UNIVERSIDAD NACIONAL DE INGENIERÍA

FACULTAD DE CIENCIAS

Optimizar un modelo BERT para dispositivos móviles con técnicas de pruning, quantization y knowledge distillation

Curso: Procesamiento del lenguaje Natural

Profesor: Cesar Jesus Lara Avila

Alumno: Olivares Ventura Ricardo Leonardo, 20192002A

índice

1. Introducción

En los últimos años, el modelo BERT (Bidirectional Encoder Representations from Transformers) ha revolucionado el campo del procesamiento de lenguaje natural (NLP) gracias a su capacidad para comprender contextos bidireccionales y su desempeño excepcional en una amplia variedad de tareas, como clasificación de texto, respuesta a preguntas, análisis de sentimientos, y más. Bert, desarrollado por Google, ha establecido un estándar en el uso de arquitecturas basadas en transformadores, permitiendo a las máquinas capturar relaciones semánticas complejas en grandes volúmenes de datos textuales.

Sin embargo, el éxito de BERT viene acompañado de desafíos significativos. Su arquitectura, aunque poderosa, es compleja y computacionalmente costosa, con cientos de millones de parámetros y un alto consumo de memoria. Estas características hacen que BERT sea difícil de implementar en dispositivos móviles o sistemas con recursos limitados, donde las capacidades de procesamiento y almacenamiento son significativamente menores en comparación con servidores y supercomputadoras. La creciente demanda de aplicaciones móviles que incorporen capacidades avanzadas de NLP, como asistentes virtuales y chatbots, ha puesto de manifiesto la necesidad de optimizar modelos como BERT para que puedan operar de manera eficiente en estos entornos restringidos.

Este proyecto aborda precisamente este desafío, utilizando técnicas avanzadas de optimización para reducir el tamaño y mejorar la eficiencia de BERT sin comprometer significativamente su rendimiento.

El objetivo en este primer entregable es doble. Por un lado, buscamos implementar y analizar la técnica de pruning en BERT para entender su impacto en términos de reducción de tamaño y cambios en el rendimiento. Por otro lado, realizaremos un ajuste fino post-pruning para mitigar cualquier pérdida de precisión y garantizar que el modelo resultante sea competitivo en tareas específicas, como la clasificación de texto. Finalmente, los resultados serán

comparados con el modelo original para evaluar los beneficios alcanzados y establecer una base sólida para futuras optimizaciones

2. Bert

Bert es un modelo basado en transformadores desarrollado por Google en 2018 para tareas de Procesamiento del Lenguaje Natural (NLP) como clasificación de texto, respuestas a preguntas, análisis de sentimientos, etc.

Bert ha revolucionado el campo del NLP al proporcionar un enfoque bidireccional en el pre entrenamiento de modelos de lenguaje. Su arquitectura basada en transformadores, presentada por primera vez en el artículo seminal "Attention is All You Need" (Vaswani et al., 2017), permite que BERT comprenda el contexto tanto hacia adelante como hacia atrás de un texto.

Con BERT nos interesa tener un modelo capaz de codificar un texto, obteniendo así una representación numérica que permita su correcta interpretación, por ello, es que BERT es prácticamente el resultado de tomar la red transformer y quedarnos únicamente con la parte del codificador

BERT se diferencia de modelos anteriores como los basados en redes recurrentes (RNN) o Long Short-Term Memory (LSTM) que leen el texto de izquierda a derecha o de derecha a izquierda, es decir, unidireccional, ya que BERT en cada paso puede considerar tanto el contexto anterior como el posterior a la palabra en cuestión, lo que permitirá capturar matices más complejos del lenguaje

BERT viene en dos formas:

- BERT base: 12 codificadores y 110 millones de parámetros
- BERT large: 24 codificadores y 340 millones de parámetros

2.1. Entrenamiento:

El entrenamiento de BERT se realiza en dos fases:

- Pre Entrenamiento: El pre entrenamiento se realiza en corpus de texto gigantes como el de Wikipedia y Google Books. Con este pre entrenamiento BERT aprende a clasificar palabras de forma bidireccional,

es decir, aprende a codificar cada palabra teniendo en cuenta todo su contexto, tanto lo que está a la izquierda como lo que está a la derecha.

Por ejemplo, una misma palabra puede tener 2 significados distintos dependiendo de su contexto:

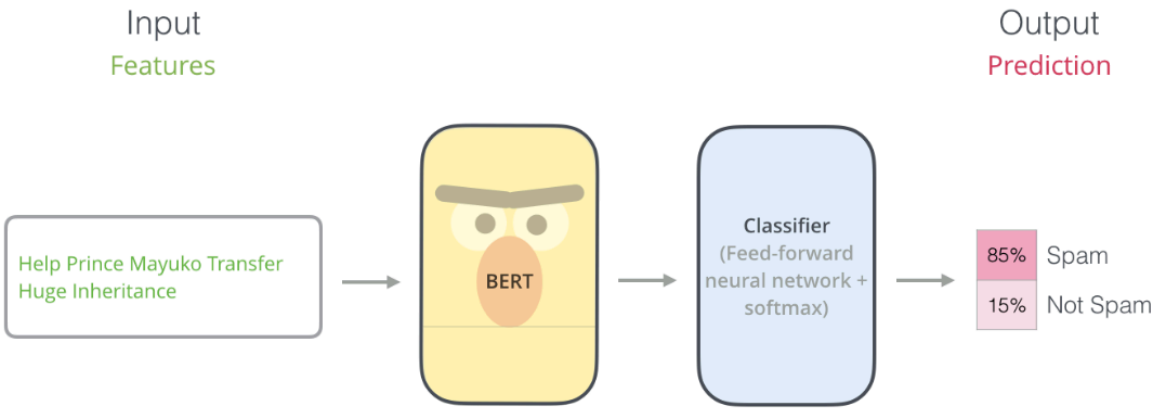
- Me siento en un **banco**
- Fui al **banco** a retirar dinero

Con esta bidireccionalidad es posible codificar de manera precisa el significado de la palabra **banco** en el ejemplo anterior, para lograr ello se realizan las siguientes dos tareas:

- El Modelado de Lenguaje Enmascarado (MLM): En esta tarea se le enseña al modelo a completar una palabra faltante en una frase (la palabra "faltante" se logra enmascarando ciertos tokens en la entrada), esta palabra faltante puede estar en cualquier ubicación, con esto se está forzando al modelo a que analice el texto en forma bidireccional aprendiendo así a comprender el lenguaje tal y como lo hacemos los seres humanos.
- Predicción de la Siguiente Oración (NSP): En esta tarea se le enseña al modelo a predecir la continuidad de una frase, dada dos frases A y B, la frase B le siga al A (isNext = TRUE) o es simplemente una frase aleatoria, para que BERT aprenda a hacer esto, el 50% de las veces se debe dar el caso donde efectivamente la frase B es la que sigue a la frase A y el 50% de las veces no lo es

Con estas dos tareas tenemos un modelo robusto capaz de representar de una manera muy completa relaciones bidireccionales entre palabras, a partir de este modelo base podemos ajustar BERT para tareas específicas.

Una de las formas directas de emplear BERT es usarlo para clasificar un fragmento de texto. El modelo tendría este aspecto:



Para entrenar un modelo de este tipo, principalmente se tiene que entrenar el clasificador, con cambios mínimos en el modelo BERT durante la fase de entrenamiento. Este proceso de entrenamiento se llama Fine-Tunning.

Otros ejemplos de este uso podrían incluir:

- Análisis de sentimientos
    - Entrada: Reseña de película/producto. Salida: ¿La revisión es positiva o negativa?
- Comprobación de hechos
    - Entrada: oración. Salida: “Declaración” o “No declaración”

2.2. Fine-Tuning:

Luego del pre entrenamiento de BERT, se debe realizar un proceso de ajuste para que resuelva tareas específicas (como clasificación de texto, etiquetado de entidades, preguntas, análisis de sentimientos, etc). Para ello, se agregan capas adicionales:

- Red neuronal (Capa Densa o Fully Connected Layer):
    - Se coloca justo después de la salida de BERT
    - Convierte representaciones contextuales producidas por BERT en un formato adecuado para la tarea específica
    - Por ejemplo, si BERT produce vectores de 768 dimensiones (para BERT base), esta capa puede reducir esta dimensión a un número más manejable
- Capa Softmax:
    - Se utiliza para tareas de clasificación.

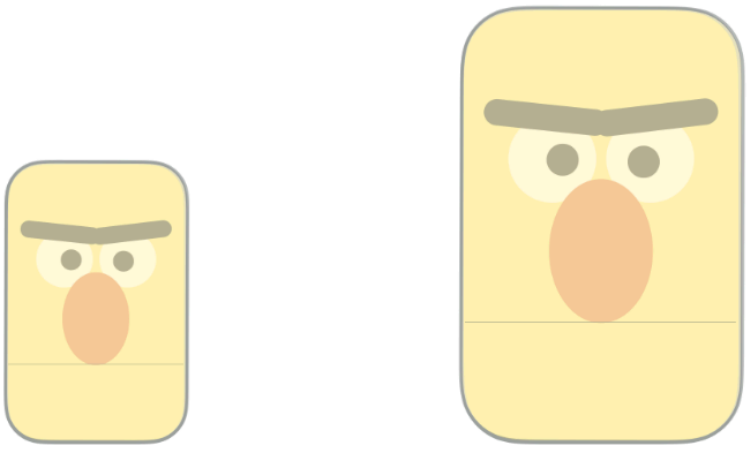
- Transforma los valores en probabilidades (suma 1) para cada clase.
- Por ejemplo, en una tarea de clasificación binaria (positivo/negativo), esta capa produce dos probabilidades (una para cada clase).

Luego de agregar las capas, se vuelve a entrenar de extremo a extremo (es decir, se ajustan los pesos de todo el modelo BERT y las capas adicionales):

- Sin embargo, los ajustes en las capas internas de BERT suelen ser menores porque ya están pre entrenadas
- El entrenamiento utiliza un conjunto de datos etiquetados específico para la tarea, lo que permite adaptar las representaciones generales aprendidas por BERT al problema concreto
- El entrenamiento solo refina el modelo para que sea más efectivo en la tarea particular

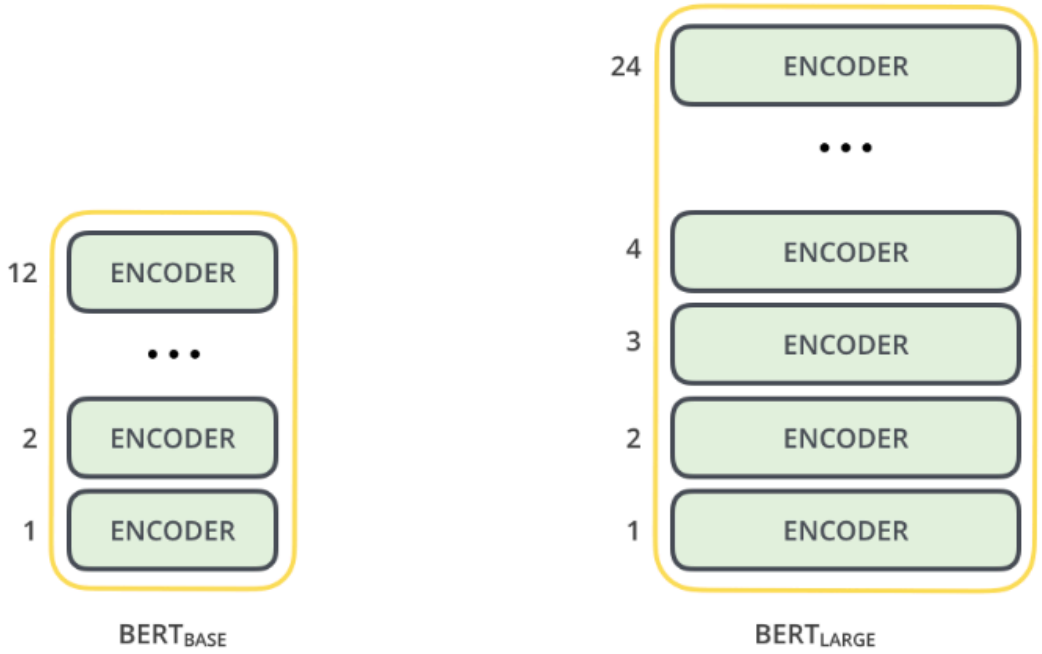
2.3. Arquitectura de BERT:

Como se explicó en líneas anteriores, BERT se presenta en dos tamaños:



- BERT BASE: comparable en tamaño al Transformer OpenAI (para comparar rendimiento)
- BERT LARGE: un modelo ridículamente enorme increíbles resultados presentes en el paper original de BERT de Google

BERT es básicamente una pila de transformers encoders entrenados

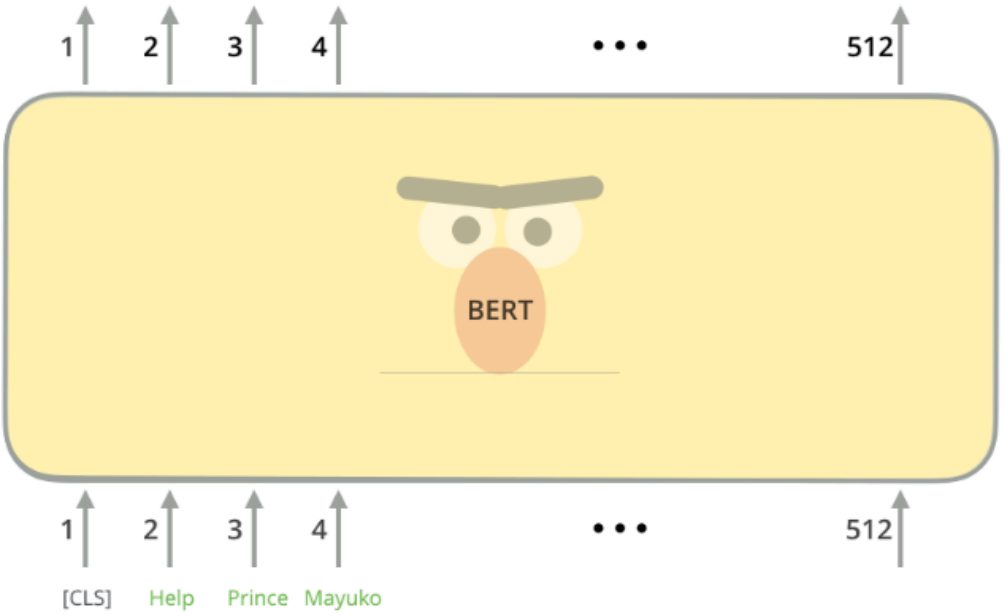


Ambos tamaños de modelos BERT tienen una gran cantidad de capas de encoder: 12 para la versión básica y 24 para la versión grande. Estos también tienen densas más grandes (768 y 1024 unidades ocultas respectivamente) y más cabezales de atención (12 y 16 respectivamente) que la configuración predeterminada en la implementación de referencia del Transformer en el paper inicial (6 capas de codificador, 512 unidades ocultas, y 8 cabezales de atención).

Características	Transformer	BERT base	BERT large
Cantidad de capas	6	12	24

Longitud de estados ocultos	512	768	1024
Cantidad de cabezales de atención	8	12	16

2.3.1. Entrada del modelo



a. Estructura del Dato de Entrada:

- En muchos casos (como en tareas de Pregunta-Respuesta o Reconocimiento de Entidades), el dato de entrada está compuesto por dos elementos:
  - Un párrafo (contexto)
  - Una pregunta o dos preguntas relacionadas con ese párrafo
- Estos dos elementos se concatenan en una única secuencia, separadas por un token especial [SEP] que indica el fin de cada segmento.
- Ejemplo, supongamos que tenemos:
  - Párrafo: “BERT es un modelo de lenguaje basado en transformadores”
  - Pregunta “¿Qué es BERT?”
  - El modelo los concatena como:

- `[CLS] BERT es un modelo de lenguaje basado en transformadores . [SEP] ¿Qué es BERT? [SEP]`
- El token especial [CLS] se coloca al inicio y se utiliza para tareas de clasificación o como un resumen de toda la entrada, este token será la dirección en donde vamos a tratar de clasificar todo lo que venga después de nuestra oración.
- [SEP] separa las dos frases y marca el final de cada segmento

b. Representaciones del Token:

Cada token en la secuencia de entrada recibe tres tipos de representaciones (embeddings) que se suman antes de ser procesadas por BERT:

- Token Embedding:
  - Representa cada palabra o subpalabra como un vector
  - Ejemplo: El token “BERT” se convierte en un vector numérico de tamaño fijo
- Positional Embedding:
  - Codifica la posición del token en la secuencia
  - Esto es necesario porque BERT no utiliza recurrencias ni convoluciones, y necesita información sobre el orden de los tokens
- Segment Embedding:
  - Indica a qué segmento pertenece cada token:
    - Segmento A (por ejemplo, el párrafo) se codifica con un embedding específico
    - Segmento B (por ejemplo, la pregunta) se codifica con otro embedding.

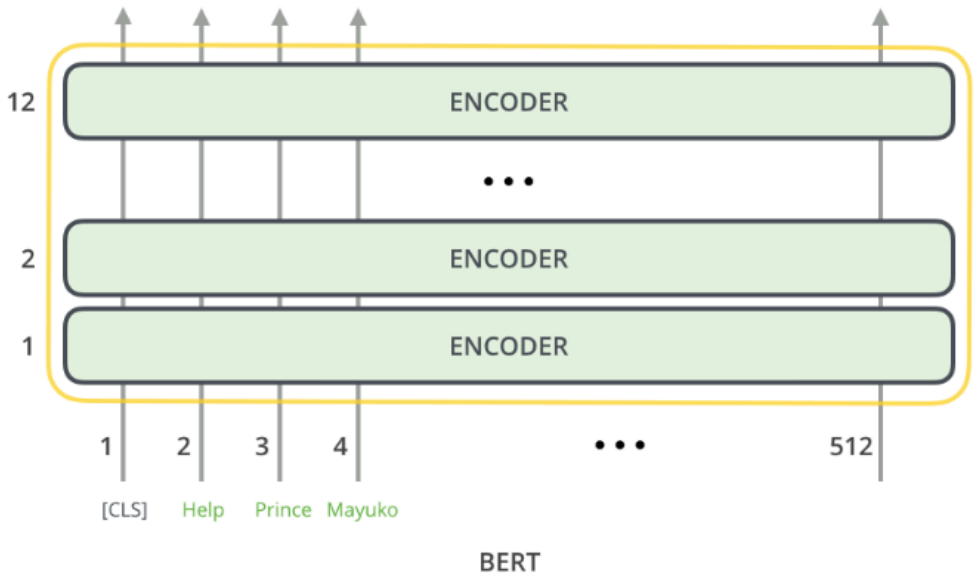
Podríamos colocarlo en la siguiente tabla:

Token	Token Embedding	Positional Embedding	Segment Embedding	Representación Final
[CLS]	Vec(1)	Pos (1)	Seg(1)	Vec(1) + Pos(1) + Seg(1)

BERT	Vec(2)	Pos (2)	Seg(1)	Vec(2) + Pos(2) + Seg(1)
[SEP]	Vec(3)	Pos (3)	Seg(1)	Vec(3) + Pos(3) + Seg(1)
¿Qué	Vec(4)	Pos (4)	Seg(2)	Vec(4) + Pos(4) + Seg(2)
[SEP]	Vec(5)	Pos (5)	Seg(2)	Vec(5) + Pos(5) + Seg(2)

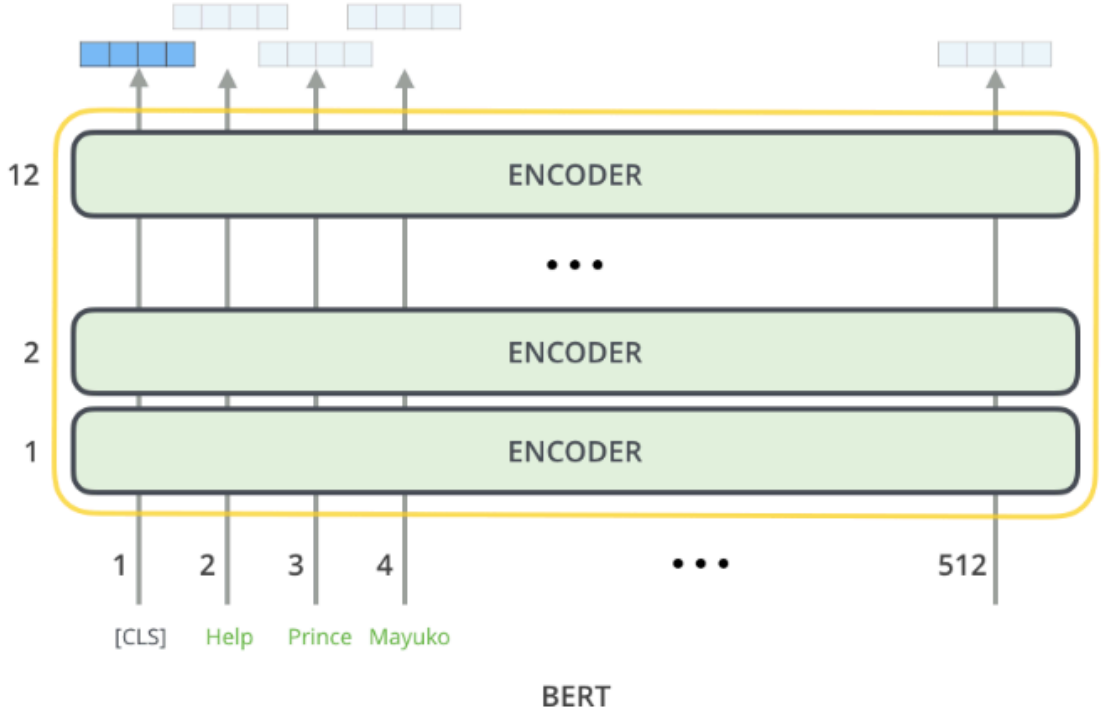
- Vec: Representación del token
- Pos: Embedding posicional
- Seg: Embedding de segmento
- **Suma de Representaciones**: Cada token tiene una representación final que es la suma de las tres componentes

BERT toma una secuencia de palabras como entrada que avanza hacia arriba en la pila, cada capa aplica autoatención, pasa sus resultados a través de una red de avance y luego los devuelve al siguiente encoder:

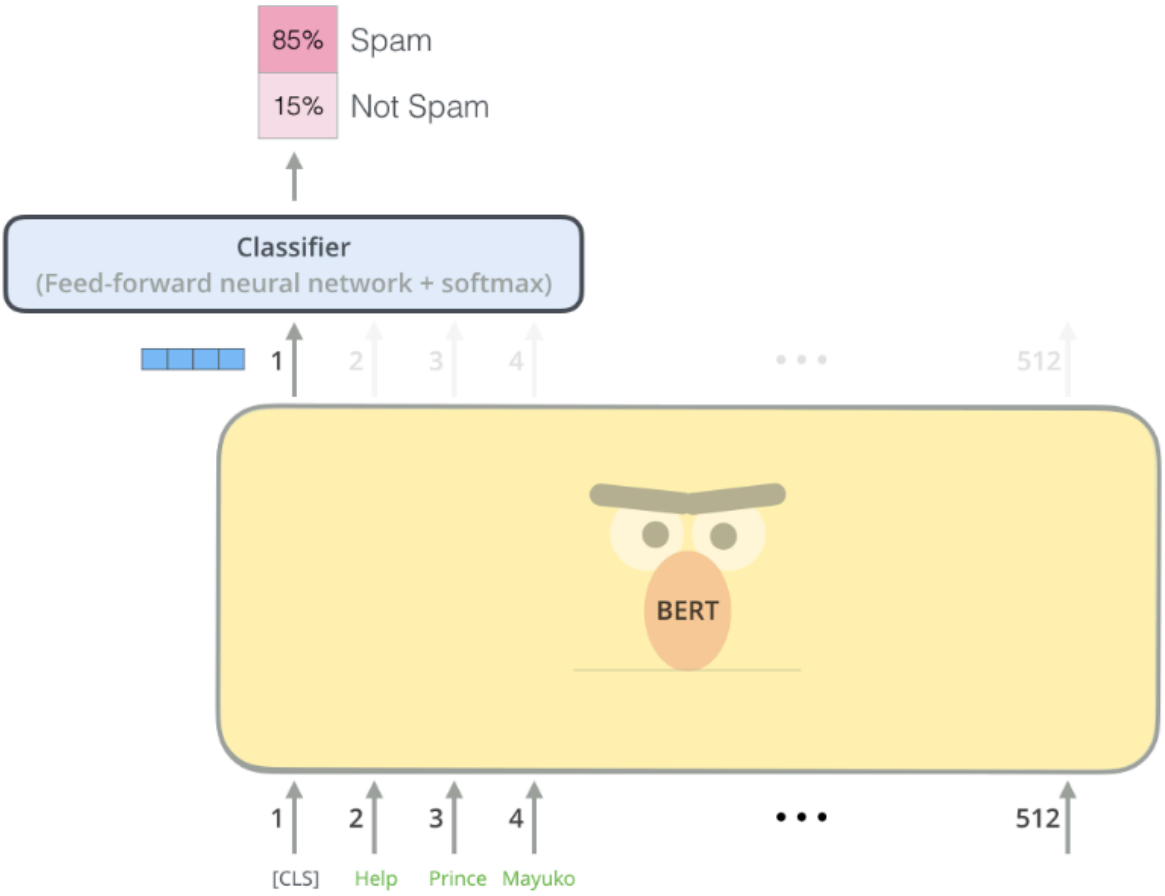


2.3.2. Procesamiento por BERT:

- La representación combinada de los tokens se pasa a las **capas del codificador** de BERT.
- Estas capas son redes **transformer** que procesan las representaciones y generan un vector de salida para cada token, tomando en cuenta el contexto de toda la secuencia.
- Cada posición genera un vector de un tamaño fijo (768 en BERT Base). Para temas de clasificación de oraciones, nos enfocamos en la salida de solo la primera posición (es decir, a la que le pasamos el token [CLS] especial)



- Este vector ahora se puede usar como entrada para un clasificador de nuestra elección para una determinada tarea, por ejemplo, podemos tener el siguiente caso:



BERT está compuesto por varios componentes clave:

2.3.1. Embeddings:

- Como se explicó líneas arriba, cada token tiene una representación inicial que combina:
  - Token Embedding.
  - Positional Embedding.
  - Segment Embedding.

2.3.2. Capas del Codificador:

- **BERT usa capas transformer encoder, que consisten en:**
  - Multi-Head Attention: Permite que cada token "preste atención" a otros tokens en la secuencia, capturando relaciones contextuales.
  - Feedforward Neural Network (FFN): Una red neuronal que procesa las salidas del mecanismo de atención.

    - Normas y Residuales: Ayudan a estabilizar el entrenamiento.
- **Números de capas:**
    - BERT Base: 12 capas.
    - BERT Large: 24 capas.
- **Diagrama del Encoder:**

```
Input Embeddings --> [Multi-Head Attention] --> [Feedforward Layer] --> Output
```

2.3.3. Clasificador:

- Una red neuronal adicional que toma la salida del primer token [CLS] para tareas de clasificación
- Para tareas de clasificación, se agrega una capa densa o softmax encima del token [CLS]
- Este clasificador toma el vector generado por BERT para [CLS] y produce la salida deseada

2.3.4. Decodificador:

- Este componente predice los tokens enmascarados durante el pre entrenamiento, permitiendo al modelo aprender representaciones contextuales ricas.

2.4. Complejidad Computacional de BERT:

El modelo BERT es computacionalmente costoso debido a su dependencia de la arquitectura Transformer, en particular el mecanismo de Atención Multi-Cabeza (Multi-Head Attention), y su diseño profundo con múltiples capas. A continuación se detalla su complejidad y los factores que contribuyen a su alto costo computacional

2.4.1. Complejidad del Mecanismo de Atención

El componente clave en BERT, el mecanismo de **Self-Attention**, tiene una complejidad **cuadrática** respecto al tamaño de la secuencia de entrada n.

Funcionamiento del Self-Attention

- Cada token en la secuencia debe calcular su relación con todos los demás tokens.
- Esto implica calcular n×nn \times nn×n pesos de atención para una secuencia de longitud nnn.

Costos en Self-Attention

- **Construcción de Matrices de Atención:**
    - Para calcular la atención, se generan tres matrices:
        - Matriz de Consulta (Query): Q
        - Matriz de Clave (Key): K
        - Matriz de Valor (Value): V
    - Cada matriz tiene dimensiones n x d, donde d es la dimensión de la representación oculta.
- **Producto Escalar y Softmax:**
    - El producto escalar QK^T genera una matriz de dimensiones n x n, lo que requiere: O (n^2 . d)
    - Luego se aplica la normalización softmax a esta matriz para obtener los pesos de atención
- **Multiplicación por la Matriz de Valores V:**
    - Una vez calculados los pesos de atención, se multiplican con la matriz VVV, lo que también tiene un costo de: O (n^2 . d)

Complejidad Total de Self-Attention: O (n^2 . d). Esto hace que el modelo sea particularmente costoso para secuencias largas

2.4.2. Capas del Codificador:

BERT utiliza L capas codificadoras, lo que amplifica el costo del mecanismo de atención. La complejidad total del modelo se escala como:

$$O (L . n^2 . d)$$

Donde:

- L: Número de capas del codificador.
- n: Longitud de la secuencia de entrada.
- d: Dimensión de la representación oculta.

2.4.3. Factores que Aumentan la Complejidad

- **Profundidad del Modelo**
    - **BERT Base:** 12 capas, 12 cabezas de atención por capa y 768 longitud de entrada
    - **BERT Large:** 24 capas, 16 cabezas de atención por capa y 1024 longitud de entrada
    - Cada capa requiere calcular el mecanismo de atención y pasar los resultados por redes feedforward, lo que aumenta el costo.
- **Longitud de la Secuencia**
    - La complejidad cuadrática respecto a nnn hace que las secuencias largas sean especialmente costosas. Por ejemplo:
    - Para n=512, el cálculo de n^2 implica 262,144 operaciones para cada capa, por cada cabeza de atención.
- **Número de Cabezas de Atención**
    - El uso de múltiples cabezas de atención mejora la capacidad del modelo para aprender relaciones complejas, pero también incrementa el costo computacional. Cada cabeza requiere realizar su propio cálculo de Q, K, V y la atención correspondiente.

2.4.4. Entrenamiento de BERT

El entrenamiento de BERT es especialmente demandante debido a:

- **Pre Entrenamiento con MLM y NSP:**
    - MLM: Requiere múltiples pasos de cómputo para predecir las palabras enmascaradas en una secuencia.
    - NSP: Procesa pares de oraciones, duplicando la entrada y el cómputo en comparación con un modelo basado en una sola oración.
- **Recursos Computacionales**
    - Para entrenar BERT desde cero, se requieren GPUs o TPUs con grandes cantidades de memoria y capacidad de procesamiento paralelo.
    - Ejemplo: BERT Large se entrenó con 16 TPUs durante 4 días usando el conjunto de datos BooksCorpus (800M palabras) y Wikipedia (2,500M palabras).

2.4.5. Uso durante Inferencia

La inferencia también es costosa debido a:

- **Atención Completa:**
    - Incluso para tareas más pequeñas (e.g., clasificación de texto corto), el modelo aplica atención a toda la secuencia, lo que lo hace ineficiente para textos largos.
- **Tamaño del Modelo:**
    - El gran número de parámetros (110M en BERT Base y 340M en BERT Large) implica una gran carga para la memoria y el tiempo de cómputo.

2.5. Técnicas de Optimización

Para reducir la complejidad computacional y los requisitos de memoria, se utilizan tres técnicas principales: **pruning**, **quantization**, y **knowledge distillation**. Estas técnicas permiten optimizar BERT sin comprometer significativamente su desempeño.

2.5.1. Pruning (Poda de Modelo):

- Esta técnica elimina partes del modelo que contribuyen poco al rendimiento, reduciendo el tamaño y el costo computacional
- **Esta es la técnica que utilizaremos en este primer entregable**
- Algunas cabezas en el mecanismo de atención son redundantes, estas se eliminan sin una pérdida notable de precisión
- Se eliminan neuronas poco significativas en las redes feedforward
- **Ventas:**
    - Reducción significativa del número de parámetros y de los tiempos de inferencia.
    - Menor uso de memoria.
- **Desventajas:**
    - Si no se realiza con cuidado, puede degradar el rendimiento

2.5.2. Quantization:

- La cuantización reduce la precisión de los pesos y activaciones del modelo, pasando de 32 bits flotantes (FP32) a formatos más compactos como 8 bits enteros (INT8).
- Convierte pesos y cálculos de atención en representaciones de menor precisión
- Implementar cuantización post-entrenamiento o durante el entrenamiento.
- **Ventajas:**
    - Reducción del tamaño del modelo en disco.
    - Acelera la inferencia en hardware optimizado para enteros, como CPUs modernas y GPUs.
- **Desventajas:**
    - Una cuantización agresiva puede afectar el rendimiento del modelo.
- Ejemplo:
    - Técnicas como **Dynamic Quantization** ajustan la precisión durante la inferencia según el rango de valores.

2.5.3. Knowledge Distillation:

- Un modelo grande (modelo **maestro**) transfiere su conocimiento a un modelo más pequeño (**modelo estudiante**), que es más rápido y eficiente.
- **DistilBERT** es un ejemplo directo de knowledge distillation aplicado a BERT. Se entrena un modelo más pequeño para replicar las salidas del modelo maestro.
- Se optimiza el estudiante utilizando una combinación de
    - Pérdida por imitación (salidas del maestro).
    - Pérdida supervisada en tareas específicas.
- **Ventajas:**
    - Resulta en modelos más pequeños (por ejemplo, la mitad del tamaño de BERT) con una pérdida mínima de precisión.
    - Reduce tanto los costos de inferencia como los de entrenamiento.
- **Desventajas:**
    - El entrenamiento del modelo estudiante requiere tiempo y ajustes finos para balancear precisión y eficiencia.

Técnica	Reducción de Complejidad	Impacto en el Rendimiento	Facilidad de Implementación	Hardware Compatible
Pruning	Alta	Moderado	Moderada	Universal
Quantization	Alta	Bajo a Moderado	Alta	CPUs y GPUs optimizadas
Knowledge Distillation	Muy Alta	Bajo	Moderada a Alta	Universal

2.6. Pruning:

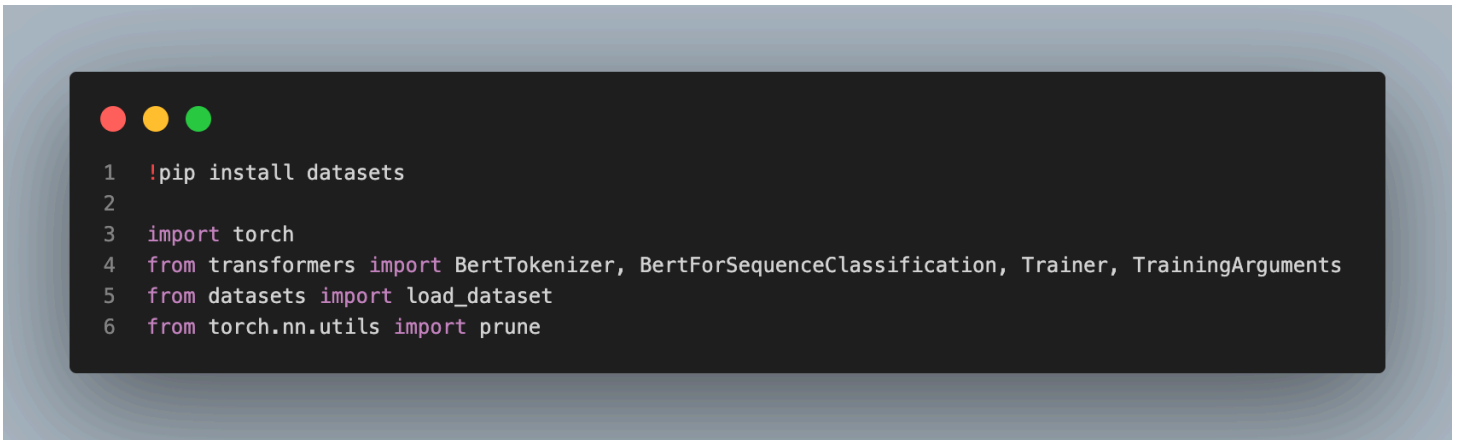
El pruning es una técnica de reducción de parámetros en redes neuronales, diseñada para simplificar el modelo, disminuir su tamaño y acelerar las inferencias, reduciendo el costo computacional. El proceso implica identificar y eliminar (o enmascarar) pesos menos relevantes en las capas del modelo, como aquellos cercanos a cero

Pasos principales del proceso de pruning:

- Identificación de pesos irrelevantes: Se decide qué pesos eliminar basándose en métricas como la magnitud (L1 norm) o importancia
- Aplicación de máscaras: Los pesos identificados no se eliminan directamente, sino que se enmascaran (se establecen en cero). Esto permite evaluar su impacto antes de una eliminación definitiva.
- **Eliminación de pesos (opcional):** Se pueden eliminar los pesos enmascarados para reducir el tamaño del modelo.
- **Ajuste fino (fine-tuning):** Tras el *pruning*, el modelo puede perder algo de precisión. Un ajuste fino posterior ayuda a recuperar rendimiento.

3. Código:

- Instalamos las dependencias que utilizaremos para el modelo BERT



```
!pip install datasets

import torch
from transformers import BertTokenizer, BertForSequenceClassification, Trainer, TrainingArguments
from datasets import load_dataset
from torch.nn.utils import prune
```

- Cargamos el modelo BERT base preentrenado y tokenizer

```
# Cargar modelo BERT base preentrenado y tokenizer
model_name = "bert-base-uncased"
tokenizer = BertTokenizer.from_pretrained(model_name)
model = BertForSequenceClassification.from_pretrained(model_name, num_labels=2)
```

- Cargamos el dataset que utilizaremos en este proyecto, en este caso utilizaremos el dataset sst2, el cual es un dataset que contiene frases extraídas de reseñas de películas y está etiquetada para el análisis de sentimientos. Cada frase está asociada con una etiqueta que indica si el sentimiento expresado es positivo o negativo

```python
# Cargar conjunto de datos
# Este dataset contiene frases extraídas de reseñas de películas
# y está etiquetado para el análisis de sentimientos. Cada frase
# está asociada con una eitqueta que indica si el sentimiento expresado
# es positivo o negativo
dataset = load_dataset("glue", "sst2")
encoded_dataset = dataset.map(
    lambda examples: tokenizer(examples['sentence'], truncation=True, padding='max_length'), batched=True
)

# Separar en conjuntos de entrenamiento y prueba
train_dataset = encoded_dataset["train"]
test_dataset = encoded_dataset["validation"]
```

- Función para evaluar el modelo, esta función la utilizaremos para evaluar el modelo antes y después de aplicar pruning, ya que nos devolverá las métricas de evaluación para ver el rendimiento del modelo BERT

```python
# Función para evaluar el modelo
def evaluate_model(model, dataset):
    trainer = Trainer(
        model=model,  # El modelo que será evaluado
        eval_dataset=dataset,  # Conjunto de datos para evaluación
        tokenizer=tokenizer,
    )
    eval_results = trainer.evaluate()  # Realiza la evaluación
    return eval_results  # Devuelve las métricas de evaluación
```

- Configuración del entrenamiento:

```python
# Configuración del entrenamiento
training_args = TrainingArguments(
    output_dir="./results",
    evaluation_strategy="epoch",
    learning_rate=2e-5,
    per_device_train_batch_size=16,
    num_train_epochs=3,
    weight_decay=0.01,
    logging_dir="./logs",
    logging_steps=10,
)

# Crear el Trainer
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=test_dataset,
    tokenizer=tokenizer,
)

# Evaluar modelo antes del pruning
print("Evaluación antes del pruning:")
eval_before = evaluate_model(model, test_dataset)
print(eval_before)
```

- Función para aplicar pruning al modelo BERT

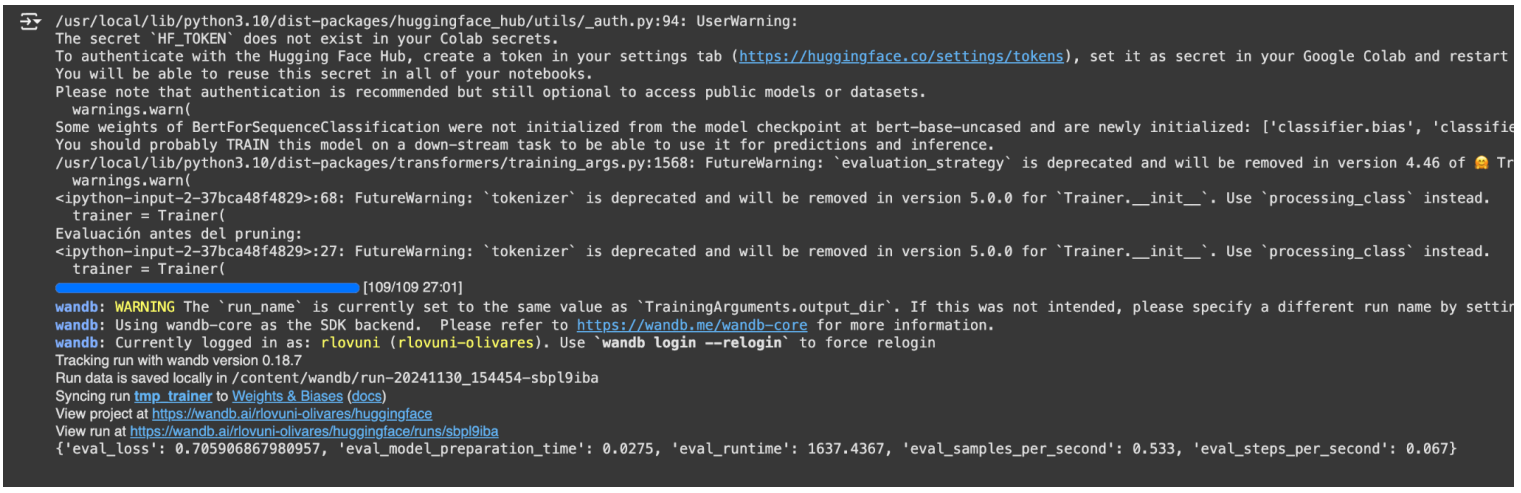
```python
def prune_bert(model, amount=0.3, prune_type='l1_unstructured', layers_to_prune=None):
    """
    Aplica pruning a las capas del modelo BERT de forma flexible.

    Args:
        model: El modelo BERT a optimizar.
        amount: Proporción de pesos a eliminar (entre 0 y 1).
        prune_type: El tipo de pruning que se desea aplicar. Puede ser 'l1_unstructured',
                    'random_unstructured', etc.
        layers_to_prune: Lista de nombres de capas a las que se les aplicará pruning. Si es None,
                         se aplicará a todas las capas lineales.

    """
    if layers_to_prune is None:
        # Si no se especifican capas, aplicamos pruning a todas las capas lineales
        layers_to_prune = [name for name, module in model.named_modules() if isinstance(module, torch.nn.Linear)]

    print(f"Aplicando pruning con {amount*100}% de reducción utilizando {prune_type} a las siguientes capas:")
    print(layers_to_prune)

    # Iterar sobre todas las capas especificadas y aplicar pruning
    for name, module in model.named_modules():
        if name in layers_to_prune and isinstance(module, torch.nn.Linear):
            print(f"Pruning en la capa: {name} ({module.__class__.__name__})")

            # Aplicar el tipo de pruning especificado
            if prune_type == 'l1_unstructured':
                prune.l1_unstructured(module, name="weight", amount=amount)
            elif prune_type == 'random_unstructured':
                prune.random_unstructured(module, name="weight", amount=amount)
            else:
                raise ValueError(f"Tipo de pruning '{prune_type}' no soportado.")

            # Consolidar pruning eliminando la máscara y dejando solo los pesos
            prune.remove(module, "weight")

    print("Pruning completado.")

# Aplicar pruning
prune_bert(model, amount=0.3)
```

- Evaluamos el modelo luego del pruning para luego comparar los resultados con el modelo antes de aplicar pruning

```python
# Evaluar modelo luego del pruning
print("Evaluación luego del pruning:")
eval_before = evaluate_model(model, test_dataset)
print(eval_before)
```

4. Resultados

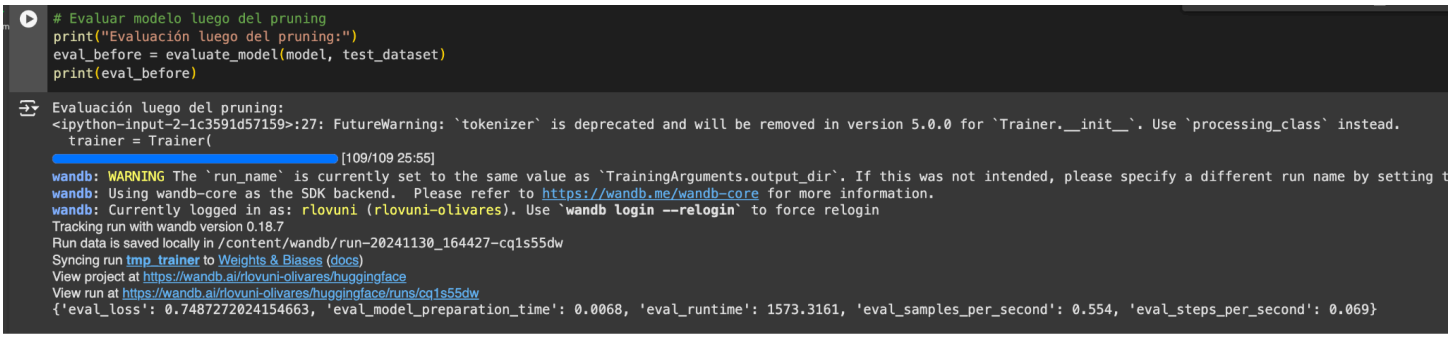
- Primero, entrenamos el modelo BERT sin aplicar pruning, y obtuvimos los siguientes resultados por consola:

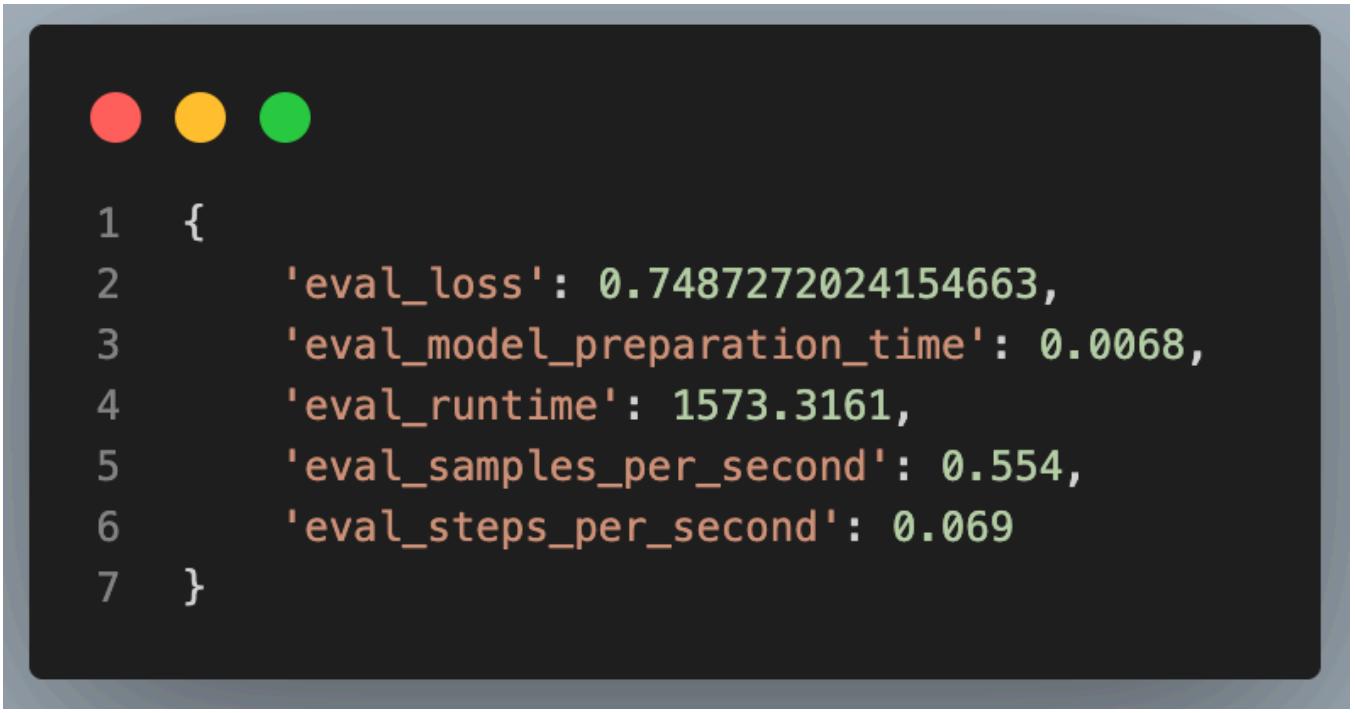



```python
{
    'eval_loss': 0.705906867980957,
    'eval_model_preparation_time': 0.0275,
    'eval_runtime': 1637.4367,
    'eval_samples_per_second': 0.533,
    'eval_steps_per_second': 0.067
}
```

- Luego, aplicamos pruning sobre el modelo BERT y obtuvimos los siguientes resultados

```
[2]
/usr/local/lib/python3.10/dist-packages/huggingface_hub/utils/_auth.py:94: UserWarni
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab (http
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access public m
  warnings.warn(
Some weights of BertForSequenceClassification were not initialized from the model ch
You should probably TRAIN this model on a down-stream task to be able to use it for
Aplicando pruning con 30.0% de reducción utilizando l1_unstructured a las siguientes
['bert.encoder.layer.0.attention.self.query', 'bert.encoder.layer.0.attention.self.k
Pruning en la capa: bert.encoder.layer.0.attention.self.query (Linear)
Pruning en la capa: bert.encoder.layer.0.attention.self.key (Linear)
Pruning en la capa: bert.encoder.layer.0.attention.self.value (Linear)
Pruning en la capa: bert.encoder.layer.0.attention.output.dense (Linear)
Pruning en la capa: bert.encoder.layer.0.intermediate.dense (Linear)
Pruning en la capa: bert.encoder.layer.0.output.dense (Linear)
Pruning en la capa: bert.encoder.layer.1.attention.self.query (Linear)
Pruning en la capa: bert.encoder.layer.1.attention.self.key (Linear)
Pruning en la capa: bert.encoder.layer.1.attention.self.value (Linear)
Pruning en la capa: bert.encoder.layer.1.attention.output.dense (Linear)
Pruning en la capa: bert.encoder.layer.1.intermediate.dense (Linear)
Pruning en la capa: bert.encoder.layer.1.output.dense (Linear)
Pruning en la capa: bert.encoder.layer.2.attention.self.query (Linear)
Pruning en la capa: bert.encoder.layer.2.attention.self.key (Linear)
Pruning en la capa: bert.encoder.layer.2.attention.self.value (Linear)
Pruning en la capa: bert.encoder.layer.2.attention.output.dense (Linear)
Pruning en la capa: bert.encoder.layer.2.intermediate.dense (Linear)
Pruning en la capa: bert.encoder.layer.2.output.dense (Linear)
Pruning en la capa: bert.encoder.layer.3.attention.self.query (Linear)
Pruning en la capa: bert.encoder.layer.3.attention.self.key (Linear)
Pruning en la capa: bert.encoder.layer.3.attention.self.value (Linear)
Pruning en la capa: bert.encoder.layer.3.attention.output.dense (Linear)
Pruning en la capa: bert.encoder.layer.3.intermediate.dense (Linear)
Pruning en la capa: bert.encoder.layer.3.output.dense (Linear)
Pruning en la capa: bert.encoder.layer.4.attention.self.query (Linear)
Pruning en la capa: bert.encoder.layer.4.attention.self.key (Linear)
Pruning en la capa: bert.encoder.layer.4.attention.self.value (Linear)
Pruning en la capa: bert.encoder.layer.4.attention.output.dense (Linear)
```

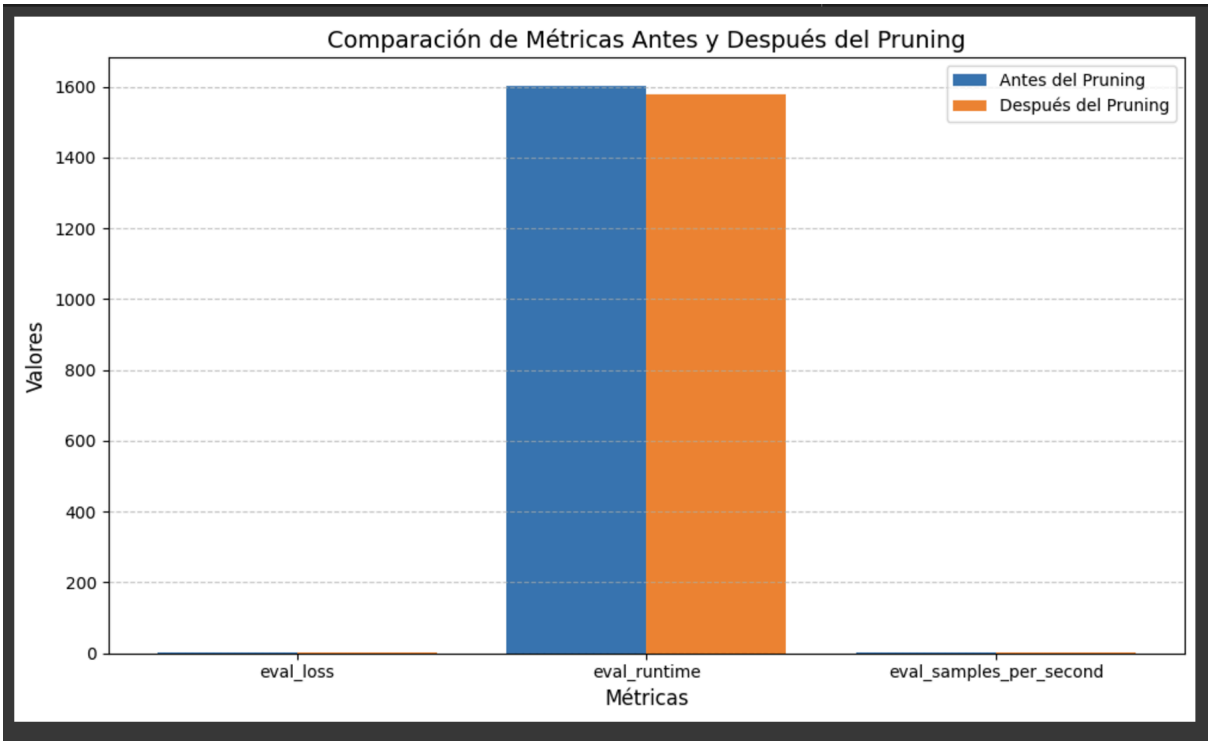




- Analizando los resultados:
  - Pérdida (eval_loss):
    - **Modelo original**: 0.7059
    - **Modelo con pruning**: 0.7487
    - La **pérdida** ha aumentado después de aplicar el pruning. Esto podría indicar que, aunque el pruning eliminó algunos parámetros del modelo, también redujo la capacidad del modelo para aprender o generalizar sobre los datos de prueba, lo que afecta negativamente el rendimiento.
    - **Posibles causas**: El pruning elimina pesos que pueden haber sido útiles, lo que reduce la capacidad del modelo de

adaptarse a ciertos patrones, especialmente si no se realiza un ajuste fino adecuado después del pruning.

- Tiempo de ejecución (eval_runtime):
  - **Modelo original**: 1637.44 segundos
  - **Modelo con pruning**: 1573.32 segundos
  - El **tiempo de ejecución** ha disminuido, lo que podría indicar que el pruning hizo que el modelo fuera más eficiente a nivel de cómputo al reducir su tamaño.
  - Aunque el modelo con pruning tiene una mayor pérdida, el hecho de que se tarde menos tiempo en realizar la evaluación sugiere que se está utilizando menos memoria y recursos, lo cual es una de las ventajas esperadas del pruning, ya que elimina parámetros no esenciales.
- Samples por segundo (eval_samples_per_second):
  - **Modelo original**: 0.533
  - **Modelo con pruning**: 0.554
  - El modelo con pruning tiene un pequeño **aumento en el número de muestras procesadas por segundo**. Esto también sugiere que el modelo es más eficiente después de reducir su tamaño.
- Pasos por segundo (eval_steps_per_second):
  - **Modelo original**: 0.067
  - **Modelo con pruning**: 0.069
  - También hay un ligero **aumento en los pasos por segundo**, lo que refuerza la idea de que el modelo es más rápido después de ser podado, incluso si la precisión en términos de pérdida es menor.



5. Conclusiones:

- Pruning y eficiencia: El pruning ha logrado hacer que el modelo sea más rápido (reduciendo el tiempo de ejecución y aumentando el número de muestras y pasos procesados por segundo), lo que indica una mejora en la eficiencia computacional. Sin embargo, esto ha venido a costa de un aumento en la pérdida del modelo, lo que sugiere que el modelo está perdiendo algo de capacidad para generalizar a los datos de prueba.
- Impacto en el rendimiento: El aumento en la pérdida muestra que el pruning, por sí solo, puede no ser suficiente para mantener el rendimiento si no se realiza un ajuste fino posterior al pruning. El ajuste fino post-pruning es crucial para que el modelo se recupere de la pérdida de capacidad después de eliminar parámetros, lo cual podría mejorar tanto la precisión como la eficiencia.

6. Bibliografía:

- Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2019). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. Proceedings of the 2019 Conference of the North

American Chapter of the Association for Computational Linguistics: Human Language Technologies, 1, 4171–4186.

- Rogers, A., Kovaleva, O., & Rumshisky, A. (2020). A Primer in BERTology: What We Know About How BERT Works. Transactions of the Association for Computational Linguistics
- Sanh, V., Debut, L., Chaumond, J., & Wolf, T. (2019). DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter.
- Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., ... & Rush, A. M. (2020).Transformers: State-of-the-Art Natural Language Processing.