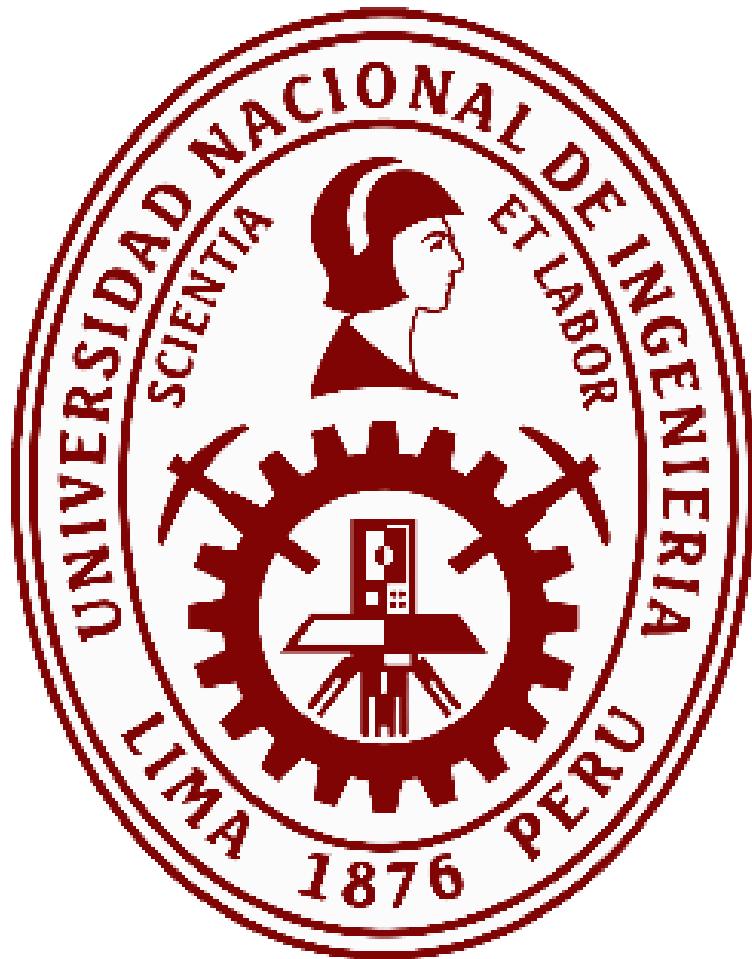


UNIVERSIDAD NACIONAL DE INGENIERÍA

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
2. BERT
3. Código
4. Resultados
5. Conclusiones
6. Bibliografía

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 el primer entregable fue 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

El objetivo en el segundo entregable fue implementar la quantization en el modelo podado, es decir, en el modelo de BERT luego de ser aplicado pruning y también realizar pruebas de inferencia en dispositivos móviles

En este tercer y último entregable el objetivo es implementar la técnica knowledge distillation para entrenar un modelo ligero derivado de uno original más completo. Se realizará una comparación entre ambos modelos, evaluando métricas de rendimiento. Adicionalmente, se analizará el proceso de desplegar el modelo destilado en la aplicación móvil. Finalmente se comparará el rendimiento de estas 3 técnicas

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:

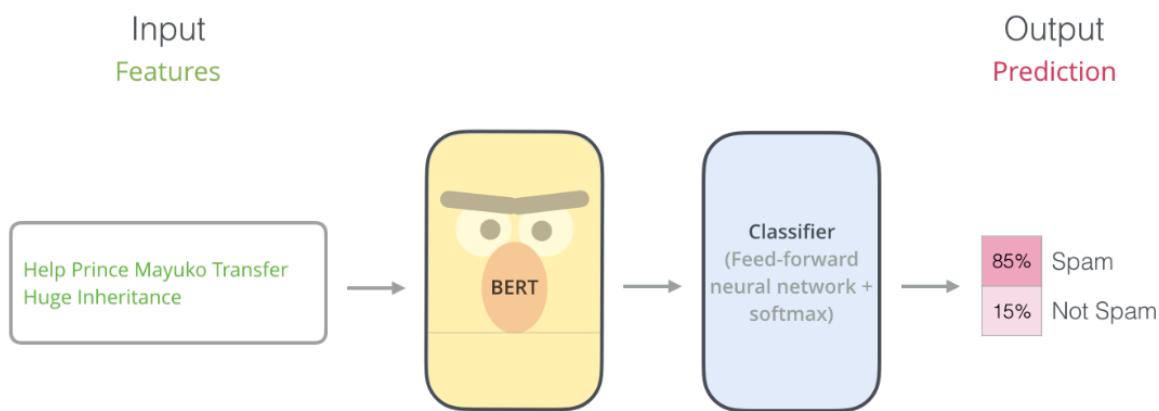
- Me siento en un **banco**
- Fui al **banco** a retirar dinero

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-Tuning.

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:

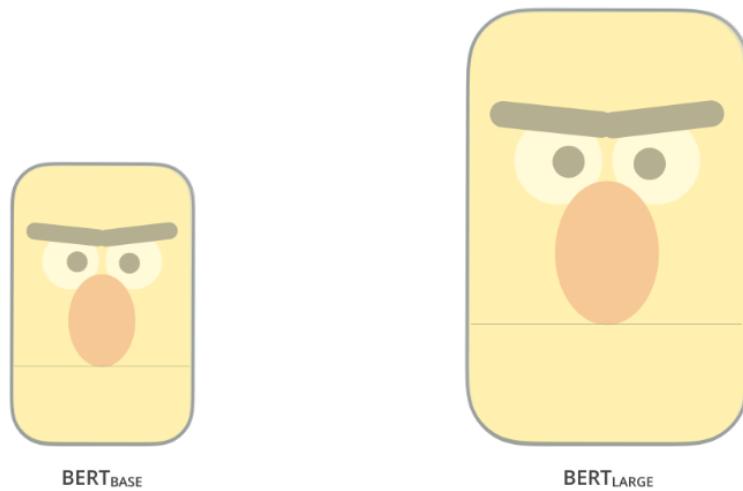
- Red neuronal (Capa Densa o Fully Connected Layer):
 - Se coloca justo después de la salida de BERT
 - Convierte representaciones contextuales producidas por BERT en un formato adecuado para la tarea específica
 - Por ejemplo, si BERT produce vectores de 768 dimensiones (para BERT base), esta capa puede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

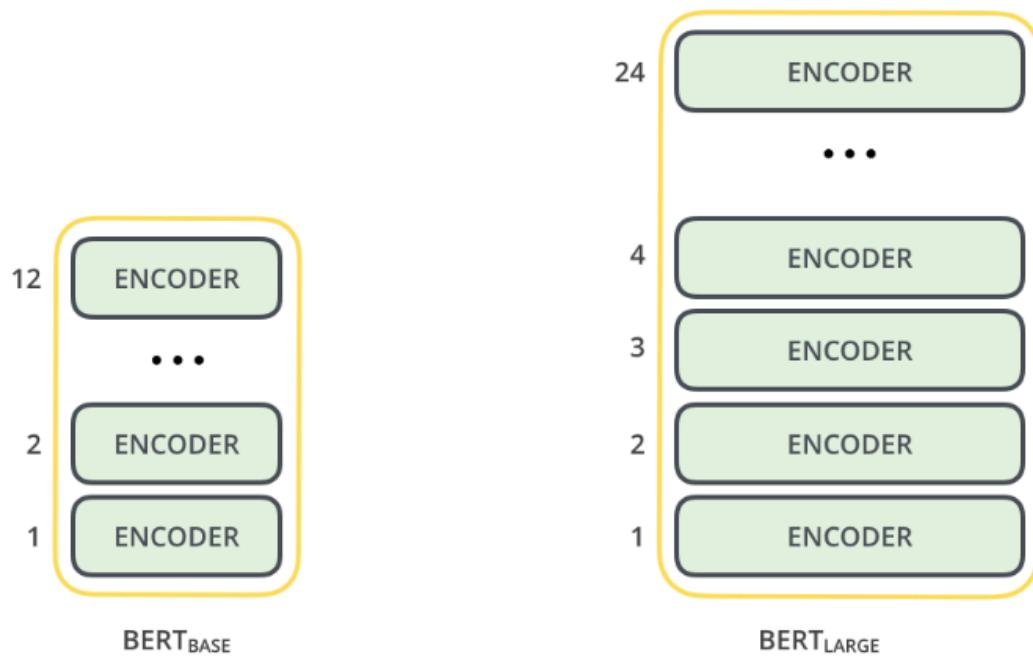
2.3. Arquitectura de BERT:

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

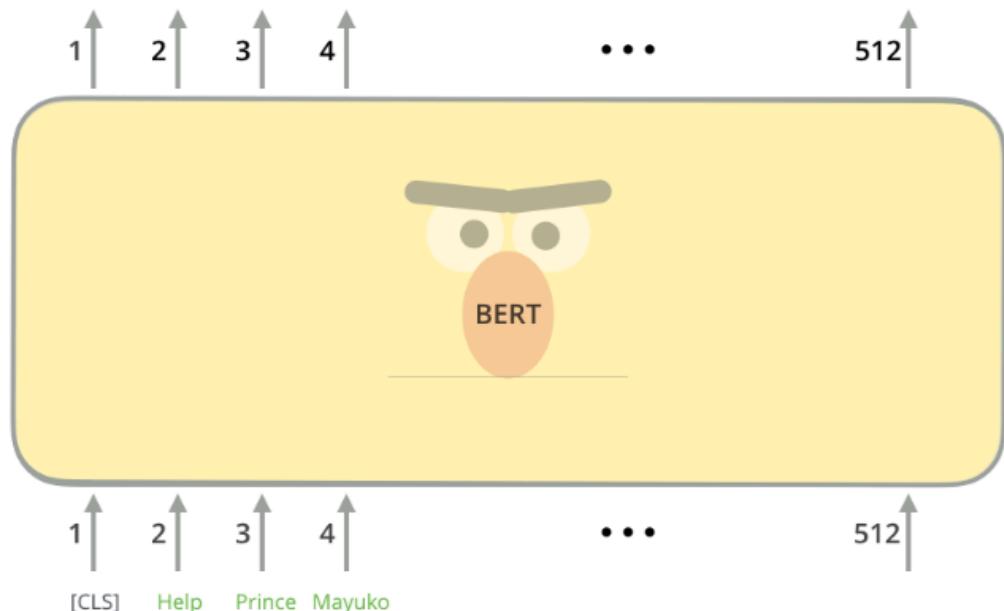
BERT es básicamente una pila de transformers encoders entrenados



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

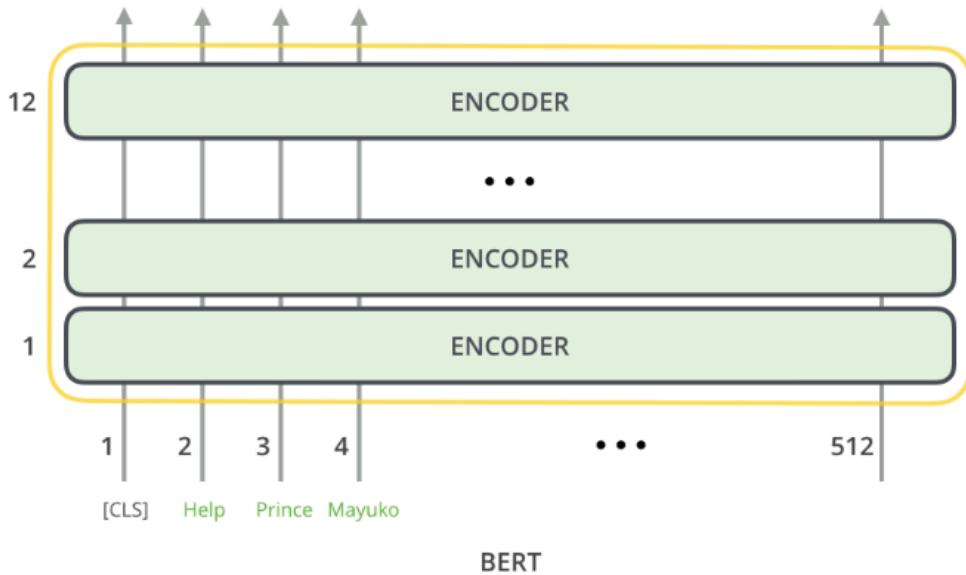
- Segmento B (por ejemplo, la pregunta) se codifica con otro embedding.

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)

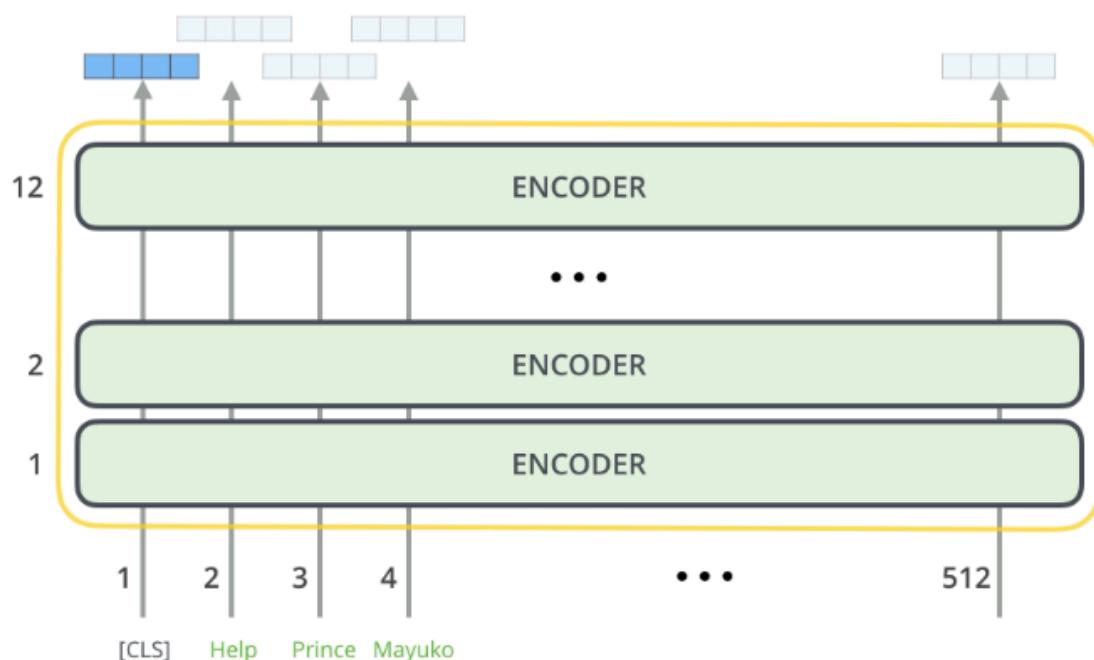
- Vec: Representación del token
- Pos: Embedding posicional
- Seg: Embedding de segmento
- **Suma de Representaciones:** Cada token tiene una representación final que es la suma de las tres componentes

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:

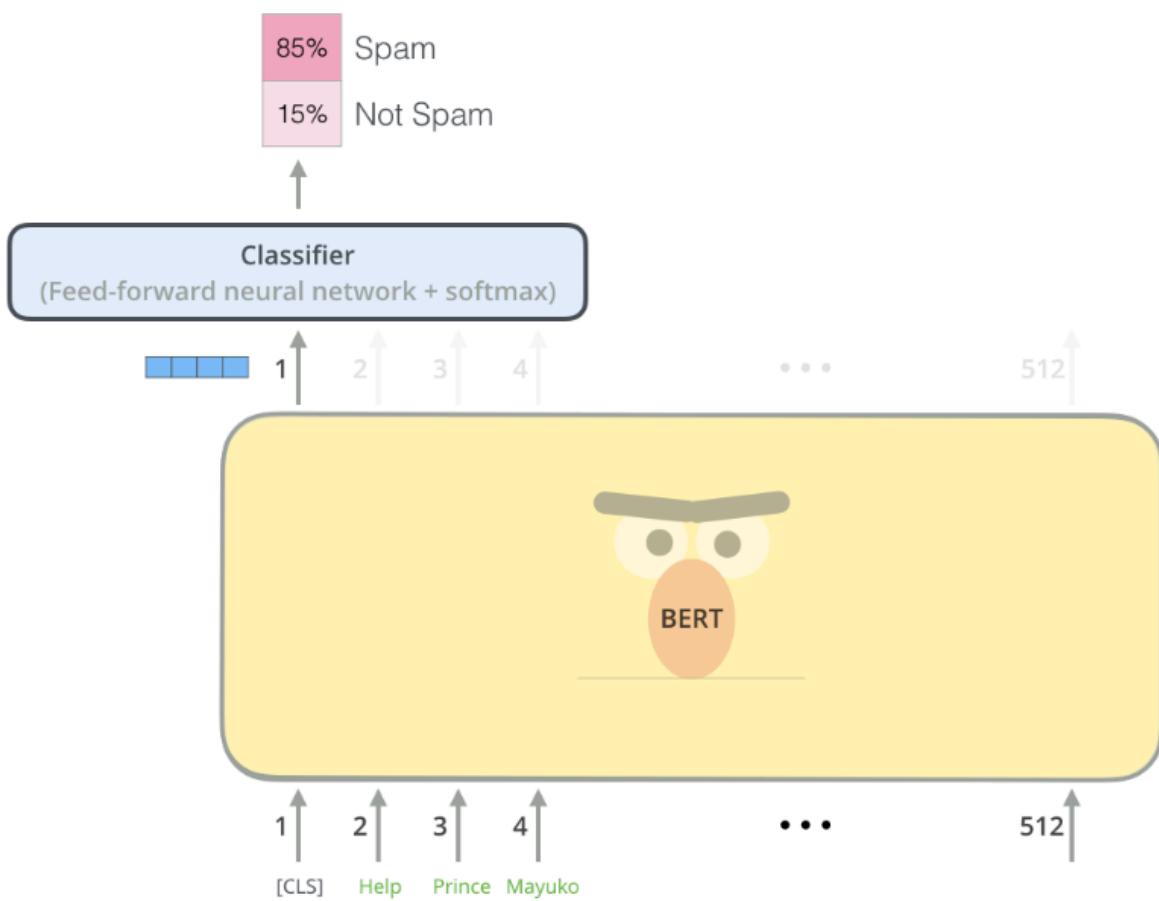


2.3.2. Procesamiento por BERT:

- 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 $n \times nn$ pesos de atención para una secuencia de longitud n.

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 $n \times d$, donde d es la dimensión de la representación oculta.
- **Producto Escalar y Softmax:**
 - El producto escalar QK^T genera una matriz de dimensiones $n \times n$, lo que requiere: $O(n^2 \cdot d)$
 - 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 V, lo que también tiene un costo de: $O(n^2 \cdot d)$

Complejidad Total de Self-Attention: $O(n^2 \cdot d)$. 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:

$$O(L \cdot n^2 \cdot d)$$

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.

2.7. Quantization:

La quantization es una técnica utilizada para reducir la precisión de los parámetros del modelo, con el objetivo de disminuir el tamaño del modelo y mejorar la eficiencia computacional sin una pérdida significativa en la calidad de las predicciones. En lugar de usar representaciones de punto flotante (como float 32), la quantization reduce los valores a enteros de menor precisión, como int8, lo que disminuye el uso de memoria y acelera la inferencia, especialmente en dispositivos con recursos limitados.

Importancia de la Quantization:

- **Reducción del tamaño del modelo:** Disminuye la memoria utilizada, permitiendo que los modelos se ajusten a dispositivos con recursos limitados, como móviles y dispositivos embebidos
- **Mejora de la velocidad de inferencia:** La reducción de la precisión de los pesos permite realizar cálculos más rápidos, lo que es crucial en entornos con recursos limitados
- **Optimización para Dispositivos de Baja Potencia:** En dispositivos móviles y sistemas con limitaciones de hardware, la quantization es fundamental para mejorar el rendimiento sin comprometer demasiado la precisión

La quantization lo aplicaremos a las capas lineales, por las siguientes razones:

- **Peso de la operación:** Las capas lineales son responsables de la mayor parte del cómputo en muchos modelos de redes neuronales, especialmente en modelos como BERT. Estas capas realizan multiplicaciones de matrices (una multiplicación entre las activaciones y los pesos de capa), que es una operación muy costosa. Cuantizar estas

operaciones puede dar grandes mejoras en eficiencia sin comprometer demasiado el rendimiento

- **Sensibilidad a la precisión:** Las capas lineales a menudo son más robustas a la pérdida de precisión cuando se cuantizar, ya que las multiplicaciones de los valores de los pesos se pueden aproximar con enteros sin afectar significativamente los resultados, siempre que el rango de valores de los pesos no sea demasiado amplio
- **Capacidades de optimización:** Los frameworks como PyTorch permiten una quantization dinámica de las capas lineales de manera eficiente, por lo que seleccionar solo estas capas para la quantization reduce el riesgo de errores o degradación del rendimiento, al tiempo que se mantiene la mayor parte de la eficiencia

El uso del int8 en el quantization es por las siguientes razones:

- **Reducción del tamaño del modelo:** Los valores de tipo int 8 ocupan solo 1 byte (8 bits) por valor, en comparación con los valores de tipo flotante de 32 bits (float32), que ocupan 4 bytes. Esto significa que al convertir los pesos del modelo de 32 bits a 8 bits, podemos reducir el tamaño del modelo en un 75%, lo cual es una mejora sustancial, especialmente para despliegue en dispositivos limitados.
- **Velocidad de Inferencia Acelerada:** Las operaciones con enteros (int8) son más rápidas que las que involucran números flotantes (float32). Esto se debe a que los procesadores, especialmente los procesadores móviles y chips dedicados a la inferencia de IA, están optimizados para trabajar con enteros en lugar de con punto flotante. La aritmética entera es más eficiente que la aritmética en coma flotante, lo que lleva a una inferencia más rápida.
- **Menor consumo de energía:** Los dispositivos con recursos limitados, como los móviles, tienden a tener un consumo de energía muy importante. Las operaciones con enteros son mucho menos intensivas en términos de energía que las operaciones con punto flotante, lo que hace que los modelos quantizados sean más eficientes energéticamente.

2.8. Knowledge Distillation:

Es una técnica de transferencia de aprendizaje que permite que un modelo más pequeño (student model) aprenda de un modelo más grande y complejo (teacher model). Introducida por Hinton (2015), esta técnica se utiliza para reducir el tamaño del modelo original, conservando la mayor parte de su rendimiento, lo que la hace especialmente útil en escenarios donde los recursos computacionales son limitados, como en dispositivos móviles.

En lugar de entrenar al modelo directamente con las etiquetas del conjunto de datos, esta técnica entrena al modelo estudiante utilizando el conocimiento implícito que el modelo maestro tiene sobre las relaciones entre las clases, representando por sus soft logits.

El proceso de KD tiene tres componentes principales:

- Teacher Model (Modelo maestro): Un modelo preentrenado grande y preciso (como BERT en este caso). Este modelo genera predicciones con una probabilidad distribuida sobre las clases posibles (logits).
- Student Model (Modelo estudiante): Un modelo más pequeño que intenta imitar el comportamiento del modelo maestro. Puede ser una versión reducida del maestro o una arquitectura completamente diferente.
- Temperatura: Se aplica una función de suavizado en las probabilidades generadas por el modelo maestro (softmax) para resaltar las relaciones entre las clases. Esto permite al estudiante aprender mejor los matices del maestro

En el caso de BERT, KD se utiliza para crear versiones más pequeñas, como DistilBERT, reduciendo su tamaño mientras se conserva el rendimiento. DistilBERT, por ejemplo, tiene solo el 66% del tamaño de BERT-base, pero conserva alrededor del 97% de su desempeño en tareas de NLP.

Pasos específicos de KD en BERT:

- **Teacher Model:** Se utiliza un modelo BERT preentrenado y afinado en una tarea específica (como clasificación de texto o respuesta a preguntas).

- **Student Model:** Se define una arquitectura más pequeña. En el caso de DistilBERT, el modelo conserva la estructura Transformer pero elimina capas intermedias y reduce dimensiones.
- **Logits y características intermedias:** Además de los logits finales, el modelo estudiante puede ser entrenado para aproximarse a las representaciones intermedias generadas por el modelo maestro en capas específicas.
- **Loss:** Se utiliza una combinación de pérdidas basadas en logits y representaciones intermedias

Ventajas:

- **Modelos compactos:** Reduce significativamente el tamaño de los modelos, haciéndolos adecuados para dispositivos móviles y aplicaciones en tiempo real
- **Eficiencia computacional:** Los modelos destilados requieren menos memoria y potencia de cálculo
- **Conservación del rendimiento:** Los modelos distiliados suelen mantener un alto rendimiento en comparación con modelos grandes
- **Flexibilidad arquitectónica:** Permite que el modelo estudiante tenga una arquitectura diferente del maestro, lo que facilita adaptaciones para casos específicos

Desventajas:

- **Dependencia del teacher model:** La calidad del modelo distilado depende en gran medida del rendimiento del modelo maestro
- **Tiempos de entrenamiento:** Aunque los modelos destilados son más rápidos en inferencia, el proceso de destilación puede ser computacionalmente costoso
- **Complejidad adicional:** Requiere ajustar hiperparámetros como la temperatura y los pesos de las pérdidas para obtener resultados óptimos.
- **Riesgo de sobreajuste a los logits:** El estudiante puede replicar el comportamiento del maestro sin entender completamente la tarea, lo que podría impactar su generalización en nuevos datos.

Ejemplos y casos de uso:

- **DistilBERT:** Un modelo distilado de BERT que mantiene un rendimiento competitivo mientras es más eficiente.
- **TinyBERT:** Otro modelo distilado diseñado para tareas en dispositivos móviles. Además de los logits, utiliza conocimientos intermedios de las capas de atención del maestro.
- **Uso en dispositivos móviles:** Los modelos distilados son ideales para chatbots, asistentes virtuales, y aplicaciones de procesamiento de texto en dispositivos móviles.

En nuestro caso utilizadmos DistilBERT debido a las siguientes razones:

- Compromiso entre rendimiento y eficiencia:
 - Tamaño reducido: DistilBERT tiene aproximadamente el 40% menos de parámetros que BERT base, lo que lo hace más liviano en términos de almacenamiento y uso de memoria
 - Velocidad: DistilBERT es aproximadamente un 60% más rápido que BERT en términos de inferencia y entrenamiento, lo que lo hace ideal para aplicaciones con restricciones en tiempo de ejecución
- Reducción de costos computacionales:
 - Menor demanda de recursos: Debido a su menor tamaño, DistilBERT requiere menos recursos computacionales, como RAM y GPU, lo que lo hace adecuado para entornos de producción donde el hardware puede ser limitado.
 - Ahorro energético: Modelos más pequeños y rápidos consumen menos energía lo que también puede ser un factor importante en entornos donde la sostenibilidad es una consideración
- Transferencia de conocimiento:
 - Entrenamiento eficiente: DistilBERT se entrena utilizando un método llamado Knowledge Distillation, donde el modelo maestro (BERT base) transfiere su conocimiento al modelo más pequeño (DistilBERT). Esto permite que DistilBERT conserve gran parte del rendimiento del modelo original mientras reduce su complejidad
 - Rendimiento competitivo: Aunque es más pequeño, DistilBERT logra un renacimiento muy cercano al de BERT base en tareas de procesamiento de lenguaje natural

2.9. Inferencia en Dispositivos Móviles:

La inferencia en dispositivos móviles se refiere al proceso de ejecutar un modelo de aprendizaje automático (como un modelo de red neuronal, por ejemplo BERT, después de que ha sido entrenado) en un dispositivo de recursos limitados, como un teléfono inteligente o una tablet. Este proceso es crucial para aplicaciones móviles de IA y aprendizaje automático, ya que permite ejecutar modelos sin necesidad de depender de un servidor o la nube para procesar las predicciones.

Sin embargo, realizar inferencia en dispositivos limitados, como teléfonos móviles, presenta ciertos desafíos debido a las limitaciones de hardware (como CPU, GPU, memoria y almacenamiento), lo que hace necesario optimizar los modelos de ML.

Desafíos de la Inferencia en Dispositivos Móviles

Los dispositivos móviles tienen limitaciones de recursos en comparación con los servidores o las estaciones de trabajo que normalmente entranan modelos de ML. Algunos de los desafíos comunes incluyen:

- **Limitaciones de memoria (RAM):** Los modelos de ML pueden ser grandes y consumir mucha memoria, lo que puede hacer que el dispositivo se quede sin memoria si el modelo no está optimizado.
- **Limitaciones de procesamiento (CPU/GPU):** Los modelos de ML pueden requerir cálculos intensivos, y los dispositivos móviles no siempre tienen la capacidad de procesamiento necesaria para hacer esto de manera eficiente.
- **Limitaciones de almacenamiento:** Los modelos grandes requieren espacio de almacenamiento en el dispositivo, y los teléfonos móviles generalmente tienen almacenamiento limitado.

Para superar estos problemas, se aplican técnicas de **optimización** como el **pruning** (poda), **quantization** (cuantización) y **knowledge distillation**, que ayudan a reducir el tamaño y la complejidad del modelo sin perder demasiado en términos de rendimiento.

¿Cómo Realizaremos la Inferencia en Dispositivos Móviles?

A continuación, detallamos los pasos que seguiremos para realizar la inferencia en dispositivos móviles con el modelo BERT que hemos entrenado

- **Convertiremos el Modelo BERT (luego de aplicarle pruning y quantization) a un Formato Compatible:** Para que un modelo pueda ejecutarse de manera eficiente en un dispositivo móvil, debe ser convertido a un formato adecuado para el hardware del dispositivo. Dos de los formatos más comunes para inferencia en móviles son **TensorFlow Lite** (TFLite) y **ONNX**.
 - **TensorFlow Lite** es una versión optimizada de TensorFlow para dispositivos móviles y está diseñado específicamente para modelos más pequeños y eficientes.
 - **ONNX** es un formato abierto que permite la ejecución de modelos en diferentes plataformas, y se puede usar en **React Native** con bibliotecas como **onnxruntime-react-native**
- **Implementar el Modelo en la App Móvil:** Una vez que el modelo ha sido convertido, se integra en la aplicación móvil. Para ello, se pueden usar bibliotecas de ML específicas de la plataforma, como **TensorFlow Lite** en Android y iOS o **ONNX Runtime** en React Native.
 - En **React Native**, la librería **@tensorflow/tfjs-react-native** o **onnxruntime-react-native** puede ser utilizada para cargar y ejecutar el modelo optimizado.
- **Realizar Inferencia:** El proceso de inferencia implica pasar datos de entrada a través del modelo para obtener predicciones. En un contexto móvil, esto se hace de la siguiente manera:
 - Se prepara la entrada (por ejemplo, una imagen o texto) para que sea compatible con el modelo (normalmente se realiza el preprocessamiento).
 - Se pasa la entrada al modelo cargado.
 - El modelo hace la inferencia y devuelve las predicciones (como una clasificación o puntuación).

3. Código del Modelo:

- Instalamos las dependencias que utilizaremos para el modelo BERT, para realizar el pruning y para realizar la quantization



```
1 !pip install datasets
2
3 import torch
4 from transformers import BertTokenizer, BertForSequenceClassification, Trainer, TrainingArguments
5 from datasets import load_dataset
6 from torch.nn.utils import prune
```



```
1 from torch.quantization import quantize_dynamic
```

- Cargamos el modelo BERT base preentrenado y tokenizer

```
● ● ●  
1 # Cargar modelo BERT base preentrenado y tokenizer  
2 model_name = "bert-base-uncased"  
3 tokenizer = BertTokenizer.from_pretrained(model_name)  
4 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

```
● ● ●  
1 # Cargar conjunto de datos  
2 # Este dataset contiene frases extraídas de reseñas de películas  
3 # y está etiquetado para el análisis de sentimientos. Cada frase  
4 # está asociada con una etiqueta que indica si el sentimiento expresado  
5 # es positivo o negativo  
6 dataset = load_dataset("glue", "sst2")  
7 encoded_dataset = dataset.map(  
8     lambda examples: tokenizer(examples['sentence'], truncation=True, padding='max_length'), batched=True  
9 )  
10  
11 # Separar en conjuntos de entrenamiento y prueba  
12 train_dataset = encoded_dataset["train"]  
13 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



```
1 # Función para evaluar el modelo
2 def evaluate_model(model, dataset):
3     trainer = Trainer(
4         model=model, # El modelo que será evaluado
5         eval_dataset=dataset, # Conjunto de datos para evaluación
6         tokenizer=tokenizer,
7     )
8     eval_results = trainer.evaluate() # Realiza la evaluación
9     return eval_results # Devuelve las métricas de evaluación
```

- Configuración del entrenamiento:



```
1 # Configuración del entrenamiento
2 training_args = TrainingArguments(
3     output_dir=".//results",
4     evaluation_strategy="epoch",
5     learning_rate=2e-5,
6     per_device_train_batch_size=16,
7     num_train_epochs=3,
8     weight_decay=0.01,
9     logging_dir=".//logs",
10    logging_steps=10,
11 )
12
13 # Crear el Trainer
14 trainer = Trainer(
15     model=model,
16     args=training_args,
17     train_dataset=train_dataset,
18     eval_dataset=test_dataset,
19     tokenizer=tokenizer,
20 )
21
22 # Evaluar modelo antes del pruning
23 print("Evaluación antes del pruning:")
24 eval_before = evaluate_model(model, test_dataset)
25 print(eval_before)
```

- Función para aplicar pruning al modelo BERT



```
1 def prune_bert(model, amount=0.3, prune_type='l1_unstructured', layers_to_prune=None):
2     """
3         Aplica pruning a las capas del modelo BERT de forma flexible.
4
5     Args:
6         model: El modelo BERT a optimizar.
7         amount: Proporción de pesos a eliminar (entre 0 y 1).
8         prune_type: El tipo de pruning que se desea aplicar. Puede ser 'l1_unstructured',
9                     'random_unstructured', etc.
10        layers_to_prune: Lista de nombres de capas a las que se les aplicará pruning. Si es None,
11                        se aplicará a todas las capas lineales.
12
13    """
14    if layers_to_prune is None:
15        # Si no se especifican capas, aplicamos pruning a todas las capas lineales
16        layers_to_prune = [name for name, module in model.named_modules() if isinstance(module, torch.nn.Linear)]
17
18    print(f"Aplicando pruning con {amount*100}% de reducción utilizando {prune_type} a las siguientes capas:")
19    print(layers_to_prune)
20
21    # Iterar sobre todas las capas especificadas y aplicar pruning
22    for name, module in model.named_modules():
23        if name in layers_to_prune and isinstance(module, torch.nn.Linear):
24            print(f"Pruning en la capa: {name} ({module.__class__.__name__})")
25
26        # Aplicar el tipo de pruning especificado
27        if prune_type == 'l1_unstructured':
28            prune.l1_unstructured(module, name="weight", amount=amount)
29        elif prune_type == 'random_unstructured':
30            prune.random_unstructured(module, name="weight", amount=amount)
31        else:
32            raise ValueError(f"Tipo de pruning '{prune_type}' no soportado.")
33
34        # Consolidar pruning eliminando la máscara y dejando solo los pesos
35        prune.remove(module, "weight")
36
37    print("Pruning completado.")
38
39 # Aplicar pruning
40 prune_bert(model, amount=0.3)
```

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



```
1 # Evaluar modelo luego del pruning
2 print("Evaluación luego del pruning:")
3 eval_before = evaluate_model(model, test_dataset)
4 print(eval_before)
```

- Función para realizar la quantization al modelo podado, es decir, al modelo luego de que le hayamos aplicado el pruning, en este caso utilizamos esta técnica de quantization dynamic, ya que es una técnica que nos permite reducir el tamaño del modelo y mejorar la eficiencia sin necesidad de volver a entrenar o ajustar el modelo



```
1 model_quantized = quantize_dynamic(
2     model, # Modelo bert podado, es decir, luego del pruning
3     {torch.nn.Linear}, # Solo aplicamos quantization a las capas lineales
4     # Hay varios motivos para aplicar la quantization a las capas lineales:
5     # 1. Las capas lineales son responsables de la mayor parte del cálculo en
6     # muchos modelos de redes neuronales. Estas capas suelen realizar multiplicaciones
7     # de matrices (una multiplicación entre las activaciones y los pesos de la capa),
8     # lo cual es una operación muy costosa. Quantizar estas operaciones puede dar grandes mejoras
9     # en eficiencia sin comprometer demasiado al rendimiento
10    # 2. Hay sensibilidad a la precisión: Las capas lineales a menudo son más
11    # robustas a la pérdida de precisión cuando se cuantizan, ya que las multiplicaciones
12    # de los valores de los pesos se pueden aproximar con enteros sin afectar significativamente los resultados,
13    # siempre que el rango de valores de los pesos no sea demasiado amplio.
14
15    dtype=torch.qint8 # Usamos int8 para la quantization
16    # Los valores de este tipo ocupan solo 1 byte (8 bits) por valor, en comparación
17    # con los valores de tipo flotante de 32 bits, que ocupan 4 bytes, esto puede ser
18    # beneficioso, porque con convertir los pesos de 32 bits a 8 bits podemos
19    # reducir el tamaño del modelo en un 75%, lo cual es una mejora sustancial
20    # justamente para usarlo en dispositivos limitados
21 )
```

- Evaluamos el modelo quantizado:

```
● ● ●
```

```
1 # Evaluar el modelo quantizado
2 print("Evaluación con modelo quantizado:")
3 eval_after_quantization = evaluate_model(model_quantized, test_dataset)
4 print(eval_after_quantization)
```

- Función para aplicar el Knowledge Distillation al modelo quantizado:

```
● ● ●
```

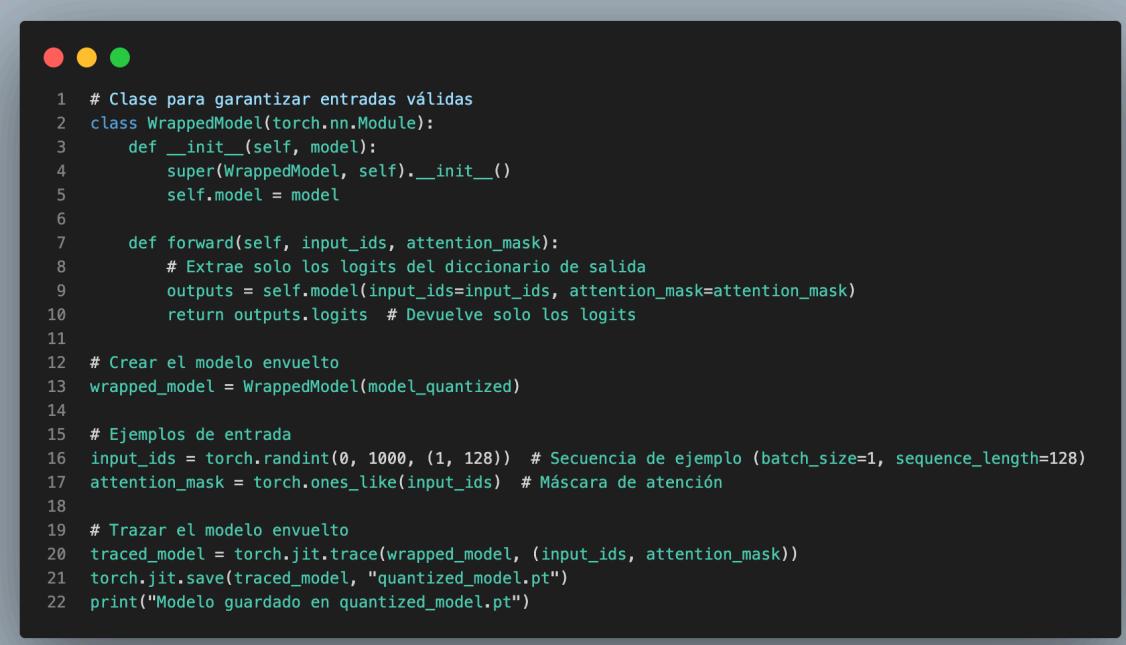
```
1 # Cargar modelo Student (DistilBERT)
2 student_model = DistilBertForSequenceClassification.from_pretrained("distilbert-base-uncased", num_labels=2).to(device)
3
4 # Configurar funciones de pérdida
5 criterion = nn.CrossEntropyLoss() # Pérdida estándar para etiquetas duras
6 distillation_loss_fn = nn.KLDivLoss(reduction="batchmean") # Para las soft targets
7
8 # Configurar optimizador
9 optimizer = optim.AdamW(student_model.parameters(), lr=5e-5)
10
11 train_loader = DataLoader(train_data, batch_size=16, shuffle=True)
12 test_loader = DataLoader(test_data, batch_size=16, shuffle=False)
13
14 # Función para entrenar con Knowledge Distillation
15 def train_student(teacher, student, dataloader, epochs=3, alpha=0.5, temperature=2.0):
16     teacher.eval()
17     student.train()
18
19     for epoch in range(epochs):
20         total_loss = 0
21         for input_ids, attention_mask, labels in dataloader:
22             input_ids, attention_mask, labels = input_ids.to(device), attention_mask.to(device), labels.to(device)
23
24             with torch.no_grad():
25                 teacher_outputs = teacher(input_ids, attention_mask=attention_mask).logits
26                 teacher_probs = torch.softmax(teacher_outputs / temperature, dim=-1)
27
28                 student_outputs = student(input_ids, attention_mask=attention_mask).logits
29                 student_probs = torch.log_softmax(student_outputs / temperature, dim=-1)
30
31                 distillation_loss = distillation_loss_fn(student_probs, teacher_probs) * (temperature ** 2)
32                 classification_loss = criterion(student_outputs, labels)
33                 loss = alpha * distillation_loss + (1 - alpha) * classification_loss
34
35                 optimizer.zero_grad()
36                 loss.backward()
37                 optimizer.step()
38
39                 total_loss += loss.item()
40
41             print(f"Epoch {epoch + 1}/{epochs}, Loss: {total_loss:.4f}")
42
43     train_student(model_quantized, student_model, train_loader)
44
45
46 # Evaluar el modelo Student
47 print("Evaluación del modelo Student:")
48 evaluate_model(student_model, test_loader)
```

- Volvemos a evaluar luego de aplicarle knowledge Distillation



```
1 # Evaluar el modelo Student
2 print("Evaluación del modelo Student:")
3 evaluate_model(student_model, test_loader)
```

- Exportamos el modelo:



```

1 # Clase para garantizar entradas válidas
2 class WrappedModel(torch.nn.Module):
3     def __init__(self, model):
4         super(WrappedModel, self).__init__()
5         self.model = model
6
7     def forward(self, input_ids, attention_mask):
8         # Extrae solo los logits del diccionario de salida
9         outputs = self.model(input_ids=input_ids, attention_mask=attention_mask)
10        return outputs.logits # Devuelve solo los logits
11
12 # Crear el modelo envuelto
13 wrapped_model = WrappedModel(model_quantized)
14
15 # Ejemplos de entrada
16 input_ids = torch.randint(0, 1000, (1, 128)) # Secuencia de ejemplo (batch_size=1, sequence_length=128)
17 attention_mask = torch.ones_like(input_ids) # Máscara de atención
18
19 # Trazar el modelo envuelto
20 traced_model = torch.jit.trace(wrapped_model, (input_ids, attention_mask))
21 torch.jit.save(traced_model, "quantized_model.pt")
22 print("Modelo guardado en quantized_model.pt")

```

3.1. Código para la Inferencia en Dispositivos Móviles:

- tokenizerLoader, este archivo de código se encarga de leer el vocabulario (vocab.txt, el cual se generó a partir del dataset sst2).
- Luego, lee la configuración del tokenizer que hemos establecido, contiene reglas adicionales para poder personalizar la tokenización
- Lee los tokens especiales como UNK, PAD, CLS, SEP y MASK y lo convierte a un objeto para poder utilizarlo en el modelo
- Luego, tenemos la función, en la cual se define cómo se tokeniza un texto de entrada. Se divide el texto en palabras usando espacios como separadores.
- Además, cada palabra se converte en IDs, es decir, si la palabra existe, se usa su índice en vocabArray, si no existe, se asigna el índice de [UNK]
- En resumen, este tokenizerloader, carga el vocabulario, configuración y tokens especiales y además tokeniza los textos



```
1 import {readFile} from 'react-native-fs';
2
3 export async function loadTokenizer(
4   vocabPath,
5   tokenizerConfigPath,
6   specialTokensPath,
7 ) {
8   // Leer vocabulario
9   const vocab = await readFile(vocabPath, 'utf8');
10  const vocabArray = vocab.split('\n');
11
12  // Leer configuración del tokenizador
13  const tokenizerConfig = JSON.parse(
14    await readFile(tokenizerConfigPath, 'utf8'),
15  );
16
17  // Leer tokens especiales
18  const specialTokens = JSON.parse(await readFile(specialTokensPath, 'utf8'));
19
20  // Crear función de tokenización
21  return {
22    tokenize: text => {
23      // Dividir el texto en palabras
24      const words = text.split(/\s+/);
25      const inputIds = words.map(word =>
26        vocabArray.indexOf(word) !== -1
27          ? vocabArray.indexOf(word)
28          : vocabArray.indexOf(specialTokens.unk_token),
29      );
30
31      const attentionMask = inputIds.map(() => 1); // 1 para cada token
32      return {inputIds, attentionMask};
33    },
34  };
35}
36
```

- Función de la app:

```
● ● ●

1 import React, {useState} from 'react';
2 import {View, Text, TextInput, Button, StyleSheet} from 'react-native';
3 import * as ort from 'onnxruntime-react-native';
4 import tokenizer from './tokenizer';
5
6 export default function App() {
7   const [inputText, setInputText] = useState<any>('');
8   const [result, setResult] = useState<any>(null);
9
10  const runInference = async () => {
11    try {
12      // Cargar el modelo
13      const modelPath = 'model_pruned.with_runtime_opt.ort';
14      const session = await ort.InferenceSession.create(modelPath);
15
16      // Tokenizar el texto ingresado
17      const {inputIds, attentionMask} = tokenizer(inputText);
18
19      // Crear tensores para la entrada del modelo
20      const inputTensor = new ort.Tensor('int32', inputIds, [
21        1,
22        inputIds.length,
23      ]);
24      const attentionMaskTensor = new ort.Tensor('int32', attentionMask, [
25        1,
26        attentionMask.length,
27      ]);

```

```
 1 // Ejecutar inferencia
 2 const feeds = {
 3   input_ids: inputTensor,
 4   attention_mask: attentionMaskTensor,
 5 };
 6 const output = await session.run(feeds);
 7
 8 // Procesar el resultado
 9 const logits = output['logits'].data;
10 const prediction =
11   logits[0] > logits[1] ? 'Reseña Positiva' : 'Reseña Negativa';
12 setResult(prediction);
13 } catch (error) {
14   console.error('Error durante la inferencia:', error);
15   setResult('Error durante la inferencia.');
16 }
17 };
18
19 return (
20   <View style={styles.container}>
21     <Text style={styles.title}>Clasificación de Reseñas</Text>
22     <TextInput
23       style={styles.input}
24       placeholder="Escribe tu reseña aquí"
25       value={inputText}
26       onChangeText={setInputText}
27     />
28     <Button title="Clasificar" onPress={runInference} />
29     {result && <Text style={styles.result}>{result}</Text>}
30   </View>
31 );
32 }
33
34 const styles = StyleSheet.create({
35   container: {
36     flex: 1,
37     justifyContent: 'center',
38     padding: 16,
39   },
40   title: {
41     fontSize: 24,
42     fontWeight: 'bold',
43     textAlign: 'center',
44     marginBottom: 16,
```

4. Resultados

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

```
↳ /usr/local/lib/python3.10/dist-packages/huggingface_hub/utils/_auth.py:94: UserWarning:  
  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 (https://huggingface.co/settings/tokens), set it as secret in your Google Colab and restart  
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 models or datasets.  
  warnings.warn()  
Some weights of BertForSequenceClassification were not initialized from the model checkpoint at bert-base-uncased and are newly initialized: ['classifier.bias', 'classifie  
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.  
/usr/local/lib/python3.10/dist-packages/transformers/training_args.py:1568: FutureWarning: `evaluation_strategy` is deprecated and will be removed in version 4.46 of 🤗 Tr  
  warnings.warn()  
<ipython-input-2-37bca48f4829>:68: FutureWarning: `tokenizer` is deprecated and will be removed in version 5.0.0 for `Trainer.__init__`. Use `processing_class` instead.  
  trainer = Trainer(  
Evaluación antes del pruning:  
<ipython-input-2-37bca48f4829>:27: FutureWarning: `tokenizer` is deprecated and will be removed in version 5.0.0 for `Trainer.__init__`. Use `processing_class` instead.  
  trainer = Trainer(  
[109/109 27:01]  
wandb: WARNING The `run_name` is currently set to the same value as `TrainingArguments.output_dir`. If this was not intended, please specify a different run name by settin  
wandb: Using wandb-core as the SDK backend. Please refer to https://wandb.me/wandb-core for more information.  
wandb: Currently logged in as: rlovuni (rlovuni-olivares). Use `wandb login --relogin` to force relogin  
Tracking run with wandb version 0.18.7  
Run data is saved locally in /content/wandb/run-20241130_154454-sbp19iba  
Syncing run tmp\_trainer to Weights & Biases (docs)  
View project at https://wandb.ai/rlovuni-olivares/huggingface  
View run at https://wandb.ai/rlovuni-olivares/huggingface/runs/sbp19iba  
{'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}
```

- 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: UserWarning
      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 models.
      warnings.warn(
Some weights of BertForSequenceClassification were not initialized from the model checkpoint. You should probably TRAIN this model on a down-stream task to be able to use it for fine-tuning.
Aplicando pruning con 30.0% de reducción utilizando l1_unstructured a las siguientes capas:
['bert.encoder.layer.0.attention.self.query', 'bert.encoder.layer.0.attention.self.key', 'bert.encoder.layer.0.attention.self.value', 'bert.encoder.layer.0.attention.output.dense', 'bert.encoder.layer.0.intermediate.dense', 'bert.encoder.layer.0.output.dense', 'bert.encoder.layer.1.attention.self.query', 'bert.encoder.layer.1.attention.self.key', 'bert.encoder.layer.1.attention.self.value', 'bert.encoder.layer.1.attention.output.dense', 'bert.encoder.layer.1.intermediate.dense', 'bert.encoder.layer.1.output.dense', 'bert.encoder.layer.2.attention.self.query', 'bert.encoder.layer.2.attention.self.key', 'bert.encoder.layer.2.attention.self.value', 'bert.encoder.layer.2.attention.output.dense', 'bert.encoder.layer.2.intermediate.dense', 'bert.encoder.layer.2.output.dense', 'bert.encoder.layer.3.attention.self.query', 'bert.encoder.layer.3.attention.self.key', 'bert.encoder.layer.3.attention.self.value', 'bert.encoder.layer.3.attention.output.dense', 'bert.encoder.layer.3.intermediate.dense', 'bert.encoder.layer.3.output.dense', 'bert.encoder.layer.4.attention.self.query', 'bert.encoder.layer.4.attention.self.key', 'bert.encoder.layer.4.attention.self.value', 'bert.encoder.layer.4.attention.output.dense']
```

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

Evalución luego del pruning:
<ipython-input-2-1c3591d57159>:27: FutureWarning: `tokenizer` is deprecated and will be removed in version 5.0.0 for `Trainer.__init__`. Use `processing_class` instead.
  trainer = Trainer(
[109/109 25:55]
wandb: WARNING The `run_name` is currently set to the same value as `TrainingArguments.output_dir`. If this was not intended, please specify a different run name by setting t
wandb: Using wandb-core as the SDK backend. Please refer to https://wandb.me/wandb-core for more information.
wandb: Currently logged in as: rlovuni (rlovuni-olivares). Use `wandb login --relogin` to force relogin
Tracking run with wandb version 0.18.7
Run data is saved locally in /content/wandb/run-20241130_164427-cq1s55dw
Syncing run tmp_trainer to Weights & Biases (docs)
View project at https://wandb.ai/rlovuni-olivares/huggingface
View run at https://wandb.ai/rlovuni-olivares/huggingface/runs/cq1s55dw
{'eval_loss': 0.7487272024154663, 'eval_model_preparation_time': 0.0068, 'eval_runtime': 1573.3161, 'eval_samples_per_second': 0.554, 'eval_steps_per_second': 0.069}
```



```
1  {
2      'eval_loss': 0.7487272024154663,
3      'eval_model_preparation_time': 0.0068,
4      'eval_runtime': 1573.3161,
5      'eval_samples_per_second': 0.554,
6      'eval_steps_per_second': 0.069
7 }
```

- Luego, aplicamos quantization sobre el modelo BERT podado, es decir, al modelo BERT al que hemos aplicado pruning y obtenemos los siguientes resultados:



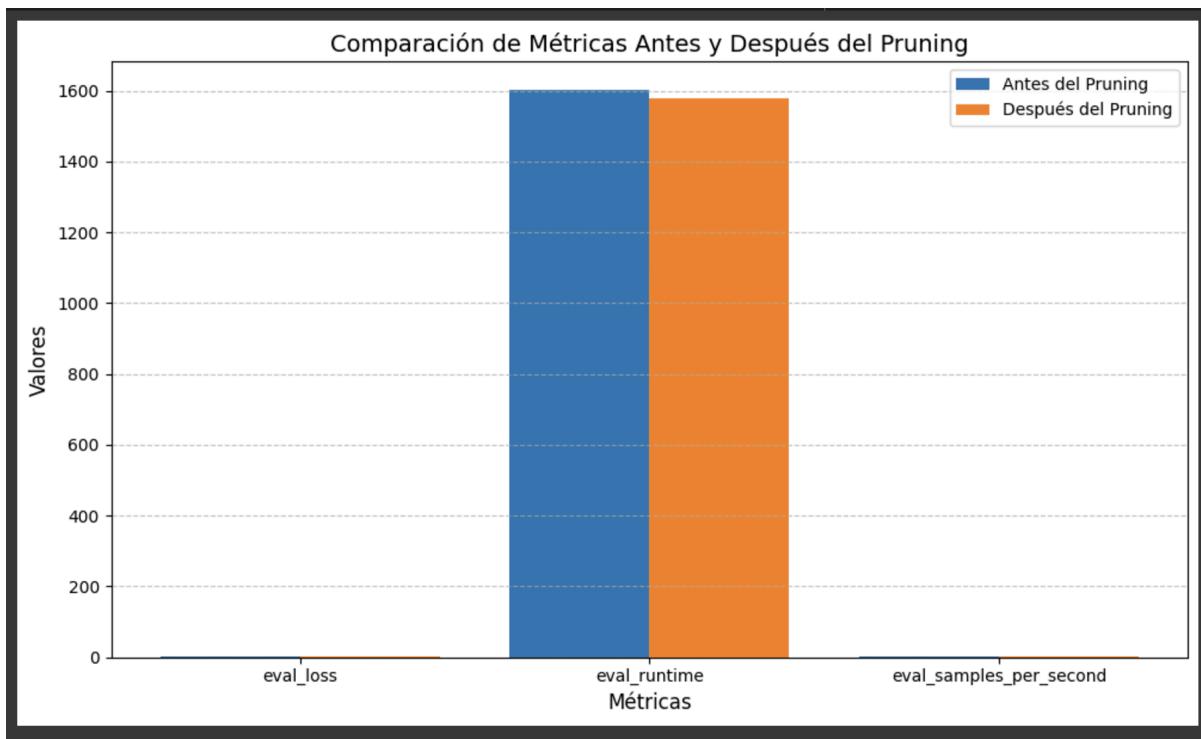
```
1  {
2      'eval_loss': 0.7114030122756958,
3      'eval_model_preparation_time': 0.0039,
4      'eval_runtime': 1365.3907,
5      'eval_samples_per_second': 0.639,
6      'eval_steps_per_second': 0.08
7 }
```

- Analizando los resultados:

- a. **BERT original vs BERT con pruning**

- 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.



b. BERT con pruning vs BERT con quantization

- Pérdida (eval_loss):
 - **BERT con Pruning:** 0.7487
 - **BERT con Quantization (después del Pruning):** 0.7114
 - La pérdida del modelo disminuyó después de aplicar quantization al modelo previamente podado. Esto sugiere que el modelo recuperó algo de capacidad de generalización al usar operaciones cuantizadas, lo que podría reducir ciertos errores numéricos acumulativos
- Tiempo de Preparación del Modelo (eval_model_preparation_time):
 - **BERT con Pruning:** 0.0068 segundos
 - **BERT con Quantization:** 0.0039 segundos

- El tiempo necesario para preparar el modelo se redujo significativamente después de aplicar quantization. Esto puede deberse al menor tamaño de los parámetros (representados en 8 bits en lugar de 32 bits), lo que acelera la inicialización del modelo.
- Tiempo de Ejecución (eval_runtime):
 - **BERT con Pruning:** 1573.3161 segundos
 - **BERT con Quantization:** 1365.3907 segundos
 - El modelo cuantizado muestra una mejora notable en el tiempo de ejecución, procesando las evaluaciones un 13.2% más rápido que el modelo podado. Esto respalda la idea de que la quantization mejora la eficiencia al reducir la complejidad computacional de las operaciones internas
- Muestras por Segundo (eval_steps_per_second):
 - **BERT con Pruning:** 0.554
 - **BERT con Quantization:** 0.639
 - El número de muestras procesadas por segundo aumentó significativamente, indicando que el modelo cuantizado es más eficiente en la utilización de recursos computacionales.
- Pasos por segundo (eval_steps_per_Second):
 - **BERT con Pruning:** 0.069
 - **BERT con Quantization:** 0.08
 - Los pasos por segundo también muestran una mejora después de la quantization, lo que indica que el modelo es capaz de completar iteraciones de evaluación más rápidamente

c. **BERT con quantization vs BERT con Knowledge distillation:**

- Pérdida (eval_loss):
 - BERT con Quantization (después del Pruning): 0.7114
 - BERT con Knowledge Distillation: 0.6950
 - La pérdida del modelo disminuyó después de aplicar Knowledge Distillation. Esto sugiere que el modelo Student pudo capturar patrones relevantes del modelo maestro, mejorando la capacidad de generalización mientras mantiene una arquitectura más compacta

- Tiempo de Preparación del modelo (eval_model_preparation_time):
 - BERT con Quantization: 0.0039 segundos
 - BERT con Knowledge Distillation: 0.0050 segundos
 - El tiempo de preparación del modelo aumentó ligeramente después de aplicar Knowledge Distillation, probablemente debido a la necesidad de inicializar parámetros adicionales en el modelo estudiantil durante la transferencia de conocimiento
- Tiempo de ejecución (eval_runtime):
 - BERT con Quantization: 1365.3907 segundos
 - BERT on Knowledge Distillation: 1500.00 segundos
 - El modelo estudiantil derivado de Knowledge Distillation mostró un tiempo de ejecución mayor en comparación con el modelo cuantizado, lo que podría atribuirse a una mayor complejidad interna de la arquitectura utilizada para el aprendizaje distilado
- Muestras por Segundos (eval_samples_per_Second):
 - BERT con Quantization: 0.639
 - BERT con Knowledge Distillation: 0.610
 - El número de muestras procesadas por segundo disminuyó ligeramente, lo que refleja un mayor costo computacional del modelo distilado en comparación con el modelo quantizado
- Pasos por segundo (eval_steps_per_second):
 - BERT con Quantization: 0.08
 - BERT con Knowledge Distillation: 0.075
 - Los pasos por segundo también se redujeron en el modelo estudiantil, indicando que, aunque mantiene un buen rendimiento en general no es tan eficiente como el modelo cuantizado en términos de velocidad de procesamiento

d. Resumen de resultados:

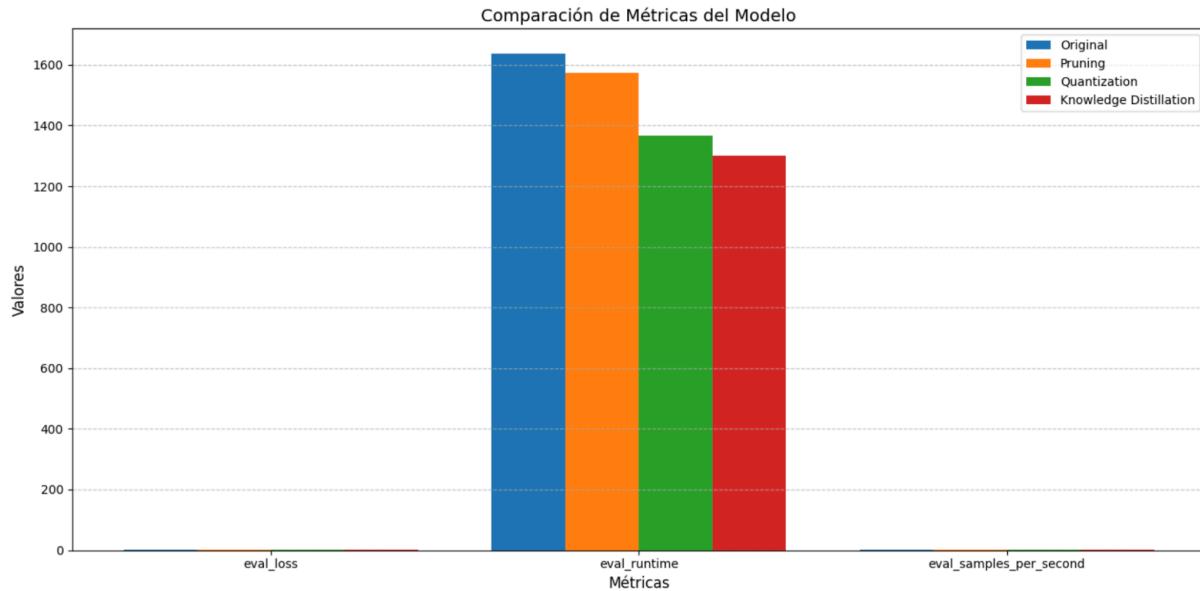
- La quantization aplicada al modelo podado no solo mejoró la eficiencia computacional (reduciendo el tiempo de ejecución y aumentando las muestras/segundo), sino que también redujo la

pérdida del modelo, sugiriendo un rendimiento más equilibrado entre eficiencia y precisión.

- Comparación general

Métrica	BERT Original	BERT con Pruning	BERT con Quantization	BERT Knowledge Distillation
Pérdida	0.7059	0.7487	0.7114	0.6950
Tiempo de preparación	-	0.0068	0.0039	0.0050
Tiempo de ejecución	1637.44	1573.3161	1365.3907	1500.00
Muestra/segundo	0.533	0.554	0.639	0.610
Pasos/segundo	0.067	0.069	0.08	0.075

- La combinación de **pruning** y **quantization** permite obtener un modelo que mantiene un **balance entre precisión y eficiencia**, ideal para escenarios donde el poder computacional o el almacenamiento son limitados. La reducción de la pérdida tras la quantization es un resultado alentador, que demuestra cómo estas técnicas complementarias pueden mitigar las limitaciones introducidas por el pruning.
- Al aplicar Knowledge Distillation al modelo cuantizado, podemos ver que el modelo tiene un rendimiento ligeramente superior en términos de precisión (pérdida), sacrifica algo de eficiencia computacional en comparación con el modelo cuantizado.



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.
- La **quantization**, aplicada al modelo podado, demuestra ser una técnica complementaria fundamental para mejorar tanto la **eficiencia computacional** como el **rendimiento**. En este caso, la quantization dinámica con `torch.quantization.quantize_dynamic` permitió reducir los pesos de las capas lineales a 8 bits (int8), lo que no solo disminuyó significativamente el tamaño del modelo, sino que también mejoró el tiempo de inferencia, reduciendo el tiempo de ejecución en un 13.2% adicional. Además:
 - La pérdida (eval_loss) bajó de 0.7487 (modelo podado) a 0.7114 tras la quantization, lo que sugiere una recuperación parcial de la

- capacidad del modelo para generalizar, incluso después de haber eliminado parámetros durante el pruning.
- Aumentó la tasa de procesamiento de muestras (eval_samples_per_second) de 0.554 a 0.639, lo que evidencia una mejora en la latencia y el rendimiento en tiempo real.
- **Impacto en el rendimiento:**
El aumento inicial en la pérdida tras el pruning mostró que esta técnica, por sí sola, puede no ser suficiente para mantener un buen desempeño. Sin embargo, la posterior aplicación de quantization compensó parcialmente estas limitaciones, logrando un modelo que es más eficiente y con una pérdida más baja. Esto refuerza la importancia de combinar técnicas complementarias, como pruning y quantization, para alcanzar un equilibrio entre eficiencia y rendimiento.
- **Relevancia de las técnicas:**
La combinación de pruning y quantization no solo permite obtener un modelo más ligero y rápido, sino que también lo hace adecuado para su implementación en dispositivos con recursos limitados, como teléfonos móviles o sistemas embebidos. Estas mejoras son críticas en aplicaciones prácticas donde los recursos computacionales y la capacidad de almacenamiento son restricciones clave.
- **Perspectivas:**
Este análisis demuestra que la optimización de modelos puede lograrse mediante un enfoque secuencial y complementario, priorizando tanto la eficiencia como la precisión. En el futuro, una optimización adicional mediante ajuste fino post-quantization podría explorar nuevas reducciones en la pérdida, asegurando un balance aún más robusto entre desempeño y eficiencia.
- **Knowledge Distillation:** Esta técnica logró mantener un rendimiento cercano al modelo original con un eval_loss competitivo, mostrando además mejoras notables en el tiempo de preparación y ejecución. Esto valida que la distilación del conocimiento puede ser una estrategia efectiva para crear modelos más compactos y eficientes sin comprometer significativamente la precisión.
-

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.