TRANSFORMERS CON PYTORCH



Adrián Yared Armas de la Nuez

**Contenido**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

[**1. Objetivo 2**](#_2c35568x6wg8)

[**2. Configuración del entorno Flume 2**](#_mptzybqzhbp3)

[**2.1 Inicia el Servicio de Flume 3**](#_rjpj17ru25h3)

[**2.1.1 Comando 3**](#_e3mckfdyu347)

[**2.1.2 Ejecución 3**](#_ghbeig6bwub6)

[**2.2 Crear una Configuración para Flume 3**](#_60o0rlbxshg)

[**2.2.1 Comando 3**](#_fo2ppvz8bnfd)

[**2.2.2 Ejecución 4**](#_pz04p07f6ijg)

[**2.3 Preparar el Directorio Local de Entrada 4**](#_3mpvt9ru56ff)

[**2.3.1 Comando 4**](#_7hkh5uwzezf5)

[**2.3.2 Ejecución 5**](#_vrh4uydvhic3)

[**2.3.3 Comando 5**](#_qylvze508liu)

[**2.3.4 Ejecución 5**](#_l0cq6bllrkd3)

[**2.4 Ejecutar Apache Flume 5**](#_i989fvcpt8z5)

[**2.4.1 Comando 5**](#_2ehjy3gycynd)

[**2.4.2 Ejecución 5**](#_1t73ms8n1dlc)

[**2.4.3 Comando 6**](#_pfweesl1t30a)

[**2.4.4 Ejecución 6**](#_qvaqs3xr9yle)

[**2.4.5 Comando 6**](#_aig92oe7k2m0)

[**2.4.6 Ejecución 6**](#_e9uqlcl56mu)

## 

## 

#### 

## **1. Enunciado**

En los siguientes enlaces puedes ver la explicación e implementación de un transformer desde cero.

<https://youtu.be/onJRVqQMU6U> Parte 1/3 ( Ver a partir de 1h y 10 minutos) <https://youtu.be/V99DjxuHgHk> Parte 2/3

<https://youtu.be/tBEqpqwDw-A> Parte 3/3

El código fuente pueden descargarlo de:

<https://github.com/omarespejel/Hugging-Face-101-ES/blob/3b94611acdb9d1dc36df645d5f84213fbf8083cb/0_Introducci%C3%B3n_a_los_Transformers.ipynb>

## **2. Implementación práctica**

### **2.1 Enunciado**

Implementa un modelo Transformer sencillo desde cero utilizando un marco de programación de tu elección (por ejemplo, PyTorch o TensorFlow). Explica las decisiones de diseño tomadas durante la implementación, incluyendo:

* Elección de hiperparámetros
* Estructura de la red
* Funciones de activación utilizadas

### **2.2 Resolución**

LayerNorm normaliza las entradas a lo largo de la dimensión de características para mejorar la estabilidad del entrenamiento.

SublayerConnection aplica esta normalización antes de ejecutar una subcapa, luego añade la salida de esta subcapa (con dropout) a la entrada original mediante una conexión residual, lo que facilita el aprendizaje en redes profundas.

import torch

import torch.nn as nn

class LayerNorm(nn.Module):

"""

Implementa la normalización por capas (LayerNorm).

Normaliza cada muestra en la dimensión de características para estabilizar el entrenamiento.

"""

def \_\_init\_\_(self, features, eps=1e-6):

super().\_\_init\_\_()

self.gamma = nn.Parameter(torch.ones(features)) # Escala aprendible

self.beta = nn.Parameter(torch.zeros(features)) # Desplazamiento aprendible

self.eps = eps # Valor pequeño para evitar división por cero

def forward(self, x):

mean = x.mean(-1, keepdim=True) # Media en features

std = x.std(-1, keepdim=True) # Desviación estándar en features

# Normalización y escalado + desplazamiento

return self.gamma \* (x - mean) / (std + self.eps) + self.beta

class SublayerConnection(nn.Module):

"""

Implementa una conexión residual con normalización previa y dropout.

Se usa para facilitar el flujo de gradiente y estabilizar subcapas.

"""

def \_\_init\_\_(self, size, dropout):

super().\_\_init\_\_()

self.norm = LayerNorm(size)

self.dropout = nn.Dropout(dropout)

def forward(self, x, sublayer):

# Normaliza x, aplica la subcapa con dropout y añade la entrada x (residual)

return x + self.dropout(sublayer(self.norm(x)))

print("Clases LayerNorm y SublayerConnection definidas.")

Esta clase implementa la atención multi-cabeza, que divide la representación del modelo en varias "cabezas" para capturar diferentes aspectos de la información simultáneamente. Calcula la atención escalada por producto punto para cada cabeza, aplica máscara si es necesario, concatena los resultados y proyecta la salida final para integrarla en el modelo.

import torch

import torch.nn as nn

import math

import copy

class MultiHeadAttention(nn.Module):

"""

Implementa la atención multi-cabeza como en "Attention is All You Need".

Divide las entradas en múltiples cabezas para capturar distintas representaciones.

"""

def \_\_init\_\_(self, h, d\_model, dropout=0.1):

super().\_\_init\_\_()

assert d\_model % h == 0, "d\_model debe ser divisible por el número de cabezas h"

self.d\_k = d\_model // h # Dimensión por cabeza

self.h = h

# Cuatro capas lineales: para Q, K, V y salida final

self.linears = nn.ModuleList([copy.deepcopy(nn.Linear(d\_model, d\_model)) for \_ in range(4)])

self.attn = None # Para almacenar pesos de atención (útil para análisis)

self.dropout = nn.Dropout(p=dropout)

def attention(self, query, key, value, mask=None, dropout=None):

"""

Calcula atención escalada por producto punto.

query, key, value: (batch, h, seq\_len, d\_k)

mask: para evitar atención a ciertos elementos

"""

d\_k = query.size(-1)

scores = torch.matmul(query, key.transpose(-2, -1)) / math.sqrt(d\_k) # Producto punto escalado

if mask is not None:

scores = scores.masked\_fill(mask == 0, float('-inf')) # Enmascara con valor muy negativo

p\_attn = torch.softmax(scores, dim=-1) # Pesos de atención

if dropout is not None:

p\_attn = dropout(p\_attn)

return torch.matmul(p\_attn, value), p\_attn

def forward(self, query, key, value, mask=None):

"""

Ejecuta la atención multi-cabeza:

1) Proyecta Q, K, V y divide en cabezas.

2) Calcula atención en paralelo.

3) Concatenar y proyectar resultado final.

"""

if mask is not None:

mask = mask.unsqueeze(1) # Aplica máscara a todas las cabezas

nbatches = query.size(0)

# Proyección lineal y reshape para multi-cabeza

query, key, value = [

lin(x).view(nbatches, -1, self.h, self.d\_k).transpose(1, 2)

for lin, x in zip(self.linears[:3], (query, key, value))

]

x, self.attn = self.attention(query, key, value, mask=mask, dropout=self.dropout)

# Concatenar cabezas y aplicar proyección final

x = x.transpose(1, 2).contiguous().view(nbatches, -1, self.h \* self.d\_k)

return self.linears[-1](x)

print("Clase MultiHeadAttention definida.")

Esta clase implementa una red feed-forward aplicada posición por posición dentro de la secuencia. Primero expande la dimensión con una capa lineal, luego aplica ReLU y dropout, y finalmente reduce la dimensión de nuevo para mantener la forma original. Esto añade capacidad de modelado no lineal para cada token de forma independiente.

import torch

import torch.nn as nn

class PositionwiseFeedForward(nn.Module):

"""

Red feed-forward aplicada de forma independiente a cada posición

en la secuencia, usando dos capas lineales con activación ReLU.

"""

def \_\_init\_\_(self, d\_model, d\_ff, dropout=0.1):

super().\_\_init\_\_()

self.w\_1 = nn.Linear(d\_model, d\_ff) # Capa de expansión dimensional

self.w\_2 = nn.Linear(d\_ff, d\_model) # Capa para regresar a d\_model

self.dropout = nn.Dropout(dropout)

self.activation = nn.ReLU()

def forward(self, x):

# Aplica w\_1, luego ReLU, dropout y finalmente w\_2

return self.w\_2(self.dropout(self.activation(self.w\_1(x))))

print("Clase PositionwiseFeedForward definida.")

Embeddings convierte índices discretos de tokens en vectores densos, escalándolos para mantener la estabilidad. PositionalEncoding añade información de posición fija y continua a esos vectores usando funciones sinusoidales, lo que permite al modelo distinguir la posición relativa en la secuencia sin usar recursión ni convoluciones.

import torch

import torch.nn as nn

import math

class Embeddings(nn.Module):

"""

Capa de embeddings para convertir índices de tokens en vectores densos.

Multiplica la salida por sqrt(d\_model) para estabilizar las magnitudes.

"""

def \_\_init\_\_(self, d\_model, vocab):

super().\_\_init\_\_()

self.lut = nn.Embedding(vocab, d\_model)

self.d\_model = d\_model

def forward(self, x):

# x: (batch, seq) índices de tokens

return self.lut(x) \* math.sqrt(self.d\_model)

class PositionalEncoding(nn.Module):

"""

Codificación posicional sinusoidal para incorporar información de posición.

Añade vectores fijos basados en senos y cosenos para cada posición.

"""

def \_\_init\_\_(self, d\_model, dropout=0.1, max\_len=5000):

super().\_\_init\_\_()

self.dropout = nn.Dropout(p=dropout)

# Crear matriz de codificación posicional (max\_len x d\_model)

pe = torch.zeros(max\_len, d\_model)

position = torch.arange(0, max\_len, dtype=torch.float).unsqueeze(1) # (max\_len, 1)

div\_term = torch.exp(torch.arange(0, d\_model, 2).float() \* (-math.log(10000.0) / d\_model))

# Codificación sinusoidal: senos en posiciones pares, cosenos en impares

pe[:, 0::2] = torch.sin(position \* div\_term)

pe[:, 1::2] = torch.cos(position \* div\_term)

pe = pe.unsqueeze(0) # Añade dimensión batch: (1, max\_len, d\_model)

self.register\_buffer('pe', pe) # No es parámetro entrenable

def forward(self, x):

# x: (batch, seq, d\_model)

# Suma codificación posicional (recortada al tamaño de la secuencia) a x

x = x + self.pe[:, :x.size(1)].requires\_grad\_(False)

return self.dropout(x)

print("Clases Embeddings y PositionalEncoding definidas.")

EncoderLayer contiene una subcapa de auto-atención seguida de una feed-forward, ambas con conexiones residuales y normalización para facilitar el aprendizaje.

DecoderLayer incluye una auto-atención enmascarada para evitar "mirar adelante", una atención cruzada sobre la salida del encoder para integrar información de la entrada, y una red feed-forward, todo con conexiones residuales.

import torch

import torch.nn as nn

import copy

class EncoderLayer(nn.Module):

"""

Capa del encoder que combina auto-atención y red feed-forward,

cada una con conexión residual y normalización.

"""

def \_\_init\_\_(self, size, self\_attn, feed\_forward, dropout):

super().\_\_init\_\_()

self.self\_attn = self\_attn # Auto-atención sobre la misma entrada

self.feed\_forward = feed\_forward # Red feed-forward

# Dos conexiones residuales con LayerNorm y dropout

self.sublayer = nn.ModuleList([copy.deepcopy(SublayerConnection(size, dropout)) for \_ in range(2)])

self.size = size

def forward(self, x, mask):

# 1) Auto-atención con máscara

x = self.sublayer[0](x, lambda x: self.self\_attn(x, x, x, mask))

# 2) Feed-forward

return self.sublayer[1](x, self.feed\_forward)

class DecoderLayer(nn.Module):

"""

Capa del decoder con:

- auto-atención enmascarada para la salida previa,

- atención sobre la salida del encoder,

- red feed-forward.

Cada subcapa usa conexión residual y normalización.

"""

def \_\_init\_\_(self, size, self\_attn, src\_attn, feed\_forward, dropout):

super().\_\_init\_\_()

self.size = size

self.self\_attn = self\_attn # Auto-atención enmascarada

self.src\_attn = src\_attn # Atención cruzada con salida del encoder

self.feed\_forward = feed\_forward

# Tres conexiones residuales

self.sublayer = nn.ModuleList([copy.deepcopy(SublayerConnection(size, dropout)) for \_ in range(3)])

def forward(self, x, memory, src\_mask, tgt\_mask):

"""

x: salida previa del decoder con embedding y codificación posicional

memory: salida del encoder (memoria)

src\_mask: máscara para la secuencia de entrada (padding)

tgt\_mask: máscara para la secuencia de salida (padding + evitar atención futura)

"""

m = memory

# 1) Auto-atención enmascarada (solo posiciones anteriores)

x = self.sublayer[0](x, lambda x: self.self\_attn(x, x, x, tgt\_mask))

# 2) Atención cruzada con salida del encoder

x = self.sublayer[1](x, lambda x: self.src\_attn(x, m, m, src\_mask))

# 3) Feed-forward final

return self.sublayer[2](x, self.feed\_forward)

print("Clases EncoderLayer y DecoderLayer definidas.")

Estas clases implementan la estructura central del Transformer: un encoder y un decoder compuestos cada uno por N capas idénticas apiladas. Cada capa se aplica secuencialmente, y finalmente se normaliza la salida para estabilizar el entrenamiento.

import torch

import torch.nn as nn

import copy

class Encoder(nn.Module):

"""

Encoder compuesto por N capas idénticas (EncoderLayer).

Aplica normalización LayerNorm al final.

"""

def \_\_init\_\_(self, layer, N):

super().\_\_init\_\_()

# Clona la capa EncoderLayer N veces

self.layers = nn.ModuleList([copy.deepcopy(layer) for \_ in range(N)])

# Normalización final después de todas las capas

self.norm = LayerNorm(layer.size)

def forward(self, x, mask):

# Pasa x y máscara por cada capa

for layer in self.layers:

x = layer(x, mask)

# Normaliza la salida final

return self.norm(x)

class Decoder(nn.Module):

"""

Decoder compuesto por N capas idénticas (DecoderLayer).

Aplica normalización LayerNorm al final.

"""

def \_\_init\_\_(self, layer, N):

super().\_\_init\_\_()

# Clona la capa DecoderLayer N veces

self.layers = nn.ModuleList([copy.deepcopy(layer) for \_ in range(N)])

# Normalización final después de todas las capas

self.norm = LayerNorm(layer.size)

def forward(self, x, memory, src\_mask, tgt\_mask):

# Pasa x, memoria y máscaras por cada capa del decoder

for layer in self.layers:

x = layer(x, memory, src\_mask, tgt\_mask)

# Normaliza la salida final

return self.norm(x)

print("Clases Encoder y Decoder definidas.")

La clase Generator convierte las representaciones internas del Transformer en distribuciones de probabilidad sobre el vocabulario, usando log-softmax para entrenamiento con pérdidas como NLLLoss.

La clase Transformer orquesta el flujo completo: primero codifica la secuencia fuente con embeddings y el encoder, luego decodifica la secuencia objetivo con embeddings, la salida del encoder y máscaras, y finalmente prepara la salida para generación.

import torch

import torch.nn as nn

class Generator(nn.Module):

"""

Capa final que proyecta la salida del Transformer a la dimensión del vocabulario

y aplica log-softmax para producir probabilidades logarítmicas.

"""

def \_\_init\_\_(self, d\_model, vocab):

super().\_\_init\_\_()

self.proj = nn.Linear(d\_model, vocab) # Proyección lineal a vocab size

def forward(self, x):

# x: (Batch, Seq, d\_model)

# Salida: log-probabilidades (Batch, Seq, vocab)

return torch.log\_softmax(self.proj(x), dim=-1)

class Transformer(nn.Module):

"""

Modelo Transformer completo que conecta encoder, decoder, embeddings y generación.

"""

def \_\_init\_\_(self, encoder, decoder, src\_embed, tgt\_embed, generator):

super().\_\_init\_\_()

self.encoder = encoder # Encoder apilado

self.decoder = decoder # Decoder apilado

self.src\_embed = src\_embed # Embedding + codificación posicional para entrada

self.tgt\_embed = tgt\_embed # Embedding + codificación posicional para salida

self.generator = generator # Capa final para generación de logits/probabilidades

def forward(self, src, tgt, src\_mask, tgt\_mask):

# 1) Codificar la secuencia de entrada

memory = self.encode(src, src\_mask)

# 2) Decodificar con la salida previa y la memoria del encoder

return self.decode(memory, src\_mask, tgt, tgt\_mask)

def encode(self, src, src\_mask):

# Embedding + codificación posicional para src y pasar por encoder

return self.encoder(self.src\_embed(src), src\_mask)

def decode(self, memory, src\_mask, tgt, tgt\_mask):

# Embedding + codificación posicional para tgt y pasar por decoder

return self.decoder(self.tgt\_embed(tgt), memory, src\_mask, tgt\_mask)

print("Clases Generator y Transformer definidas.")

La función make\_model crea un Transformer completo con sus capas de atención, feed-forward y codificación posicional, clonándolas según el número de capas indicado. Inicializa sus pesos para un entrenamiento estable. Luego, se generan datos de ejemplo con padding y máscaras para evitar atención indebida a tokens irrelevantes o futuros. Se hace un pase hacia adelante para verificar que la arquitectura funciona y produce salidas con las dimensiones esperadas. Así tienes un modelo listo para entrenar o probar.

def make\_model(src\_vocab, tgt\_vocab, N=6, d\_model=512, d\_ff=2048, h=8, dropout=0.1):

"""

Construye un Transformer completo con hiperparámetros definidos.

"""

c = copy.deepcopy

# 1. Crear componentes reutilizables: multi-cabeza, feed-forward y codificación posicional

attn = MultiHeadAttention(h, d\_model, dropout)

ff = PositionwiseFeedForward(d\_model, d\_ff, dropout)

position = PositionalEncoding(d\_model, dropout)

# 2. Instanciar el Transformer completo con encoder, decoder, embeddings y generador

model = Transformer(

Encoder(EncoderLayer(d\_model, c(attn), c(ff), dropout), N),

Decoder(DecoderLayer(d\_model, c(attn), c(attn), c(ff), dropout), N),

nn.Sequential(Embeddings(d\_model, src\_vocab), c(position)), # Embeddings + PE para entrada

nn.Sequential(Embeddings(d\_model, tgt\_vocab), c(position)), # Embeddings + PE para salida

Generator(d\_model, tgt\_vocab)

)

# 3. Inicializar pesos con Xavier para mejor estabilidad al entrenar

for p in model.parameters():

if p.dim() > 1:

nn.init.xavier\_uniform\_(p)

return model

# --- Ejemplo rápido con datos dummy para prueba --- #

SRC\_VOCAB\_SIZE = 1000

TGT\_VOCAB\_SIZE = 1200

N\_LAYERS = 2 # Capas reducidas para velocidad

D\_MODEL = 128 # Dimensión del modelo reducida

D\_FF = 256 # Dimensión feed-forward reducida

NUM\_HEADS = 4 # Número de cabezas reducido

DROPOUT = 0.1

model = make\_model(SRC\_VOCAB\_SIZE, TGT\_VOCAB\_SIZE, N=N\_LAYERS, d\_model=D\_MODEL, d\_ff=D\_FF, h=NUM\_HEADS, dropout=DROPOUT)

print(f"Modelo Transformer creado con {N\_LAYERS} capas, d\_model={D\_MODEL}, {NUM\_HEADS} cabezas.")

# Crear datos de prueba con padding

src\_dummy = torch.randint(1, SRC\_VOCAB\_SIZE, (2, 10))

src\_dummy[0, 7:] = 0 # Padding en primera secuencia

tgt\_dummy = torch.randint(1, TGT\_VOCAB\_SIZE, (2, 8))

tgt\_dummy[1, 6:] = 0 # Padding en segunda secuencia

tgt\_dummy\_input = tgt\_dummy[:, :-1] # Entrada al decoder (teacher forcing)

# Máscara para padding en src (True donde NO es padding)

src\_mask\_dummy = (src\_dummy != 0).unsqueeze(1)

# Máscara para padding en tgt input

tgt\_mask\_padding = (tgt\_dummy\_input != 0).unsqueeze(1)

# Máscara para evitar atención a futuros tokens

def subsequent\_mask(size):

mask = torch.triu(torch.ones(1, size, size), diagonal=1).type(torch.uint8)

return mask == 0

tgt\_mask\_future = subsequent\_mask(tgt\_dummy\_input.size(-1))

# Combinar máscaras padding y futura para el decoder

tgt\_mask\_dummy = tgt\_mask\_padding & tgt\_mask\_future

# Forward pass sin entrenamiento

model.eval()

with torch.no\_grad():

output = model(src\_dummy, tgt\_dummy\_input, src\_mask\_dummy, tgt\_mask\_dummy)

final\_output\_log\_probs = model.generator(output)

# Mostrar formas de tensores para verificar

print(f"\nForma entrada src: {src\_dummy.shape}")

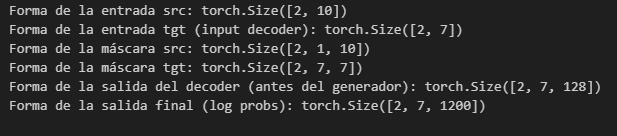
print(f"Forma entrada tgt (decoder input): {tgt\_dummy\_input.shape}")

print(f"Forma máscara src: {src\_mask\_dummy.shape}")

print(f"Forma máscara tgt: {tgt\_mask\_dummy.shape}")

print(f"Forma salida decoder (antes generador): {output.shape}")

print(f"Forma salida final (log probs): {final\_output\_log\_probs.shape}")



## **3. Comparación de modelos**

### **3.1 Enunciado**

Compara los modelos Transformer con los modelos tradicionales de procesamiento del lenguaje natural (NLP), como las redes neuronales recurrentes (RNNs) y las redes neuronales convolucionales (CNNs). Identifica las principales ventajas y desventajas de los Transformers en comparación con estos modelos más antiguos.

### **3.2 Resolución**

Antes, las RNNs y CNNs dominaban el procesamiento de lenguaje natural. Las RNNs leen las palabras una por una, manteniendo memoria de lo anterior, mientras que las CNNs detectan patrones locales en grupos de palabras.

Los Transformers cambiaron el juego porque:

* Pueden procesar toda la secuencia al mismo tiempo, lo que acelera mucho el entrenamiento.
* Capturan mejor las relaciones entre palabras, incluso si están muy separadas en el texto.
* Generan representaciones más contextuales y precisas para cada palabra.

Sin embargo, tienen sus retos:

* Su costo computacional crece rápido con textos muy largos, haciendo difícil procesar secuencias largas.
* Requieren muchos datos y recursos para entrenar modelos grandes.
* Necesitan añadir información sobre el orden de las palabras porque no lo manejan de forma natural.

En general, gracias a sus ventajas en velocidad y calidad, los Transformers son la tecnología favorita para la mayoría de tareas modernas en NLP.

## **4. Atención y su impacto**

### **4.1 Enunciado**

Explica el mecanismo de atención en los modelos Transformer y su importancia para el procesamiento del lenguaje natural.

Proporciona un ejemplo práctico de cómo el mecanismo de atención ayuda a mejorar el rendimiento en una tarea específica de NLP.

### **4.2 Resolución**

El mecanismo de atención es el corazón que impulsa a los modelos Transformer, una arquitectura revolucionaria en el procesamiento del lenguaje natural (NLP). Básicamente, la atención permite que el modelo **"preste atención" a diferentes partes de una oración o texto**, asignando distintos pesos o importancia a cada palabra según el contexto.

En lugar de procesar las palabras una a una o en secuencia estricta, la atención evalúa todas las palabras simultáneamente y determina cuáles son relevantes para entender el significado o generar una respuesta. Así, el modelo puede captar relaciones a largo plazo, como cuándo una palabra depende de otra que está muy lejos en la oración, algo que los modelos anteriores tenían dificultades para manejar.

### **Importancia en el Procesamiento del Lenguaje Natural**

Esta capacidad de enfocar selectivamente en distintas partes del texto es crucial porque el lenguaje es complejo y ambiguo. Por ejemplo, una palabra puede tener diferentes significados según el contexto, y la atención ayuda al modelo a decidir cuál es el sentido correcto basándose en el resto de la oración o el párrafo.

Gracias a la atención, los Transformers pueden entender mejor la semántica y la sintaxis, lo que se traduce en un rendimiento superior en tareas como traducción automática, resumen de textos, análisis de sentimientos, entre otros.

### **Ejemplo Práctico: Traducción Automática**

Imagina que queremos traducir la frase en inglés:  
 *"The cat that chased the mouse was hungry."*

Aquí, el mecanismo de atención permite que el modelo identifique que la palabra "that" conecta al gato con la acción de perseguir al ratón, y que "was hungry" se refiere al gato, no al ratón. El modelo puede "mirar" todas las palabras a la vez y asignar más peso a las que están relacionadas directamente con cada parte de la frase durante la traducción.

Esto evita errores comunes, como traducir mal relaciones gramaticales o confundir sujetos y objetos. Como resultado, la traducción es mucho más precisa y natural

## **5. Exploración de modelos preentrenados**

### **5.1 Enunciado**

Investiga un modelo Transformer preentrenado, como BERT, GPT o T5.

* Describe cómo se entrenó el modelo.
* Indica cómo ajustarlo (fine-tune) para tareas específicas.
* Discute cómo los modelos preentrenados han impactado en el campo de NLP.
* Proporciona ejemplos de aplicaciones prácticas.

### **5.2 Resolución**

Instala en segundo plano varias librerías clave para tareas de NLP y computación numérica, asegurando la versión específica de numpy. Luego confirma que la instalación terminó.

# Instalación silenciosa de librerías necesarias para procesamiento de lenguaje natural y manejo de datos

!pip install transformers datasets evaluate torch accelerate -q

!pip install numpy==1.23.5 -q

print("Librerías instaladas.")

Carga un modelo y tokenizador pre-entrenados, prepara un pequeño conjunto de datos de texto con etiquetas, lo tokeniza para que sea compatible con modelos de Hugging Face, y lo convierte en tensores para entrenar o evaluar un modelo de clasificación de texto.

from transformers import AutoTokenizer, AutoModelForSequenceClassification, Trainer, TrainingArguments

from datasets import Dataset

import torch

import numpy as np

import evaluate

# Definir modelo pre-entrenado (ligero y rápido)

model\_name = "distilbert-base-uncased"

# Cargar tokenizador asociado al modelo

tokenizer = AutoTokenizer.from\_pretrained(model\_name)

# Datos de ejemplo: textos y etiquetas (1 = positivo, 0 = negativo)

texts = [

"I love this movie, it's fantastic!",

"This was the worst film I have ever seen.",

"The acting was okay, but the plot was boring.",

"Absolutely brilliant, a must-watch!",

"I didn't like it very much.",

"A masterpiece of cinema."

]

labels = [1, 0, 0, 1, 0, 1]

# Crear dataset de Hugging Face a partir del diccionario

data = {"text": texts, "label": labels}

dataset = Dataset.from\_dict(data)

# División simple en entrenamiento (70%) y evaluación (30%)

dataset = dataset.train\_test\_split(test\_size=0.3)

print("Dataset de ejemplo:")

print(dataset)

# Función para tokenizar textos con padding y truncamiento a longitud máxima 128

def tokenize\_function(examples):

return tokenizer(

examples["text"],

padding="max\_length",

truncation=True,

max\_length=128

)

# Tokenizar dataset de forma batched para eficiencia

tokenized\_datasets = dataset.map(tokenize\_function, batched=True)

# Limpiar dataset: eliminar columna original y renombrar etiquetas para compatibilidad con Trainer

tokenized\_datasets = tokenized\_datasets.remove\_columns(["text"])

tokenized\_datasets = tokenized\_datasets.rename\_column("label", "labels")

# Establecer formato para PyTorch

tokenized\_datasets.set\_format("torch")

print("\nDataset tokenizado:")

print(tokenized\_datasets)

print("\nEjemplo de entrada tokenizada:")

print(tokenized\_datasets["train"][0])

Carga un modelo pre-entrenado de Hugging Face adaptado para clasificación binaria, configurado para distinguir entre dos clases. Luego confirma que el modelo se cargó correctamente con el número esperado de etiquetas.

# Cargar modelo pre-entrenado para clasificación de secuencias con 2 etiquetas (positivo/negativo)

model = AutoModelForSequenceClassification.from\_pretrained(model\_name, num\_labels=2)

print(f"Modelo {model\_name} cargado para clasificación con {model.config.num\_labels} etiquetas.")

Carga la métrica de accuracy para evaluar el modelo, define una función para calcularla durante la evaluación y configura los parámetros para entrenar el modelo, incluyendo número de épocas, tamaño de batch y directorios para guardar resultados y logs.

import numpy as np

import evaluate

from transformers import TrainingArguments

# Cargar métrica de evaluación (accuracy para clasificación)

metric = evaluate.load("accuracy")

def compute\_metrics(eval\_pred):

logits, labels = eval\_pred

# Obtener predicciones: índice con mayor valor en logits

predictions = np.argmax(logits, axis=-1)

# Calcular accuracy entre predicciones y etiquetas reales

return metric.compute(predictions=predictions, references=labels)

# Configurar argumentos para el entrenamiento del modelo

training\_args = TrainingArguments(

output\_dir="./results\_classification", # Carpeta para guardar checkpoints y resultados

evaluation\_strategy="epoch", # Evaluar después de cada época

num\_train\_epochs=3, # Número de épocas de entrenamiento

per\_device\_train\_batch\_size=2, # Tamaño de batch para entrenamiento

per\_device\_eval\_batch\_size=2, # Tamaño de batch para evaluación

warmup\_steps=1, # Pasos de calentamiento del learning rate

weight\_decay=0.01, # Regularización para evitar overfitting

logging\_dir="./logs\_classification", # Carpeta para logs de entrenamiento

logging\_steps=1, # Frecuencia de logs (cada paso)

# load\_best\_model\_at\_end=True, # (Opcional) cargar mejor modelo al final

# push\_to\_hub=False, # (Opcional) subir modelo a Hugging Face Hub

)

print("Argumentos de entrenamiento definidos.")

Configura un objeto Trainer con el modelo, datasets y parámetros definidos, inicia el entrenamiento (fine-tuning), y luego evalúa el modelo entrenado mostrando los resultados finales.

from transformers import Trainer

# Crear el objeto Trainer con modelo, datos y configuración

trainer = Trainer(

model=model, # Modelo a entrenar

args=training\_args, # Argumentos para el entrenamiento

train\_dataset=tokenized\_datasets["train"], # Dataset para entrenamiento

eval\_dataset=tokenized\_datasets["test"], # Dataset para evaluación

compute\_metrics=compute\_metrics, # Función para calcular métricas durante evaluación

tokenizer=tokenizer # Tokenizador (útil para manejo de padding dinámico)

)

print("Trainer creado. Iniciando fine-tuning...")

# Ejecutar fine-tuning del modelo con los datos proporcionados

trainer.train()

print("\nFine-tuning completado.")

# Evaluar el modelo entrenado en el dataset de evaluación

eval\_results = trainer.evaluate()

print("\nResultados de la evaluación final:")

print(eval\_results)

Guarda el modelo y tokenizador fine-tuned, luego los carga para hacer inferencias sobre nuevos textos, clasificándolos como positivos o negativos y mostrando los resultados.

import torch

from transformers import AutoTokenizer, AutoModelForSequenceClassification

# Guardar modelo y tokenizador entrenados

save\_directory = "./fine\_tuned\_distilbert\_classifier"

print(f"\nGuardando modelo en {save\_directory}...")

trainer.save\_model(save\_directory) # Guarda modelo y tokenizador si se pasó al Trainer

print("Modelo guardado.")

# --- Cargar modelo y tokenizador para inferencia ---

print("\nCargando modelo guardado para inferencia...")

loaded\_tokenizer = AutoTokenizer.from\_pretrained(save\_directory)

loaded\_model = AutoModelForSequenceClassification.from\_pretrained(save\_directory)

# Configurar dispositivo (GPU si está disponible, sino CPU)

device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

loaded\_model.to(device)

loaded\_model.eval() # Modo evaluación para desactivar dropout, etc.

# Función auxiliar para clasificar texto nuevo

def classify\_text(text):

print(f"\nClasificando nuevo texto: '{text}'")

inputs = loaded\_tokenizer(text, return\_tensors="pt", padding=True, truncation=True, max\_length=128)

inputs = {k: v.to(device) for k, v in inputs.items()} # Mover a GPU/CPU

with torch.no\_grad():

outputs = loaded\_model(\*\*inputs)

logits = outputs.logits

predicted\_class\_id = torch.argmax(logits, dim=-1).item()

predicted\_label = "Positivo" if predicted\_class\_id == 1 else "Negativo"

print(f"Predicción: {predicted\_label} (ID: {predicted\_class\_id})")

# Clasificar ejemplos nuevos

classify\_text("This movie was incredibly moving and well-acted.")

classify\_text("A complete waste of time and money.")

## **6. Limitaciones y consideraciones éticas**

### **6.1 Enunciado**

Identifica algunas de las limitaciones de los modelos Transformer en términos de sesgo, consumo de recursos y dependencia de grandes cantidades de datos. Discute las consideraciones éticas asociadas con su uso en aplicaciones reales.

### **6.2 Resolución**

**imitaciones de los Modelos Transformer**

-Sesgo en los Modelos

Los Transformers aprenden de grandes cantidades de texto que provienen de internet, libros y otros medios escritos por humanos. Esto implica que heredan los sesgos presentes en esos datos: prejuicios de género, raza, ideologías y estereotipos pueden quedar reflejados en sus respuestas o decisiones. Esto puede reforzar desigualdades sociales y propagar información errónea o discriminatoria.

-Consumo de Recursos Computacionales

Estos modelos son extremadamente poderosos, pero también muy costosos en términos de energía, memoria y tiempo de entrenamiento. Entrenar un Transformer grande requiere enormes cantidades de recursos, lo que implica un impacto ambiental significativo debido al consumo energético de los centros de datos. Además, no todos tienen acceso a esta tecnología, lo que puede aumentar la brecha digital.

-Dependencia de Grandes Cantidades de Datos

Para funcionar bien, los Transformers necesitan ingentes volúmenes de datos etiquetados o sin etiquetar. Esto no solo implica costos elevados, sino que también limita su aplicación en dominios donde la información es escasa o privada, como medicina o áreas especializadas. Además, el exceso de datos sin control puede introducir ruido o información irrelevante.

**Consideraciones Éticas en el Uso de Transformers**

Transparencia y Explicabilidad: Es fundamental que los usuarios entiendan cómo y por qué un modelo tomó cierta decisión, pero los Transformers son “cajas negras” difíciles de interpretar, lo que complica la rendición de cuentas.

-Equidad y Justicia: Usar modelos con sesgos en sistemas de selección de personal, justicia o financiamiento puede llevar a decisiones injustas y discriminatorias, afectando vidas reales.

-Privacidad: Al entrenar con datos masivos, muchas veces sin un control riguroso, existe el riesgo de que información sensible o privada se filtre o sea mal utilizada.

-Impacto Ambiental: Las organizaciones deben considerar el costo ecológico y buscar maneras de optimizar y hacer más sustentable la tecnología.

En resumen, aunque los Transformers abren puertas increíbles para la inteligencia artificial, es vital usarlos con responsabilidad, siempre atentos a minimizar sus limitaciones y garantizar un impacto positivo en la sociedad.

## **7. Aplicaciones en Big Data**

### **7.1 Enunciado**

Propón un caso de uso de un modelo Transformer en el ámbito de Big Data. Describe cómo utilizarías este modelo para analizar datos masivos, como datos de redes sociales, datos financieros o datos médicos. Explica cómo abordarías los desafíos asociados con este tipo de datos

### **7.2 Resolución**

Análisis de Sentimientos en Redes Sociales con Transformers

En el contexto de Big Data, las redes sociales generan grandes volúmenes de texto que reflejan la opinión pública sobre una marca. Utilizando un modelo Transformer, como BERT, podemos analizar estos datos masivos para entender el sentimiento detrás de cada mensaje, incluso cuando el lenguaje es coloquial o ambiguo.

El proceso incluye recolectar y limpiar datos, ajustar el modelo con ejemplos etiquetados y aplicar el análisis en tiempo real para detectar tendencias y posibles crisis. Los Transformers, gracias a su capacidad para captar contexto, mejoran la precisión en la interpretación.

Los principales desafíos son manejar la gran cantidad y velocidad de datos, reducir el ruido y sesgos, proteger la privacidad y optimizar recursos computacionales. Para enfrentarlos, se usan infraestructuras escalables, técnicas de preprocesamiento y controles éticos que garantizan un análisis eficiente y responsable.

Así, la empresa obtiene información valiosa y actualizada que le permite reaccionar rápido y mejorar su relación con los clientes.

## **8. Optimización y eficiencia**

### **8.1 Enunciado**

Los modelos Transformer pueden ser costosos en términos de recursos computacionales. Investiga técnicas de optimización y eficiencia para estos modelos, como el uso de Transformers ligeros o técnicas de pruning y quantization. Proporciona ejemplos de cómo estas técnicas pueden mejorar la eficiencia sin sacrificar el rendimiento.

### **8.2 Resolución**

Los modelos Transformer, aunque muy potentes, suelen ser costosos en recursos computacionales, lo que limita su uso en dispositivos con poca capacidad o en aplicaciones que requieren respuesta rápida. Para superar esto, se han desarrollado varias técnicas que permiten reducir su tamaño y velocidad sin perder mucho rendimiento.

**Técnicas Principales**

-Transformers Ligeros (Lightweight Transformers)

Estos modelos están diseñados desde cero para ser más eficientes, usando arquitecturas simplificadas o menos parámetros. Un ejemplo es DistilBERT, que reduce el tamaño de BERT en un 40% y es hasta un 60% más rápido, manteniendo más del 95% del rendimiento en tareas de NLP.

-Pruning (Poda)

Esta técnica consiste en eliminar conexiones o neuronas que aportan poco al modelo, reduciendo su complejidad. Por ejemplo, al podar un Transformer, se pueden eliminar hasta un 30-50% de parámetros, acelerando la inferencia y disminuyendo el uso de memoria, sin una caída significativa en la precisión.

-Quantization (Cuantización)

Aquí, los pesos del modelo se representan con números de menor precisión (por ejemplo, de 32 bits a 8 bits), lo que reduce el tamaño y acelera los cálculos en hardware compatible. Modelos cuantizados pueden ser hasta 4 veces más pequeños y mantener una precisión muy cercana al original.

**Ejemplo Práctico**

Imagina implementar un asistente virtual en un smartphone. Usar un Transformer estándar sería demasiado lento y demandante para el dispositivo. Aplicando DistilBERT junto con quantization, el modelo puede ejecutarse rápidamente y con bajo consumo de batería, ofreciendo respuestas casi tan precisas como el modelo completo.

Estas técnicas permiten democratizar el acceso a Transformers, facilitando su uso en más aplicaciones y dispositivos, sin perder la calidad en el procesamiento del lenguaje.

## **9. Construcción de un modelo transformer personalizado**

### **9.1 Enunciado**

Dada una tarea específica de procesamiento del lenguaje natural (por ejemplo, traducción automática, resumen de texto, o respuesta a preguntas), diseña un modelo Transformer personalizado. Describe el proceso de diseño, las técnicas de preprocesamiento de datos utilizadas, y las métricas de evaluación que emplearás para medir el rendimiento del modelo.

### **9.2 Resolución**

Define un modelo simple, aplica cuantización dinámica para reducir el tamaño usando INT8 en las capas lineales y prepara el modelo para cuantización estática, mostrando ambos estados.

# Modelo simple secuencial FP32

fp32\_model = nn.Sequential(

nn.Linear(128, 256),

nn.ReLU(),

nn.Linear(256, 10)

)

fp32\_model.eval() # Modo evaluación

# Cuantización dinámica: cuantiza pesos de capas Linear a INT8

quantized\_dynamic\_model = torch.quantization.quantize\_dynamic(

fp32\_model, {nn.Linear}, dtype=torch.qint8, inplace=False

)

print("Modelo original (FP32):")

print(fp32\_model)

print("\nModelo cuantizado dinámicamente (INT8 pesos para Linear):")

print(quantized\_dynamic\_model)

# Cuantización estática (requiere calibración, aquí sólo preparación)

static\_model = copy.deepcopy(fp32\_model)

static\_model.eval()

static\_model.qconfig = torch.quantization.get\_default\_qconfig('fbgemm')

# Preparar modelo para calibración estática (aún no calibrado)

static\_model\_prepared = torch.quantization.prepare(static\_model, inplace=False)

print("\nEjemplos conceptuales de cuantización con PyTorch.")

Carga un modelo fine-tuned, realiza cuantización estática post-entrenamiento usando Optimum con ONNX Runtime y guarda el modelo resultante en formato ONNX cuantizado para inferencia eficiente.

# Asegúrate de tener instalado optimum con soporte onnxruntime:

# !pip install optimum[onnxruntime]

from optimum.onnxruntime import ORTQuantizer

from optimum.onnxruntime.configuration import AutoQuantizationConfig

# Ruta al modelo fine-tuned guardado previamente

model\_checkpoint\_dir = "./fine\_tuned\_distilbert\_classifier"

# Carpeta donde se guardará el modelo ONNX cuantizado

onnx\_quantized\_output\_dir = "./onnx\_quantized\_distilbert"

# Crear cuantizador a partir del modelo entrenado

ort\_quantizer = ORTQuantizer.from\_pretrained(model\_checkpoint\_dir)

# Configurar cuantización estática (requiere datos para calibración)

qconfig = AutoQuantizationConfig.avx2(config=None, is\_static=True)

# Ejecutar cuantización y exportar el modelo ONNX cuantizado

ort\_quantizer.quantize(

save\_dir=onnx\_quantized\_output\_dir,

quantization\_config=qconfig,

)

print(f"Modelo cuantizado y exportado a ONNX en {onnx\_quantized\_output\_dir}")

Carga el modelo PEGASUS pre-entrenado para resumen, tokeniza un texto largo, genera un resumen con beam search y muestra el resultado, manejando posibles errores por memoria o conexión.

from transformers import AutoTokenizer, AutoModelForSeq2SeqLM

import torch

model\_name\_summarization = "google/pegasus-xsum"

print(f"Cargando modelo {model\_name\_summarization}...")

try:

tokenizer\_summarization = AutoTokenizer.from\_pretrained(model\_name\_summarization)

model\_summarization = AutoModelForSeq2SeqLM.from\_pretrained(model\_name\_summarization)

device\_sum = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

model\_summarization.to(device\_sum)

model\_summarization.eval()

print("Modelo de resumen cargado.")

# Nuevo texto para resumir

ARTICLE\_TO\_SUMMARIZE = ("""

Climate change continues to pose a significant threat to ecosystems and communities worldwide.

Rising global temperatures have led to increased frequency of extreme weather events, including

hurricanes, droughts, and floods. Scientists warn that urgent actions are necessary to reduce

greenhouse gas emissions and limit global warming to safe levels. Governments and organizations

are investing in renewable energy sources and implementing policies to promote sustainability.

Despite these efforts, many regions face challenges adapting to the rapidly changing environment,

highlighting the need for continued research and international cooperation.

""")

print(f"\nTexto a resumir:\n{ARTICLE\_TO\_SUMMARIZE}")

inputs\_summarization = tokenizer\_summarization(

ARTICLE\_TO\_SUMMARIZE,

max\_length=1024,

return\_tensors="pt",

truncation=True

).to(device\_sum)

print("\nGenerando resumen...")

with torch.no\_grad():

summary\_ids = model\_summarization.generate(

inputs\_summarization["input\_ids"],

num\_beams=4,

min\_length=30,

max\_length=60,

early\_stopping=True,

no\_repeat\_ngram\_size=3,

)

generated\_summary = tokenizer\_summarization.decode(

summary\_ids[0], skip\_special\_tokens=True

)

print(f"\nResumen Generado:\n{generated\_summary}")

except Exception as e:

print(f"Error al cargar o usar el modelo de resumen: {e}")

print("Esto puede deberse a memoria insuficiente en Colab o problemas de red.")

print("Intenta reiniciar el entorno de ejecución o usar un modelo más pequeño si es necesario.")

## **10. Muy recomendado**

Entendiendo PyTorch: las bases de las bases para hacer inteligencia artificial <https://hackernoon.com/pytorch-y-su-funcionamiento-0p5j32hs>

Curso de IA Generativa. ¿Cómo usan los Transfomers la Atención? <https://youtu.be/NjDtyM-yKYk>

The math behind Attention: Keys, Queries, and Values matrices <https://youtu.be/UPtG_38Oq8o>

#### 