TRANSFORMERS CON PYTORCH

Adrián Yared Armas de la Nuez

Contenido

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}")
```

```
Forma de la entrada src: torch.Size([2, 10])
Forma de la entrada tgt (input decoder): torch.Size([2, 7])
Forma de la máscara src: torch.Size([2, 1, 10])
Forma de la máscara tgt: torch.Size([2, 7, 7])
Forma de la salida del decoder (antes del generador): torch.Size([2, 7, 128])
Forma de la salida final (log probs): torch.Size([2, 7, 1200])
```

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