perplexity

Procesamiento de Secuencias y Embeddings: El Puente entre Texto y Comprensión Máquina

El **procesamiento de secuencias y embeddings** representa la frontera donde el lenguaje humano se transforma en representaciones matemáticas que las máquinas pueden comprender y manipular. Este proceso no es simplemente una conversión técnica, sino una traducción sofisticada que preserva el significado, las relaciones semánticas y la estructura posicional del texto original, constituyendo el fundamento sobre el cual se construyen todos los sistemas modernos de procesamiento de lenguaje natural.

Tokenización: La Primera Transformación del Lenguaje

¿Qué es la Tokenización?

La **tokenización** constituye el paso inicial y fundamental en el procesamiento de lenguaje natural, funcionando como el puente que conecta el texto legible por humanos con las representaciones numéricas que pueden ser procesadas por las máquinas[1] [2]. Este proceso no es simplemente dividir texto en palabras, sino una transformación inteligente que preserva el significado mientras hace que la información sea accesible para algoritmos de aprendizaje automático.

Definición esencial: La tokenización es el proceso de convertir una secuencia de texto en una serie de tokens (unidades discretas) que pueden ser palabras completas, partes de palabras (subpalabras), o incluso caracteres individuales, dependiendo del nivel de granularidad requerido por la aplicación específica[3] [4].

Evolución de la Tokenización: De Espacios a Algoritmos Sofisticados

Tokenización Básica por Espacios

El método más simple de tokenización divide el texto usando espacios como delimitadores. Por ejemplo, la frase "Los chatbots son útiles" se convertiría en:

```
["Los", "chatbots", "son", "útiles"]
```

Sin embargo, este enfoque presenta limitaciones significativas:

- **Problemas con puntuación**: "¿Cómo estás?" se tokenizaría incorrectamente
- **Palabras compuestas**: No maneja eficientemente términos técnicos
- **Vocabularios grandes**: Requiere almacenar cada palabra única

- **Palabras desconocidas**: No puede manejar términos no vistos durante el entrenamiento[2] [5]

Tokenización a Nivel de Caracteres

Este método divide el texto en caracteres individuales:

```
"Hola" → ["H", "o", "l", "a"]
```

Ventajas:

- Vocabulario muy pequeño
- Puede manejar cualquier texto en el idioma
- No hay palabras fuera del vocabulario

Desventajas:

- Secuencias muy largas
- Pérdida de información semántica
- Mayor carga computacional[2]

Byte-Pair Encoding (BPE): La Revolución de la Subtokenización

¿Qué es BPE?

Byte-Pair Encoding (BPE) es un algoritmo de subtokenización que representa una evolución significativa en el procesamiento de texto, originalmente desarrollado como un algoritmo de compresión que posteriormente fue adaptado para el procesamiento de lenguaje natural[6] [7]. BPE resuelve elegantemente el problema del equilibrio entre el tamaño del vocabulario y la capacidad de manejar palabras desconocidas.

Funcionamiento del Algoritmo BPE

Fase 1: Construcción del Vocabulario (Token Learner)

Paso 1: Inicialización del Vocabulario
El proceso comienza con un vocabulario que contiene todos los caracteres individuales presentes en el corpus de entrenamiento[8] [9]:

```
Texto: "hug", "pug", "pun", "bun", "hugs"
Vocabulario inicial: ["b", "g", "h", "n", "p", "s", "u"]
```

Paso 2: Análisis de Frecuencias
El algoritmo cuenta la frecuencia de todos los pares de símbolos adyacentes en el corpus[10] [11]:

```
Pares frecuentes:
"hu": 2 veces
"ug": 2 veces
"pu": 2 veces
"un": 2 veces
```

Paso 3: Fusión Iterativa
Se selecciona el par más frecuente y se fusiona en un nuevo token. Este proceso se repite K veces hasta alcanzar el tamaño de vocabulario deseado[7] [8]:

```
Iteración 1: "hu" → "hu" (nuevo token)
Vocabulario: ["b", "g", "h", "n", "p", "s", "u", "hu"]

Iteración 2: "ug" → "ug" (nuevo token)
Vocabulario: ["b", "g", "h", "n", "p", "s", "u", "hu", "ug"]
```

Fase 2: Aplicación del Tokenizador (Token Segmenter)

Una vez entrenado el vocabulario, el tokenizador aplica las reglas de fusión aprendidas a nuevos textos[8] [9]:

```
Texto nuevo: "hugging"
Aplicación de reglas:
1. "h" + "u" → "hu"
2. "hu" + "g" → mantener separado
3. Resultado: ["hu", "g", "g", "i", "n", "g"]
```

Ejemplo Práctico Detallado de BPE

Consideremos el entrenamiento de un tokenizador BPE con el corpus "deep learning engineer" [12]:

Inicialización:

```
Tokens iniciales: ["d", "e", "e", "p", " ", "l", "e", "a", "r", "n", "i", "n", "g", " ",
```

Iteración 1: El par más frecuente es "e" + "e"

```
Fusión: "ee"
Resultado: ["d", "ee", "p", " ", "l", "e", "a", "r", "n", "i", "n", "g", " ", "e", "n", '
```

Iteración 2: Siguiente par más frecuente "d" + "ee"

```
Fusión: "dee"
Resultado: ["dee", "p", " ", "l", "e", "a", "r", "n", "i", "n", "g", " ", "e", "n", "g",
```

El proceso continúa hasta que el vocabulario alcanza el tamaño deseado[12].

Ventajas de BPE

1. Balance óptimo: Encuentra el equilibrio perfecto entre tamaño de vocabulario y capacidad expresiva[6] [7]

2. Manejo de palabras OOV: Puede representar cualquier palabra desconocida como secuencia de subtokens conocidos[7] [9]

3. Eficiencia computacional: Reduce significativamente el tamaño del vocabulario comparado con tokenización por palabras[11]

4. Preservación morfológica: Captura estructuras morfológicas comunes (prefijos, sufijos, raíces)[7]

5. Independencia del idioma: Funciona efectivamente en múltiples idiomas sin modificaciones[6]

Embeddings de Palabras (Word Embeddings): Capturando el Significado Semántico

¿Qué son los Word Embeddings?

Los **word embeddings** son representaciones vectoriales densas de palabras en un espacio multidimensional continuo, donde cada palabra se mapea a un vector numérico que captura su significado semántico y relaciones contextuales[13] [14] [15]. Esta técnica revolucionaria permite que las máquinas no solo procesen palabras como símbolos arbitrarios, sino que comprendan sus relaciones de significado.

Fundamentos Matemáticos de los Embeddings

Representación Vectorial

Cada palabra se representa como un vector en un espacio de alta dimensionalidad (típicamente 100-300 dimensiones para aplicaciones estándar, hasta 12,000+ en modelos avanzados)[13] [16]:

```
"rey" → [0.2, -0.1, 0.8, 0.3, ..., 0.5]  (vector de 300 dimensiones)
"reina" → [0.18, -0.08, 0.75, 0.28, ..., 0.48]
```

Propiedades Semánticas

La característica fundamental de los embeddings es que **palabras con significados similares tienen vectores similares en el espacio vectorial**[13] [15]:

```
distancia("gato", "perro") < distancia("gato", "automóvil")
```

Dimensión del Embedding (d_model)

La **dimensión del embedding**, comúnmente denominada `d_model` en la arquitectura Transformer, es un hiperparámetro crucial que determina el tamaño de la representación vectorial de cada token[16] [17]:

Configuraciones típicas:

- **Modelos pequeños**: d_model = 256 o 512
- **Modelos medianos**: d_model = 768 (BERT-base)
- **Modelos grandes**: d_model = 1024-4096
- **Modelos masivos**: d_model = 12,000+ (GPT-4, PaLM)[16] [18]

Impacto de d_model en el Rendimiento

Mayor dimensionalidad permite:

- Capturar matices semánticos más finos
- Representar relaciones lingüísticas más complejas
- Mejor rendimiento en tareas complejas

Costo computacional:

- Incremento cuadrático en parámetros de atención
- Mayor uso de memoria
- Entrenamiento más lento[16] [17]

Ejemplo Práctico: Construcción de Embeddings

Representación Simplificada

Consideremos un vocabulario pequeño con embeddings de 4 dimensiones[13]:

```
"guitarra": [0.3, 0.8, -0.1, 0.2]
"bajo":     [0.4, 0.7, -0.2, 0.3]
"batería":  [0.2, 0.9, -0.1, 0.1]
"piano":    [0.1, 0.6, -0.3, 0.4]
"coche":    [0.8, -0.1, 0.6, 0.9]
"bicicleta":[0.7, -0.2, 0.5, 0.8]
```

En este espacio, instrumentos musicales (`guitarra`, `bajo`, `batería`, `piano`) están agrupados cerca entre sí, mientras que medios de transporte (`coche`, `bicicleta`) forman otro grupo distinto[13].

Operaciones Semánticas

Los embeddings permiten realizar operaciones algebraicas que capturan relaciones semánticas[19] [20]:

```
# Analogía famosa: Rey - Hombre + Mujer ≈ Reina
vector_resultado = embedding("rey") - embedding("hombre") + embedding("mujer")
# vector_resultado debería estar cerca de embedding("reina")
```

Métodos de Generación de Embeddings

Word2Vec: El Pionero Moderno

Word2Vec es uno de los métodos más influyentes para generar embeddings, con dos arquitecturas principales[19] [21]:

CBOW (Continuous Bag of Words):

- Predice la palabra objetivo basándose en el contexto
- Entrada: palabras circundantes
- Salida: palabra central

Skip-gram:

- Predice las palabras del contexto basándose en la palabra objetivo
- Entrada: palabra central
- Salida: palabras circundantes[19]

Integración en Arquitecturas Modernas

En los modelos Transformer, los embeddings se aprenden durante el entrenamiento como parte de la primera capa de la red[22] [23]:

```
# Implementación conceptual
class TransformerEmbedding(nn.Module):
    def __init__(self, vocab_size, d_model):
        self.embedding = nn.Embedding(vocab_size, d_model)

    def forward(self, x):
        return self.embedding(x) * math.sqrt(d_model)
```

Embedding Posicional (Positional Encoding): Inyectando Orden en el Procesamiento Paralelo

El Problema de la Pérdida de Orden

La arquitectura Transformer, a diferencia de las redes recurrentes, procesa todos los tokens de una secuencia en paralelo. Esta capacidad de paralelización es una de sus grandes fortalezas, pero introduce un problema fundamental: **el modelo pierde inherentemente la información sobre el orden de los tokens**[24] [25] [26].

Ejemplo del problema:

```
"El gato persigue al ratón"
"El ratón persigue al gato"
```

Sin información posicional, un Transformer vería ambas oraciones como conjuntos idénticos de palabras, perdiendo el significado completamente diferente que transmite el orden[27].

¿Qué es el Positional Encoding?

El **positional encoding** es una técnica elegante que inyecta información sobre la posición de cada token en la secuencia sin aumentar la dimensionalidad de entrada[24] [25]. Esta información se suma a los embeddings de palabras, creando representaciones enriquecidas que contienen tanto significado semántico como información posicional[28] [26].

Implementación Matemática del Positional Encoding

Fórmulas Fundamentales

Los autores del paper original "Attention Is All You Need" propusieron usar funciones sinusoidales para generar las codificaciones posicionales[24] [25]:

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$

Donde:

- `pos`: posición del token en la secuencia (0, 1, 2, ...)
- `i`: índice de la dimensión en el vector de embedding (0, 1, 2, ..., d_model/2)
- `d_model`: dimensión del embedding
- **Dimensiones pares** (2i): usan función seno
- **Dimensiones impares** (2i+1): usan función coseno[24] [25] [29]

Ejemplo Práctico de Cálculo

Consideremos una secuencia con `d_model = 4` y calculemos el positional encoding para las primeras posiciones[25] [30]:

Posición 0 (primera palabra):

```
PE(0,0) = sin(0/10000^(0/4)) = sin(0) = 0
PE(0,1) = cos(0/10000^(0/4)) = cos(0) = 1
PE(0,2) = sin(0/10000^(2/4)) = sin(0) = 0
PE(0,3) = cos(0/10000^(2/4)) = cos(0) = 1
Vector: [0, 1, 0, 1]
```

Posición 1 (segunda palabra):

```
PE(1,0) = sin(1/10000^(0/4)) = sin(1) ≈ 0.841
PE(1,1) = cos(1/10000^(0/4)) = cos(1) ≈ 0.540
PE(1,2) = sin(1/10000^(2/4)) = sin(0.1) ≈ 0.100
PE(1,3) = cos(1/10000^(2/4)) = cos(0.1) ≈ 0.995
Vector: [0.841, 0.540, 0.100, 0.995]
```

Implementación Práctica en Código

Implementación en PyTorch

```
import torch
import torch.nn as nn
import math

class PositionalEncoding(nn.Module):
    def __init__(self, d_model, max_len=5000):
        super(PositionalEncoding, self).__init__()

        # Crear matriz de codificación posicional
        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len).unsqueeze(1).float()

        # Crear término divisor para diferentes frecuencias
        div_term = torch.exp(torch.arange(0, d_model, 2).float() *
                             -(math.log(10000.0) / d_model))

        # Aplicar seno a índices pares
        pe[:, 0::2] = torch.sin(position * div_term)
        # Aplicar coseno a índices impares
        pe[:, 1::2] = torch.cos(position * div_term)

        pe = pe.unsqueeze(0)  # Agregar dimensión de batch
        self.register_buffer('pe', pe)

    def forward(self, x):
```

```
        # Sumar positional encoding a embeddings
        return x + self.pe[:, :x.size(1)]
```

Propiedades Matemáticas del Positional Encoding Sinusoidal

Unicidad de Representaciones

Cada posición recibe una representación única debido a las diferentes frecuencias utilizadas en las funciones trigonométricas[24] [31]:

- **Posiciones cercanas**: Tienen representaciones similares
- **Posiciones lejanas**: Tienen representaciones distintivas
- **Patrones regulares**: Permiten al modelo aprender relaciones posicionales relativas[24]

Ventajas de las Funciones Sinusoidales

1. Rango normalizado: Los valores están siempre en [-1, 1][28]

2. Periodicidad: Permite al modelo generalizar a secuencias más largas que las vistas durante entrenamiento[24]

3. Propiedades de interpolación: Posiciones intermedias tienen representaciones intermedias[31]

4. Relaciones lineales: Permite cálculos eficientes de posiciones relativas[24]

Proceso de Integración: Sumando Embeddings y Posiciones

Combinación Aditiva

El positional encoding se **suma** (no concatena) con los word embeddings[24] [25]:

```
# Embedding de palabra: [0.2, 0.5, -0.1, 0.8]
# Positional encoding: [0.841, 0.540, 0.100, 0.995]
# Resultado final: [1.041, 1.040, 0.000, 1.795]

final_embedding = word_embedding + positional_encoding
```

Esta suma preserva la dimensionalidad mientras enriquece la representación con información posicional[32].

Ejemplo Visual del Proceso

```
Secuencia: "El gato duerme"

Paso 1 - Word Embeddings:
"El":     [0.1, 0.2, 0.3, 0.4]
"gato":   [0.5, 0.6, 0.7, 0.8]
```

```
"duerme": [0.9, 1.0, 1.1, 1.2]

Paso 2 - Positional Encodings:
Pos 0: [0.0, 1.0, 0.0, 1.0]
Pos 1: [0.841, 0.540, 0.100, 0.995]
Pos 2: [0.909, -0.416, 0.200, 0.980]

Paso 3 - Suma Final:
"El":     [0.1, 1.2, 0.3, 1.4]
"gato":   [1.341, 1.140, 0.800, 1.795]
"duerme": [1.809, 0.584, 1.300, 2.180]
```

Variantes Avanzadas de Positional Encoding

Embeddings Posicionales Aprendibles

Algunos modelos utilizan embeddings posicionales que se aprenden durante el entrenamiento en lugar de usar funciones fijas[33]:

```
class LearnedPositionalEncoding(nn.Module):
    def __init__(self, d_model, max_len):
        super().__init__()
        self.pos_embedding = nn.Embedding(max_len, d_model)

    def forward(self, x):
        positions = torch.arange(x.size(1), device=x.device)
        return x + self.pos_embedding(positions)
```

RoPE (Rotary Position Embedding)

RoPE es una técnica avanzada que codifica posiciones usando matrices de rotación, utilizada en modelos modernos como LLaMA[33] [31]:

- Preserva información posicional relativa y absoluta
- Se integra directamente en el mecanismo de atención
- Mejor rendimiento en secuencias largas[33]

Impacto en el Mecanismo de Atención

Influencia en las Puntuaciones de Atención

El positional encoding afecta directamente cómo el mecanismo de atención calcula las relaciones entre tokens[24] [27]:

```
# Sin positional encoding:
attention_score = query · key

# Con positional encoding:
```

```
query_pos = query + pos_query
key_pos = key + pos_key
attention_score = query_pos · key_pos
```

Esta modificación permite que el modelo considere tanto la similitud semántica como la proximidad posicional al decidir en qué tokens enfocarse[27].

Integración Completa: De Texto a Representación Enriquecida

Pipeline Completo de Procesamiento

El proceso completo de transformación de texto a representaciones procesables por Transformer involucra varios pasos coordinados:

Paso 1: Tokenización

```
"El gato duerme" → [101, 1945, 15722, 102]  (IDs de tokens)
```

Paso 2: Word Embeddings

```
[101, 1945, 15722, 102] → [
    [0.1, 0.2, 0.3, 0.4],     # Embedding de "El"
    [0.5, 0.6, 0.7, 0.8],     # Embedding de "gato"
    [0.9, 1.0, 1.1, 1.2],     # Embedding de "duerme"
    [0.2, 0.3, 0.4, 0.5]      # Embedding de token especial
]
```

Paso 3: Positional Encoding

```
Agregar información posicional a cada embedding
```

Paso 4: Entrada al Transformer

```
Matriz final lista para procesamiento por capas de atención
```

Consideraciones de Diseño en la Práctica

Escalado de Embeddings

En implementaciones reales, los embeddings se escalan por `√d_model` antes de agregar el positional encoding[18] [34]:

```
scaled_embeddings = embeddings * math.sqrt(d_model)
final_representation = scaled_embeddings + positional_encoding
```

Este escalado ayuda a mantener un balance adecuado entre la información semántica y posicional[34].

Manejo de Secuencias de Longitud Variable

Los modelos deben manejar eficientemente secuencias de diferentes longitudes:

```
# Truncar o pad secuencias al tamaño máximo
max_seq_length = 512
if seq_length > max_seq_length:
    sequence = sequence[:max_seq_length]
elif seq_length < max_seq_length:
    sequence = pad_sequence(sequence, max_seq_length)
```

Aplicaciones Prácticas y Casos de Uso

Modelos de Lenguaje Generativos

En modelos como GPT, todo el pipeline funciona coordinadamente para generar texto coherente:

1. **Tokenización BPE**: Convierte texto de entrada en tokens manejables
2. **Embeddings contextuales**: Cada token se representa con información semántica rica
3. **Positional encoding**: Mantiene el orden y estructura del texto
4. **Generación**: El modelo predice el siguiente token basándose en toda esta información enriquecida

Sistemas de Traducción Automática

Los Transformers de traducción utilizan estos componentes para:

- **Codificar** el texto fuente con embeddings y posiciones
- **Alinear** semánticamente palabras entre idiomas
- **Generar** traducciones que preservan tanto significado como estructura

Análisis de Sentimientos y Clasificación

Para tareas de clasificación, los embeddings enriquecidos con información posicional permiten:

- Detectar patrones dependientes del orden (ej: "no me gusta" vs "me gusta")
- Capturar matices semánticos complejos
- Considerar el contexto posicional de palabras clave

Desafíos y Limitaciones Actuales

Limitaciones de Longitud de Secuencia

Los modelos tradicionales están limitados por la longitud máxima de secuencia para la cual fueron entrenados. Secuencias más largas requieren:

- Técnicas de segmentación inteligente
- Métodos de positional encoding que se extrapolen eficientemente
- Arquitecturas que manejen ventanas de contexto deslizantes

Eficiencia Computacional

El procesamiento de embeddings de alta dimensionalidad conlleva:

- **Costo de memoria**: O(sequence_length × d_model)
- **Costo computacional**: O(sequence_length² × d_model) para atención
- **Necesidad de optimización**: Técnicas como attention sparse o local

Preservación Semántica en BPE

Aunque BPE es efectivo, puede fragmentar palabras de maneras que no preservan completamente el significado:

```
"unhappiness" → ["un", "hap", "pi", "ness"]
```

Esta fragmentación puede requerir que el modelo reconstituya el significado a partir de partes semánticamente inconexas.

El Futuro del Procesamiento de Secuencias

Tendencias Emergentes

1. Tokenización adaptativa: Algoritmos que ajustan dinámicamente la granularidad de tokenización según el contexto

2. Embeddings contextuales dinámicos: Representaciones que cambian según el contexto específico

3. Positional encodings más sofisticados: Técnicas como RoPE y ALiBi que manejan mejor secuencias largas

4. Eficiencia computacional: Métodos que reducen la complejidad sin sacrificar calidad

Integración Multimodal

Los desarrollos futuros buscan integrar:

- **Texto + imágenes**: Embeddings unificados para contenido multimodal
- **Texto + audio**: Procesamiento conjunto de señales de habla y texto
- **Texto + video**: Comprensión temporal y visual combinada

Conclusión: La Alquimia Digital del Significado

El **procesamiento de secuencias y embeddings** representa uno de los logros más elegantes de la inteligencia artificial moderna: la capacidad de transformar el lenguaje humano, con toda su complejidad, ambigüedad y riqueza, en representaciones matemáticas precisas que preservan no solo el significado, sino también las relaciones semánticas y la estructura posicional.

La **tokenización**, especialmente a través de algoritmos sofisticados como **BPE**, ha resuelto el antiguo dilema entre granularidad y eficiencia, permitiendo que los modelos manejen vocabularios prácticamente ilimitados mientras mantienen representaciones computacionalmente manejables[6] [7]. Este proceso no es meramente técnico, sino una forma de traducción que encuentra el equilibrio perfecto entre precisión lingüística y viabilidad computacional.

Los **word embeddings** han revolucionado nuestra comprensión de cómo las máquinas pueden capturar significado. Al representar palabras como vectores en espacios multidimensionales, estos sistemas no solo almacenan información léxica, sino que codifican relaciones semánticas complejas que permiten operaciones algebraicas con significado[13] [15]. La dimensión de estos embeddings (d_model) se ha convertido en un parámetro fundamental que equilibra capacidad expresiva con eficiencia computacional[16] [17].

El **positional encoding** resuelve elegantemente uno de los desafíos más fundamentales de la arquitectura Transformer: cómo preservar información temporal en un sistema inherentemente paralelo[24] [25]. Las funciones sinusoidales no son simplemente una solución técnica, sino una codificación matemática ingeniosaque permite a los modelos comprender tanto el "qué" como el "dónde" de cada elemento en una secuencia.

Juntos, estos componentes forman un pipeline de procesamiento que transforma texto crudo en representaciones ricas y multidimensionales que capturan:

- **Significado semántico** a través de embeddings densos
- **Relaciones morfológicas** mediante subtokenización inteligente
- **Estructura temporal** a través de codificación posicional
- **Contexto global** mediante representaciones vectoriales continuas

Esta transformación no es simplemente un preprocesamiento técnico, sino la base sobre la cual se construyen todas las capacidades modernas de comprensión y generación de lenguaje natural. Desde los chatbots que mantienen conversaciones coherentes hasta los sistemas de traducción que preservan matices culturales, desde los motores de búsqueda semántica hasta

los asistentes de programación que comprenden intención, todos dependen de esta alquimia digital fundamental.

El futuro promete refinamientos aún más sofisticados: tokenización adaptativa que se ajusta al contexto, embeddings que evolucionan dinámicamente, y técnicas de codificación posicional que manejan secuencias de longitud arbitraria. Sin embargo, los principios fundamentales que hemos explorado continuarán siendo la piedra angular sobre la cual se construirá la próxima generación de sistemas de inteligencia artificial.

En última instancia, el procesamiento de secuencias y embeddings representa la materialización de una aspiración humana milenaria: enseñar a las máquinas no solo a procesar símbolos, sino a comprender significado. Es el puente que conecta la precisión matemática de los algoritmos con la riqueza expresiva del lenguaje humano, permitiendo que la inteligencia artificial participe genuinamente en el acto más fundamentalmente humano: la comunicación con significado.

⁂

1. https://es.eitca.org/inteligencia-artificial/eitc-ai-tff-tensorflow-fundamentos/procesamiento-de-lenguaje-natural-con-tensorflow/secuenciación-convirtiendo-oraciones-en-datos/examen-revisión-secuenciación-convertir-oraciones-en-datos/¿Cuál-es-la-importancia-de-la-tokenización-en-el-preprocesamiento-de-texto-para-redes-neuronales-en-el-procesamiento-del-lenguaje-natural%3F/
2. https://www.datacamp.com/es/blog/what-is-tokenization
3. https://www.ultralytics.com/es/glossary/tokenization
4. https://www.hpe.com/es/es/what-is/nlp.html
5. https://es.linkedin.com/pulse/semana-18-procesamiento-de-lenguaje-natural-técnicas-clave-silva-o--e2r4e
6. https://huggingface.co/learn/llm-course/es/chapter6/5
7. https://codelabsacademy.com/es/blog/byte-pair-encoding-bpe-in-natural-language-processing-nlp/
8. https://www.toolify.ai/es/ai-news-es/aprende-sobre-el-potente-algoritmo-byte-pair-encoding-1123217
9. https://www.toolify.ai/es/ai-news-es/tokenizacin-de-byte-pair-encoding-mejora-tu-modelo-de-lenguaje-2029483
10. https://blog.ando.ai/posts/bpe-tokenizer/
11. https://www.geeksforgeeks.org/byte-pair-encoding-bpe-in-nlp/
12. https://mindfulml.vialabsdigital.com/post/tokenizacin-para-modelos-de-lenguaje/
13. https://datos.gob.es/en/blog/understanding-word-embeddings-how-machines-learn-meaning-words
14. https://datascientest.com/es/word-embedding
15. https://www.athos-cap.com/post-11-word-embeddings-el-corazon-de-la-revolucion-del-nlp/
16. https://www.guideglare.com/es/plataforma/chat-ia/tecnologia-chatbots/arquitectura-grandes-modelos-lenguaje
17. https://datascience.stackexchange.com/questions/93768/dimensions-of-transformer-dmodel-and-depth
18. https://github.com/hyunwoongko/transformer
19. https://datascientest.com/es/nlp-word-embedding-word2vec-es
20. https://somosnlp.org/nlp-de-cero-a-cien/sesion-01

21. https://es.wikipedia.org/wiki/Word_embedding
22. https://repositorio.comillas.edu/jspui/bitstream/11531/69691/2/TFG-Palomino Bravo, Marina.pdf
23. https://www.dlsi.ua.es/~japerez/materials/transformers/attention/
24. https://www.machinelearningmastery.com/a-gentle-introduction-to-positional-encoding-in-transformer-models-part-1/
25. https://www.geeksforgeeks.org/nlp/positional-encoding-in-transformers/
26. https://www.aprendemachinelearning.com/como-funcionan-los-transformers-espanol-nlp-gpt-bert/
27. https://www.athos-cap.com/positional-encoding-el-gps-de-los-modelos-transformer/
28. https://matematicas.etsiaab.upm.es/~md/bookIAA/08.01_Transformer_translate_training.html
29. https://datascience.stackexchange.com/questions/51065/what-is-the-positional-encoding-in-the-transformer-model
30. https://www.linkedin.com/pulse/positional-encoding-details-stephan-ao-ph-d--mzeve
31. https://huggingface.co/blog/designing-positional-encoding
32. https://uceltec.ucel.edu.ar/course/view.php?id=95&section=403
33. https://codelabsacademy.com/es/news/roformer-enhanced-transformer-with-rotary-position-embedding-2024-5-30/
34. https://blog.gopenai.com/transformer-from-scratch-in-tf-part-1-embedding-and-positional-encoding-de4bbd73b61f