso tanto al significado individual de las palabras como a su posición relativa dentro de la frase. Así puede capturar diferencias según dónde aparezca la palabra, no es lo mismo ün banco de peces"que "dinero en el banco".(Pérez, 2020)

Esta matriz contiene información semántica de las palabras, pero dado que este tipo de arquitecturas procesan las secuencias en paralelo, falta información sobre la posición de dichas palabras. A continuación se presenta una solución a este problema, es lo que se conoce como Positional Encoding.

Implementación

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

Cabe destacar que inicialmente la matriz de embeddings se encuentra aleatorizada. A medida que se entrena el modelo, estos vectores se ajustan mediante retropropagación, lo que reduce la función de pérdida y permite que los embeddings capturen mejor la semántica, el contexto y otras relaciones relevantes entre los tokens. El resultado final es un tensor en el que cada fila representa un vector denso asociado a un token del vocabulario, adaptado a la tarea específica. (Bao, 2022)

4.5. Mecanismos de atención

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

Como se ha mencionado previamente, se basa en 3 matrices entrenables, inicializadas originalmente con valores aleatorios, y una operación, softmax, que calcula una distribución de probabilidad para cada token (ver section 3.1.2).

4.5.1. Implementación mecanismo de atención

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

El primer punto es crear las matrices de pesos

$$W_Q, W_K, W_V$$

. Aquí, PyTorch ofrece el módulo *nn.Linear*, que lo que hace es aplicar una transformación lineal (

$$y = xW^T + b$$

) a los datos de entrada (Popovic, 2023), incializando aleatoriamente una matriz de pesos y un vector de sesgos, el cual, en este caso, no se va a inicializar pues no interesa que los vectores se desplacen en el

espacio. Este módulo, toma dos parámetros: *in_features* y *out_features*, y la matriz resultante tiene un tamaño:

$$in\_features * out\_features$$

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

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

Figura 4.7: Proyección matrices Q, K, V

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

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

atencion = torch.softmax(metrica_atencion, dim=-1)

res_atencion = torch.matmul(atencion, V)
```

Figura 4.8: Implementación Self Attention

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

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

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

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

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

Figura 4.9: Implementación máscara causal

4.5.2. Implementación de las múltiples cabezas de atención

Para poder aplicar la atención múltiple las matrices originales Q, K, V se reorganizan en múltiples cabezas. Para ello, se emplea la función `view` de PyTorch, que permite reorganizar el tensor sin modificar

sus datos en memoria. En este caso, se pasa de una estructura de tamaño (B, T, d_{model}) a otra de tamaño (B, num_heads, T, d_k), donde $d_k = d_{model}/num\_heads$ es la dimensión de cada cabeza. Posteriormente, se aplica una transposición para situar la dimensión de las cabezas en la segunda posición:

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

Figura 4.10: Implementación Multi-Head Attention

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

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

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

Figura 4.11: Salida Multi-Head Attention

4.6. Normalización (Add & Norm)

Capítulo 5

Conclusiones

Referencias Bibliográficas

Analytics Vidhya. (2020). *Understanding Q, K, V in Transformer Self-Attention*. Medium. https://medium.com/analytics-vidhya/understanding-q-k-v-in-transformer-self-attention-9a5eddaa5960

AutoNLP. (2020). *Linked Wikitext-2*. AutoNLP. https://autonlp.ai/datasets/linked-wikitext-2

Bao, G. (2022). *How to use PyTorch's nn.Embedding: A comprehensive guide with examples*. Medium. https://medium.com/@garybao/how-to-use-pytorchs-nn-embedding-a-comprehensive-guide-with-examples-da00ea42e952

Epichka. (2023). *QKV in Transformers*. Epichka Blog. https://epichka.com/blog/2023/qkv-transformer/

GeeksforGeeks. (2021). *Word Embedding in PyTorch*. GeeksforGeeks. https://www.geeksforgeeks.org/deep-learning/word-embedding-in-pytorch/

Google. (2018). *SentencePiece*. GitHub. https://github.com/google/sentencepiece

HuggingFace. (2019a). *Tiny Shakespeare Dataset*. HuggingFace. https://huggingface.co/datasets/karpathy/tiny_shakespeare

HuggingFace. (2019b). *WikiText Dataset*. HuggingFace. https://huggingface.co/datasets/Salesforce/wikitext

HuggingFace. (2021). *Tokenizer Summary*. HuggingFace Docs. https://huggingface.co/docs/transformers/es/tokenizer_summary

LMPO. (2020). *From text to tokens: Understanding BPE, WordPiece and SentencePiece in NLP*. Medium. https://medium.com/@lmpo/from-text-to-tokens-understanding-bpe-wordpiece-and-sentencepiece-in-nlp-1367d9d610af

Neuraforge. (2023). *The Ultimate Guide to Preparing Text Data (Embeddings)*. Substack. https://neuraforge.substack.com/p/the-ultimate-guide-to-preparing-text

Pérez, J. A. (2020). *Embeddings en Transformers*. Universidad de Alicante. https://www.dlsi.ua.es/~japerez/materials/transformers/embeddings/

Phillips, H. J. (2019, agosto). *Positional encoding*. Medium. https://medium.com/@hunter-j-phillips/positional-encoding-7a93db4109e6

Plain English AI. (2021). *The 8 mathematical operations in GPT that accidentally created consciousness*. Medium. https://ai.plainenglish.io/the-8-mathematical-operations-in-gpt-that-accidentally-created-consciousness-ccaee58b241e

Popovic, M. (2023). *nn.Linear in PyTorch: Clearly Explained*. Kanaries Docs. https://docs.kanaries.net/topics/Python/nn-linear

Reddit. (2021). *SentencePiece, WordPiece, BPE: which tokenizer is best?* Reddit r/MachineLearning. https://www.reddit.com/r/MachineLearning/comments/rprmq3/d_sentencepiece_wordpiece_bpe_which_tokenizer_is/?tl=es-es

Shabbir, S. (2021). *Text Cleaning in NLP: Libraries, Techniques and Getting Started*. Medium. https://medium.com/@datascientist_SheezaShabbir/text-cleaning-in-nlp-libraries-techniques-and-how-to-get-started-8c7c7e8ba7cf

Swarms. (2021). *Understanding Masking in PyTorch for Attention Mechanisms*. Medium. https://medium.com/@swarms/understanding-masking-in-pytorch-for-attention-mechanisms-e725059fd49f

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention Is All You Need. *Advances in Neural Information Processing Systems*, 5998-6008. https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

Wolfe, C. R. (2025). *Scaling Laws for LLMs: From GPT-3 to o3*. Substack. https://cameronrwolfe.substack.com/p/llm-scaling-laws