Tu código implementa un pipeline completo para clasificación de sentimiento en texto usando un Transformer "desde cero" en Keras, con el dataset IMDB. Aquí va el desglose y opinión detallada:

**¿Qué hace tu código?**

1. **Carga y preprocesamiento de datos**
   * Usa el dataset IMDB de Keras, que es un clásico para clasificación binaria de sentimiento (positivo/negativo).
   * Limita el vocabulario a las 20,000 palabras más frecuentes y padea/recorta las reseñas a las primeras 200 palabras (tokens) .
2. **Capa personalizada de Token + Position Embedding**
   * La clase TokenAndPositionEmbedding suma dos embeddings: uno para las palabras (tokens) y otro para la posición de cada palabra en la secuencia. Esto es esencial para Transformers, que no tienen información secuencial inherente.
   * Usa el rango [0, maxlen) para las posiciones y suma al embedding de tokens.
3. **Bloque Transformer personalizado**
   * La clase TransformerBlock implementa un bloque clásico de Transformer:
     + MultiHeadAttention (auto-atención sobre la secuencia)
     + Normalización y residual
     + FeedForward (MLP), dropout y otra normalización con residual
   * El parámetro training=None en call asegura funcionamiento correcto en training/inference.
4. **Construcción del modelo**
   * Input → TokenAndPositionEmbedding → 4 bloques Transformer → GlobalAveragePooling1D (pro media sobre la secuencia) → Dropout + Dense → Dropout + Dense(sigmoid)
   * El output es una probabilidad entre 0 y 1, para clasificación binaria.
5. **Compilación, entrenamiento y evaluación**
   * Usa Adam y binary\_crossentropy.
   * Entrena el modelo y evalúa en test. Luego grafica accuracy y loss para train/val a lo largo de las epochs.

**¿Qué tan "Transformer" es?**

* Sí, es un Transformer real (aunque pequeño). Implementa atención multi-cabeza, normalizaciones y MLP como en los papers originales.
* No es un transformer pre-entrenado (como BERT), sino entrenado desde cero para esta tarea.
* La arquitectura es muy similar a la de modelos clásicos de NLP, pero tu embedding y bloques son totalmente customizables.

**Cosas positivas**

* El código es claro, modular y fácil de modificar.
* Implementa de forma explícita los componentes clave de un Transformer ("from scratch").
* El pipeline es reproducible y estándar.
* Usas buenas prácticas: residuals, normalization y dropout.

**Posibles mejoras/recomendaciones**

* Puedes probar más bloques Transformer (o menos), variar embed\_dim y ff\_dim para buscar mejor desempeño.
* En vez de GlobalAveragePooling, podrías intentar usar solo el embedding del primer token (como hace BERT con [CLS]) o un GlobalMaxPooling.
* El modelo es relativamente pequeño para problemas complejos, pero suficiente para experimentar.
* Para mejor rendimiento, podrías usar embeddings pre-entrenados o preprocesar el texto (en vez de ID de palabras).

**¿Qué aprende el modelo?** El modelo aprende a asociar secuencias de 200 palabras (representadas como IDs) con una etiqueta binaria (sentimiento positivo/negativo) usando atención (puede enfocar diferentes partes de la reseña) y representaciones contextuales profundas.

Mejoras:

En BERT, la práctica estándar es anteponer un token especial ([CLS]) al inicio de cada secuencia y usar solo su embedding (la primera posición de la salida de la red) para la clasificación final.  
En tu pipeline, puedes simular esto usando la primera posición del tensor de salida de tu bloque transformer (índice 0 en la dimensión de secuencia) en vez de hacer un pooling global.

Como tu dataset IMDB ya está tokenizado por ID y no incluye [CLS], puedes usar la primera posición (índice 0). Esto es equivalente si todas las secuencias tienen el mismo largo y el modelo aprende que la primera posición es la relevante para la tarea de clasificación.

En Keras, puedes usar una capa Lambda para extraer esa primera posición del tensor [batch, sequence, features] antes de las capas densas.

Vamos a ver como implementar algunas de las mejoras que sugieres ... como podemos sustituir GlobalAveragePooling1D por la alternativa que utilza BERT ?

¡Perfecto! Para simular el enfoque de BERT y usar el embedding del primer token (posición 0) como representación para la clasificación, solo tienes que reemplazar la capa GlobalAveragePooling1D() por una extracción de la primera posición de la secuencia.

**¿Cómo hacerlo en Keras?**  
Usa la capa Lambda para quedarte con el primer vector de cada secuencia (índice 0).

También vamos a incluir el token especial CLS.

**1. Reserva un ID para el token [CLS]**

Supón que tu vocabulario original es de 20,000 palabras (vocab\_size = 20000). El nuevo ID será CLS\_ID = vocab\_size y tu embedding debe ser de tamaño vocab\_size + 1.

**2. Modifica el preprocesamiento**

Antes de padear/recortar las secuencias, antepone el token [CLS] (con valor CLS\_ID) a cada secuencia, y luego padéala o recórtala a maxlen (200).

**3. Ajusta el embedding**

embedding\_layer = TokenAndPositionEmbedding(maxlen, vocab\_size\_with\_cls, embed\_dim)

**4. Usa el embedding del token [CLS] para clasificación**

En vez de GlobalAveragePooling1D, extrae el embedding de la primera posición:

x = Lambda(lambda t: t[:, 0, :])(x)

**5. El resto del modelo queda igual**

No necesitas cambiar la arquitectura de los bloques Transformer ni la salida.

Aquí tienes varias ideas para mejorar tu Transformer, tanto a nivel arquitectónico como de entrenamiento y datos:

**1. Regularización y prevención de overfitting**

* **Aumenta el Dropout:** Sube la tasa de dropout en los bloques transformer y/o después de pooling.
* **L2 regularización:** Agrega regularización L2 en las capas densas o embeddings.
* **EarlyStopping:** Usa EarlyStopping para evitar sobreentrenamiento.

**2. Mejorar el preprocesamiento y los embeddings**

* **Embeddings preentrenados:** En vez de embeddings aleatorios, usa embeddings como GloVe, FastText o Word2Vec (requiere tokenizar el texto original, no IDs).
* **Data Augmentation:** Introduce ruido o haz swap de palabras para aumentar la variedad del dataset.

**3. Ajustes arquitectónicos**

* **Más bloques Transformer:** Experimenta con el número de bloques (num\_transformer\_blocks), más bloques dan más capacidad pero pueden causar sobreajuste en datasets pequeños.
* **Aumenta el tamaño del embedding (**embed\_dim**) y la dimensión de la FFN (**ff\_dim**).**
* **Más cabezas de atención:** Prueba con más (num\_heads) si tienes suficiente capacidad de cómputo.
* **Residual connections en el embedding:** Añade una residual entre el embedding de entrada y el output del último bloque transformer (como algunos modelos recientes).
* **LayerNorm en la entrada:** Añade una capa LayerNormalization justo después del embedding.

**4. Optimización y entrenamiento**

* **Learning Rate Schedule:** Usa un learning rate scheduler (por ejemplo, reduce el LR si la validación no mejora).
* **Warmup:** Prueba un warmup del learning rate al inicio del entrenamiento.
* **Batch size:** Experimenta con batch size más alto o más bajo.

**5. Técnicas avanzadas**

* **Preentrenar tu transformer:** Preentrena el modelo con una tarea tipo "language modeling" antes de afinarlo en clasificación (requiere mucho cómputo y datos).
* **Multi-Task Learning:** Si tienes acceso a otras tareas de texto, entrena el modelo en varias tareas a la vez.
* **Ensemble:** Combina la salida de varios modelos (distintos seeds, estructuras o hiperparámetros) para mayor robustez.

**6. Otros trucos de NLP**

* **Más contexto:** Aumenta maxlen si tu GPU/memoria lo permite, para dar más contexto al modelo.
* **Ponderación de clases:** Si los datos están desbalanceados, usa class\_weight en el fit.

**7. Evaluación y diagnóstico**

* **TensorBoard:** Usa TensorBoard para visualizar internamente la atención y el comportamiento del modelo.
* **Confusion Matrix:** Analiza los errores del modelo con una matriz de confusión.