Representación y creación del grafo de ejecución de LLAMA 2, ViT, Whisper y DeepSeek

Manuel Bojorge Araya
Instituto Tecnológico de Costa Rica
mbojorge@estudiantec.cr

***Abstract* — Este informe presenta un análisis comparativo de la representación y creación de grafos de ejecución en cuatro arquitecturas de modelos de lenguaje de gran tamaño: LLAMA 2, ViT, Whisper y DeepSeek. Se exploran sus semejanzas, diferencias y las oportunidades para el reuso de código, destacando estrategias para abordar sus particularidades pensando su integración en hardware para computación de alto rendimiento.**

Palabras clave: LLM, GGML, MoE, MLA, RMS norm, FFN, Transformer.

I. INTRODUCCIÓN

Los modelos LLAMA 2, DeepSeek, ViT y Whisper han revolucionado los campos del procesamiento de lenguaje natural (NLP), visión por computadora y reconocimiento de voz. Sin embargo, la falta de un estándar en la representación de sus grafos de ejecución ha generado fragmentación en sus implementaciones.

Para este trabajo se utiliza GGML, la cuál es una biblioteca de aprendizaje automático, cuyos detalles se pueden consultar en introducción a GGML.

Estos modelos están basados en transformadores, pero no siguen la arquitectura original del transformador, sino que tienen sus variaciones. Antes de analizar a detalle cada modelo, es válido hacer un breve análisis del transformador original el cual se muestra en la figura 1. Este cuenta con dos bloques principales: encoder y decoder.

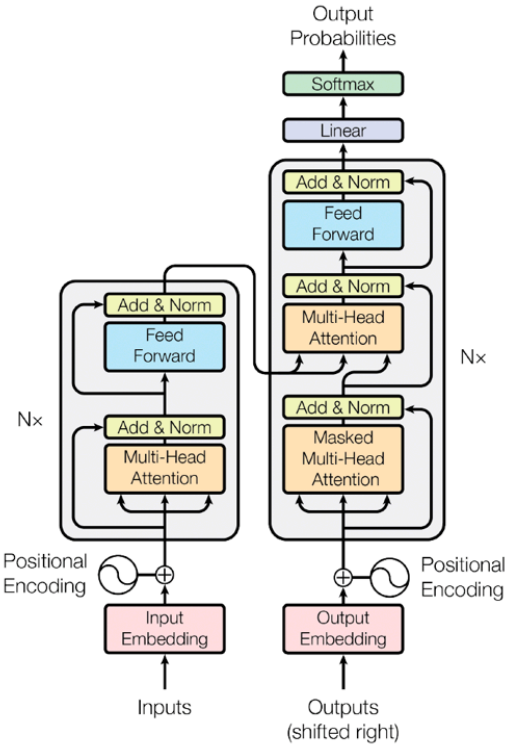

Figura 1. Transformador propuesto en el paper ”Attention is all you need”.

Encoder: Procesa las entradas y genera una representación codificada de ellas. Está compuesto por múltiples capas, cada una de ellas contiene:

• Una subcapa de Multi-Head Attention.

• Una red de Feed Forward.

• Normalización de capa (Add & Norm) después de cada subcapa.

Decoder: Genera las salidas basándose en las representaciones codificadas. Es similar al encoder, pero incluye una subcapa adicional de atención multi-cabezal enmascarada (Masked Multi-Head Attention) para procesar las salidas anteriores.

En cuanto a los embeddings y positional encodings, las entradas y salidas se transforman en vectores mediante embeddings y se suman codificaciones posicionales para incorporar información sobre el orden de los elementos en las secuencias.

Teniendo esto en cuenta, ahora se resume cada modelo por separado.

A. LLAMA 2

Es un encoder modificado que introduce varias modificaciones en comparación con la arquitectura original del transformador:

• Utiliza la función de activación SwiGLU en lugar de ReLU en la red Feed Forward.

• Aplica normalización antes de los bloques de atención y Feed Forward, en contraste con la normalización posterior en la arquitectura original.

• Emplea RMSNorm en lugar de la normalización estándar (Add & Norm).

Su arquitectura se muestra en la figura 2.

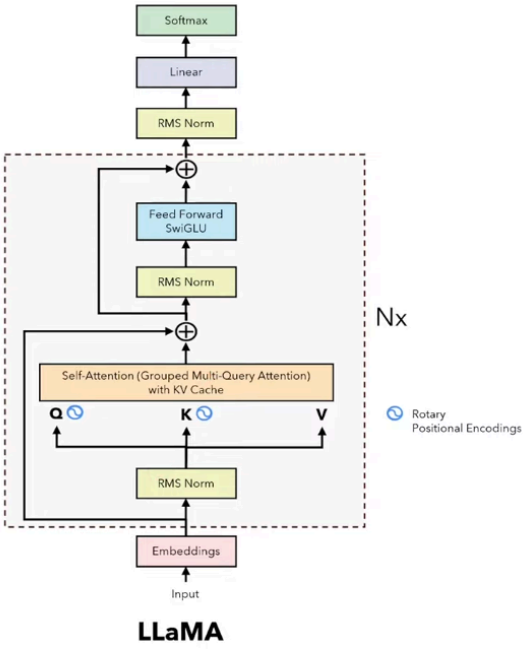

Figura 2. Arquitectura de Llama.

B. ViT (Vision Transformer)

Es un encoder modificado diseñado para tareas de visión por computadora. En lugar de procesar imágenes como matrices de pixeles completas, ViT las divide en parches más pequeños, los cuales son transformados en representaciones numéricas mediante embeddings. Luego, estos parches son procesados con un mecanismo de autoatención similar al de los transformadores en NLP, permitiendo capturar relaciones espaciales a diferentes escalas.

Su arquitectura se muestra en la figura 3.

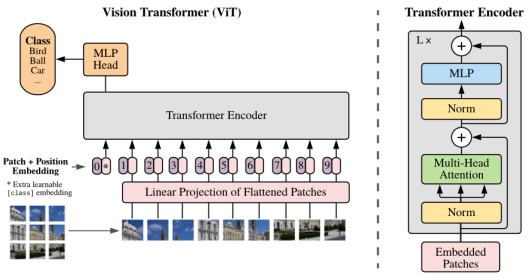

Figura 3. Arquitectura de ViT.

C. Whisper

Whisper es un modelo de reconocimiento de voz desarrollado por OpenAI, diseñado para convertir audio en texto. Su proceso comienza con la digitalización del audio, donde una onda sonora continua se convierte en valores discretos para su procesamiento en dispositivos digitales. El modelo opera con varias capas encoder-decoder:

• El encoder extrae características relevantes del audio procesado.

• El decoder genera la transcripción en texto basada en las representaciones codificadas.

Whisper puede integrarse con llama.cpp para generar respuestas automáticas basadas en el texto transcrito. Además, en sistemas Linux, herramientas como Piper pueden convertir la salida de llama.cpp nuevamente en voz.

Su arquitectura se muestra en la figura 4 y el proceso de transcripción en la figura 5.

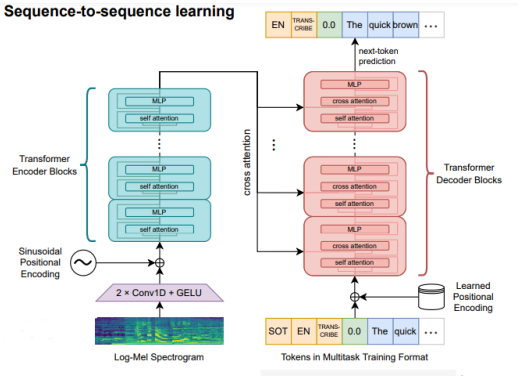

Figura 4. Arquitectura de Whisper.

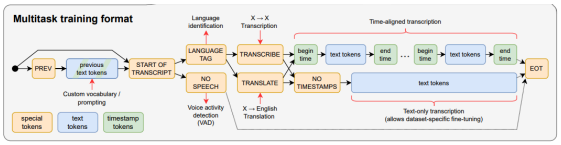

Figura 5. Proceso de transcripción de Whisper

D. DeepSeek

DeepSeek es una empresa enfocada en el desarrollo de modelos de lenguaje de código abierto. Su primer modelo, DeepSeek Coder estaba diseñado específicamente para tareas relacionadas con la codificación. Posteriormente se lanzó DeepSeek LLM, su primer modelo de propósito general. Luego se lanzó DeepSeek-V2, la segunda versión de su modelo de lenguaje general.

DeepSeek ha introducido varias mejoras significativas en la arquitectura clásica de los transformadores, optimizando tanto el rendimiento como la eficiencia computacional. Estas innovaciones se centran en tres áreas principales: Multi-Head Latent Attention (MLA), Mixture of Experts (MoE) y Predicción Multi-Token

En el caso de MLA, el problema del KV cache en transformadores clásicos, que consume grandes cantidades de memoria en contextos largos, se resuelve mediante la compresión de bajo rango de las matrices de claves y valores. Esto reduce el tamaño del KV cache a un 5%-13% del original, permitiendo una inferencia más eficiente sin sacrificar la calidad del modelo. Además, MLA fusiona las multiplicaciones de matrices necesarias para reconstruir las claves y valores con las proyecciones de consulta y post-atención, optimizando aún más el rendimiento.

Por otro lado, DeepSeek aborda los problemas tradicionales de los modelos Mixture of Experts (MoE), como el colapso del enrutamiento y la pérdida de información común, mediante un balanceo de carga dinámico sin pérdidas auxiliares. Al dividir los expertos en compartidos y enrutados, se asegura un uso equilibrado mientras se mantiene la especialización. Finalmente, la Predicción Multi-Token permite generar varios tokens en un solo paso forward, acelerando la inferencia y facilitando la decodificación especulativa

Su arquitectura se compone por varios bloques de transformador, su arquitectura se muestra en las figuras 6, 7 y 8

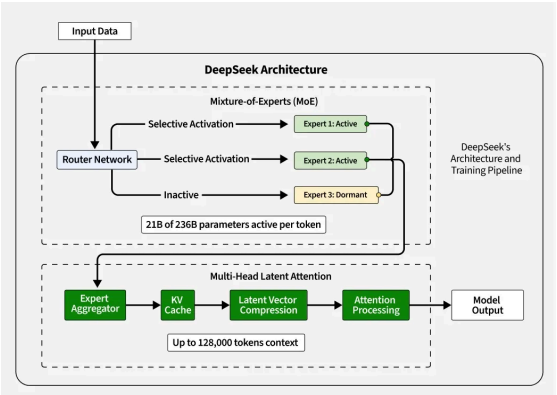

Figura 6. Arquitectura de DeepSeek.

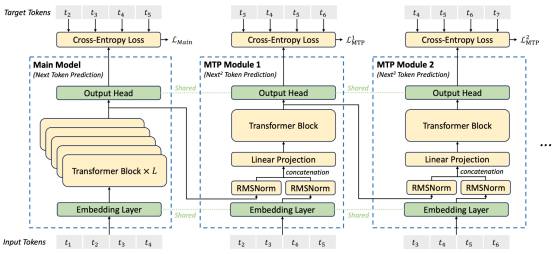

Figura 7. Arquitectura de DeepSeek.

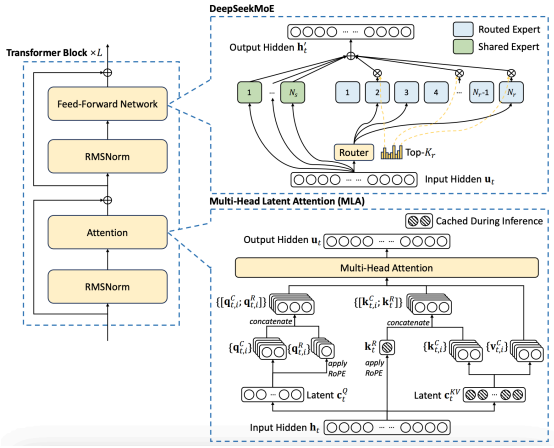

Figura 8. Arquitectura de DeepSeek.

II. CONSTRUCCIÓN DEL GRAFO

Para la creación del grafo y sus nodos en GGML, se emplean dos estructuras fundamentales:

• ggml_tensor: Representa un tensor n-dimensional para almacenar datos, metadatos y la relación con otros tensores en el grafo. [enlace al código fuente]

• ggml_cgraph: Define la estructura del grafo de cómputo, incluyendo los nodos, gradientes y hojas; lo necesario para la evaluación de las operaciones. [enlace al código fuente]

Estas estructuras permiten representar datos, operaciones y la topología del grafo de computación en GGML.

En el archivo llama.cpp, la función llama_build_graph() construye y retorna un grafo de cómputo (ggml_cgraph) en función del tipo de arquitectura del modelo. [enlace al código fuente]

La arquitectura del modelo se evalúa mediante un bloque switch-case, lo que permite definir el flujo de construcción del grafo según las características específicas del modelo.

Esta función recibe tres parámetros:

• llama_context & lctx [enlace al código fuente]: Es una referencia a una estructura llama_context, que contiene:

– El modelo (lctx.model).
– Parámetros de configuración (lctx.cparams).
– Un programador de tareas (lctx.sched).
– Backends de ejecución (lctx.backends).

• const llama_ubatch & ubatch [enlace al código fuente]: Es una referencia constante a una estructura llama_ubatch, que representa un lote de tokens a procesar. Gestiona múltiples secuencias de tokens en paralelo y contiene información sobre:

– La cantidad de tokens.
– La cantidad de tokens por secuencia.

• bool worst_case Es un valor booleano que indica si el grafo debe construirse en un escenario de peor caso. Este parámetro en el constructor llm_build_context [enlace al código fuente] se utiliza para ajustar ciertos valores del modelo, afectando:

– La cantidad de elementos en la memoria key-value (kv_self).
– El número de salidas (n_outputs y n_outputs_enc).
– La posición del cabezal en la memoria (kv_head).

A. LLAMA 2

La función build_llama() [enlace al código fuente] construye el grafo de cómputo en una estructura ggml_cgraph para un modelo basado en LLaMA

Esta función build_llama() hace lo siguiente:

• Inicializa la estructura:

Inicializa un grafo para almacenar todas las operaciones que se ejecutarán en el

modelo.
(struct ggml_cgraph * gf =
ggml_new_graph_custom(ctx0,
model.max_nodes(), false);)

• Define variables importantes:

Número de tokens procesados en la inferencia. (int32_t n_tokens =
this->n_tokens;)

Número de cabezales de atención
(const int64_t n_embd_head =
hparams.n_embd_head_v;)

Se verifica que los parámetros clave del modelo sean consistentes:
(GGML_ASSERT(n_embd_head ==
hparams.n_embd_head_k);)
(GGML_ASSERT(n_embd_head ==
hparams.n_rot);)

• Construcción de las capas del modelo:

- Capa de embeddings:

Convierte los tokens en vectores de embeddings.
(inpL = llm_build_inp_embd(ctx0,
lctx, hparams, ubatch, model.tok_embd, cb);)
La definición de esta función se puede revisar en el [enlace al código fuente]

- Máscaras y posiciones:

maneja las posiciones de los tokens en la secuencia (struct ggml_tensor * inp_pos =
build_inp_pos();)

Crea una máscara de atención en float32, por defecto es causal pero si se pasa el valor booleano 'false' se indica que es no causal.
(struct ggml_tensor * KQ_mask =
build_inp_KQ_mask();)

• Loop sobre las capas del modelo (n_layer):

- Normalización de entrada:

cur = llm_build_norm(ctx0, inpL, hparams,
model.layers[il].attn_norm, NULL,
LLM_NORM_RMS, cb, il);

- Self-Attention:

Se construyen las matrices Q (Consulta), K (Clave) y V (Valor). Se les aplica RoPE (Rotary Positional Embeddings) que ayuda a modelar el orden en las secuencias.

Se realiza la atención multi-cabezal con:
cur = llm_build_kv(ctx0, lctx, kv_self, gf,
model.layers[il].wo, model.layers[il].bo, Kcur,
Vcur, Qcur, KQ_mask, n_tokens, kv_head,
n_kv, kq_scale, cb, il);

- Se utiliza Mixture of Experts o Feed Forward Network:

Si el modelo no usa MoE, usa un FFN normal.
Se agrega el residual al output de la capa:
cur = ggml_add(ctx0, cur, ffn_inp);

• Capa final del modelo:
Aplica normalización.

- Expande el grafo de cómputo:

Con la función
ggml_build_forward_expand(gf,
cur);
la cual llama a otra función
ggml_build_forward_impl() con expand
= true [enlace al código fuente]. Esta última registra el número de nodos antes de la expansión. Añade recursivamente los tensores padres de tensor al grafo. Registra cuántos nodos nuevos se añadieron. Verifica que el tensor que se está agregando sea efectivamente el último nodo agregado.

Finalmente se retorna el grafo.

—----------

B. ViT

El grafo computacional para el modelo ViT se construye a partir del procesamiento de una imagen. La función vit_encode_image se utiliza para procesar la imagen de entrada y generar una predicción de clasificación. Esta función devuelve un grafo de ejecución (ggml_cgraph) que contiene todas las operaciones necesarias para realizar la predicción. [enlace al código fuente]

Esta función recibe los siguientes parámetros:

• const vit_model &model [enlace al código fuente]
Este parámetro contiene toda la información relacionada con el modelo ViT. El tipo vit_model incluye varios elementos como:

– hparams: Parámetros del modelo como tamaño de imagen, número de capas y número de cabezas de atención. [enlace al código fuente]

– enc_img: Es una estructura que contiene los elementos para procesar la imagen de entrada y transformarla en una representación que pueda ser entendida por las capas posteriores. [enlace al código fuente]

– classifier: La cabeza clasificadora del modelo, que es responsable de hacer las predicciones finales. Se conforma por tensores asociados a la capa de normalización y a la capa final (dense o fully connected) que almacenan pesos y sesgos (biases). [enlace al código fuente]

– ctx: Contexto de ggml que se usa para gestionar la memoria y los objetos utilizados durante la ejecución de operaciones en el modelo. [enlace al código fuente]

– tensors: Un mapa de tensores que permite manejar diferentes operaciones de tensores durante la inferencia. [enlace al código fuente]

• vit_state &state [enlace al código fuente]
Este parámetro contiene el estado de la ejecución del modelo, lo que incluye:

– prediction: Un tensor donde se almacenará la predicción final.

– ctx: Contexto de ggml, necesario para la gestión de memoria y ejecución de operaciones.

– work_buffer: Un buffer de trabajo utilizado durante las operaciones intermedias.

– buf_alloc_img_enc y buf_compute_img_enc: Buffers utilizados para almacenar datos de imágenes y para la codificación de la imagen.

– allocr: Un objeto para gestionar la asignación de memoria de los tensores.

• const image_f32 &img [enlace al código fuente]
Este parámetro es una estructura que contiene la imagen que se va a procesar y almacena:

– nx y ny: Las dimensiones de la imagen (ancho y alto).

– data: Un vector que almacena los valores de los píxeles de la imagen en formato de punto flotante (float)

A continuación se describe el flujo de operaciones:

1) Inicialización

Se cargan los hiper parámetros del modelo como tamaño de imagen, capas ocultas y cabezas de atención.

Se inicializa el contexto de ejecución (ggml context) para manejar memoria y operaciones de ggml.
Se inicializa el grafo computacional (ggml_cgraph) en donde se almacenarán todas las operaciones.

2) Procesamiento de la imagen

La imagen de entrada (img) se carga en un tensor 4D (inp) de tipo flotante de 32 bits. Se reserva la memoria necesaria para el tensor utilizando ggml allocr alloc.

3) Patch Embedding

Se realiza una convolución 2D (ggml_conv_2d_sk_p0) para obtener los patch embeddings aplicando pesos (proj_w) y sesgos (proj_b).
Se reorganizan las dimensiones del tensor para su posterior procesamiento.
Se añaden los positional embeddings (enc.pe) al tensor para conservar la información espacial de los parches.

4) Codificador Transformer (Bucle por capa)

– Normalización de capa (Norm 1):
Se normalizan las entradas y se aplican pesos y sesgos (norm1 w, norm1 b).

– Self-attention:
Se calculan las matrices Q, K, V.
Se realiza la multiplicación KQ, se escalan y se normalizan con softmax.
Se calcula la salida KQV, se reordena y se proyecta a la dimensión de salida.

– Conexión residual:
Se suma la entrada original de la capa al resultado de atención.

– Red feed-forward:
Normalización adicional (Norm 2).
Capa densa con activación GELU y proyección final.
Otra conexión residual.

5) Pooling y clasificación

Se extrae la representación del token de clase (posición 0).
Se normaliza y se proyecta la salida para preparar la clasificación.
Se proyecta el tensor con pesos (head_w) y sesgos (head_b).
Se aplica una función softmax para obtener probabilidades de clase.

6) Finalización

La salida (probs) se copia al tensor de predicciones del estado.
Se expande el grafo computacional para que incluya todas las operaciones necesarias para la inferencia.
Se libera la memoria del contexto.
Se retorna el grafo.

—----------

C. Whisper

Este modelo construye varios grafos por etapas o bloques para la convolución, encoder, cross attention y decoder.

a. Convolución

El grafo computacional para la etapa de convolución configura la representación de la entrada de audio y construye las primeras capas del modelo, aplicando convoluciones 1D seguidas de activaciones GELU. Devuelve un puntero a una estructura ggml_cgraph, que representa el grafo. [enlace al código fuente]

i. Parámetros de entrada

• wctx: El contexto del modelo de Whisper encapsula todo lo necesario para cargar y

preparar el modelo, configurar el entorno de ejecución (tipos de dato y parámetros) y manejar el estado de las inferencias. [enlace al código fuente]

• wstate: Gestiona tiempos y rendimiento, mantiene el estado del procesamiento desde la entrada del espectrograma mel hasta la salida en texto y optimiza la inferencia. [enlace al código fuente]

ii. Extracción de hiperparametros

• n_ctx: Longitud del contexto de audio. Se usa wstate.exp_n_audio_ctx si está definido, de lo contrario, se toma hparams.n_audio_ctx.

• n_state: Tamaño del estado de audio en la red (aunque no se usa directamente en la función).

• n_mels: Número de canales del espectrograma mel

iii. Inicialización del contexto

Se crea un contexto ggml_context con los siguientes parámetros:

• mem_size: Tamaño de la memoria reservada para el grafo, tomado desde wstate.sched_conv.meta.size().

• mem_buffer: Puntero a la memoria preasignada.

• no_alloc: true, lo que significa que no se asignará memoria adicional dinámicamente.

Se inicializa grafo:
ggml_cgraph * gf = ggml_new_graph(ctx0);

iv. Definición del tensor de entrada

Se define el tensor de entrada mel, que representa el espectrograma de entrada con:

• 2*n_ctx: El doble del contexto de audio.

• n_mels: Número de canales mel.

Se establece su nombre ("mel") y se marca como entrada del grafo con:
ggml_set_input(mel);

v. Se elige entre usar codificación externa y convolución

- Si se usa codificación externa, en lugar de la convolución, se hace lo siguiente:

• Se agrega mel al grafo:
ggml_build_forward_expand(gf, mel);

• Se crea un nuevo tensor cur de tamaño (n_state, n_ctx), que almacenará la salida de un codificador externo.

• Se marca este tensor como entrada:
ggml_set_input(cur);

• Se nombra "embd_enc" y se almacena en wstate.embd_enc.

- Si no se usa codificación externa, se aplican dos capas convolucionales con activaciones GELU:

1) Primera convolución 1D:

• Se aplica la convolución al tensor mel:
cur = ggml_conv_1d_ph(ctx0, model.e conv 1 w, mel, 1, 1);
• Se suma el sesgo:
cur = ggml_add(ctx0, cur, model.e conv_1_b);

• Se aplica la activación GELU:
cur = ggml gelu(ctx0, cur);

2) Segunda convolución 1D:

• Se toma la salida de la primera capa y se aplica otra convolución 1D:
cur = ggml_conv_1d_ph(ctx0, model.e_conv_2_w, cur, 2, 1);

• Se suma el sesgo:
cur = ggml_add(ctx0, cur, model.e_conv_2_b);

• Se vuelve a aplicar la activación GELU:
cur = ggml_gelu(ctx0, cur);
Se asigna el nombre "embd_conv" a la salida y se almacena en wstate.embd_conv.

vi. finalización del grafo

• Se marca cur como la salida del grafo:
ggml_set_output(cur);

• Se expande el grafo para incluir todas las operaciones necesarias:
ggml_buildforward_expand(gf, cur);

• Se libera la memoria del contexto:
ggml_free(ctx0);

• Se retorna el grafo gf.

—----------

b. ***Encoder***

El grafo computacional para la etapa de codificación configura las representaciones internas del modelo a partir de la entrada de audio preprocesada. Devuelve un puntero a una estructura ggml_cgraph, que representa el grafo. [enlace al código fuente]

i. Parámetros de entrada

• wctx: El contexto del modelo de Whisper encapsula todo lo necesario para cargar y preparar el modelo, configurar el entorno de ejecución (tipos de dato y parámetros) y manejar el estado de las inferencias. [enlace al código fuente]

• wstate: Gestiona tiempos y rendimiento, mantiene el estado del procesamiento desde la entrada en Mel hasta la salida en texto y optimiza la inferencia. [enlace al código fuente]

ii. Extracción de hiperparametros

• n_ctx: Longitud del contexto temporal de entrada.

• n_layer: Número total de capas en el encoder.

• n_state: Dimensión del espacio de representación para cada paso temporal.

• n_head: Número de cabezas en la atención multi-cabeza.

iii. Inicialización del contexto

Se inicializa un contexto ggml_context con:

• mem_size: Memoria reservada para el grafo, tomada desde wstate.sched_enc.meta.size().

• mem_buffer: Puntero a la memoria preasignada.

• no_alloc: true, evitando asignaciones de memoria adicionales.

iv. Definición del tensor de entrada

Se toma la salida de la capa convolucional embd_conv y se ajusta la escala para las operaciones de atención con const float KQscale = 1.0f/sqrtf(float(n_state_head))

La escala KQscale se usa más adelante para normalizar los valores Key-Query en la autoatención.

v. Positional encodings

Se añaden las codificaciones posicionales para preservar la información de orden y que el modelo entienda la secuencia temporal del audio

```
struct ggml_tensor * e_pe =
ggml_view_2d(ctx0, model.e_pe,
model.e_pe->ne[0], n_ctx, e_pe_stride,
e_pe_offset);

cur = ggml_add(ctx0, e_pe, ggml_cont(ctx0,
ggml_transpose(ctx0, cur)));
```

vi. Procesamiento de las capas

• Se itera por cada capa del encoder

• Se normaliza la entrada y se ajusta con pesos y sesgos aprendidos

• Se obtienen las matrices Q, K y V para el mecanismo de autoatención.

• Para el mecanismo de atención dependiendo de si flash attn está habilitado, se usa un enfoque optimizado o estándar para la atención.

– Con flash attn: Se usa un método más eficiente para grandes secuencias.
– Sin flash attn: Se realiza el producto K * Q, seguido de softmax y proyección con V.

vii. Proyección

Después de la atención, se realiza una proyección y se añade una conexión residual.

viii. Feed-Forward Network

Cada capa del encoder tiene una subred FFN. Se normaliza, pasa por capas lineales y se aplica activación GELU.

ix. Normalización final

Se normaliza la salida final del encoder antes de devolver el grafo

x. Finalización del grafo

Se expande el grafo con las operaciones construidas, se asigna el resultado, se libera la memoria del contexto y se retorna el grafo

—----------

c. ***Cross attention***

La función whisper_build_graph_cross construye el grafo de cómputo para la atención cruzada. Devuelve un puntero a una estructura ggml_cgraph, que representa el grafo. [enlace al código fuente]

i. Parámetros de entrada

• wctx: El contexto del modelo de Whisper encapsula todo lo necesario para cargar y preparar el modelo, configurar el entorno de ejecución (tipos de dato y parámetros) y manejar el estado de las inferencias. [enlace al código fuente]

• wstate: Gestiona tiempos y rendimiento, mantiene el estado del procesamiento desde la entrada en Mel hasta la salida en texto y optimiza la inferencia. [enlace al código fuente]

ii. Extracción de hiperparametros

• n_ctx: Longitud del contexto de audio a procesar. Usa un valor esperado (exp_n_audio_ctx) si se ha definido, o el predeterminado del modelo.

• n_state: Dimensión del espacio latente (estado interno) del modelo.

• n_head: Número de cabezas en la atención múltiple.

• n_state_head: Dimensión del estado por cada cabeza.

• n_ctx_pad: Ajuste del tamaño de contexto para alineación de memoria (a múltiplos de 256).

iii. Inicialización del contexto

Se crea un contexto ggml_context con los siguientes parámetros:

• mem size: Tamaño de la memoria reservada para el grafo, proveniente de wstate.sched_cross.meta.size().

• mem buffer: Puntero a la memoria preasignada.

• no_alloc: true, lo que significa que no se asignará memoria adicional dinámicamente.

Se inicializa el grafo utilizando el contexto: ggml_cgraph * gf = ggml_new_graph(ctx0);

iv. Carga del Embedding del Codificador

cur: Tensor que representa la salida del codificador (embedding del audio).

Kscale: Factor de escalado para las matrices clave
(Key).

v. Construcción de la atención cruzada por capa

Se recorre cada capa del decodificador para construir la operación de atención

• Cálculo de la Matriz Key (Kcross): con ggml_mul_mat se hace la multiplicación de matrices entre la entrada y los pesos de Key. Luego se hace un escalado.

• Cálculo de la Matriz Value (Vcross): Multiplicación: Aplica los pesos V al embedding. Sesgo (bias): Se a˜nade el vector de sesgo para la matriz Value.

vi. Gestión de Memoria para K y V

Dependiendo de si se usa atención rápida (flash_attn), se manejan las vistas de K y V de manera distinta:

• Con flash attn habilitado: Usa vistas 1D

• Sin flash attn: Realiza una transposición y ajuste en 2D para Vcross antes de almacenarlo. Define las vistas en 2D para v

vii. Construcción del Grafo

Se expanden las operaciones al grafo:

ggml_cpy: Copia los tensores Kcross y Vcross a los buffers correspondientes en la memoria del estado.

ggml_build_forward_expand: Añade estas operaciones al grafo para su ejecución.

Finalmente se retorna el grafo

—----------

d. Decoder

La función whisper_build_graph_decoder construye el grafo de cómputo para la fase de decodificación. [enlace al código fuente]

i. Parámetros de entrada

• wctx: El contexto del modelo de Whisper encapsula todo lo necesario para cargar y

preparar el modelo, configurar el entorno de ejecución (tipos de dato y parámetros) y manejar el estado de las inferencias. [enlace al código fuente]

• wstate: Gestiona tiempos y rendimiento, mantiene el estado del procesamiento desde la entrada en Mel hasta la salida en texto y optimiza la inferencia. [enlace al código fuente]

• batch: Referencia al batch de entrada con los tokens a procesar. Incluye el número total de tokens (n tokens). [enlace al código fuente]

• save_alignment_heads_QKs: Es un valor booleano que indica si se deben guardar los alignment heads durante el proceso de decodificación.

• worst_case: Es un valor booleano que controla si se debe considerar el peor caso para la dimensión del key-value cache (n_kv). Esto es útil para asignar memoria cuando hay recursos limitados.

ii. Inicialización del contexto

Se inicializa un contexto GGML (ctx0) con los parámetros del planificador de decodificación (sched_decode).
Se crea el grafo computacional (gf) con la función ggml_new_graph_custom.

iii. Preparación de tensores de entrada

embd y position: Representan los tokens y sus posiciones en la secuencia.

KQ_mask: Máscara para el mecanismo de atención, que previene que los tokens atiendan posiciones futuras. [enlace al código fuente]

iv. Codificación de tokens

Se suman las codificaciones de token (d_te) y posición (d_pe). Esto forma la representación inicial de entrada al decodificador. [enlace al código fuente]

v. Procesamiento por capas

Se iteran todas las capas (n_layer), y para cada una se realizan:
[enlace al código fuente]

• LayerNorm: Aplica ggml_norm, seguido de una transformación lineal con pesos y sesgos de normalización.

• Self-Attention: Multiplicaciones matriciales (Qcur, Kcur, Vcur) para obtener las matrices de consulta, clave y valor. Los tensores se almacenan en la memoria kv_self para su reutilización.

- Para la atención:

Si flash_attn está habilitado, se usa ggml flash attn ext, optimizando la atención multi-cabeza.

Pero si flash_attn no está habilitado, se realiza la multiplicación K * Q, seguida de una softmax para obtener pesos de atención, y finalmente se combinan
con V para obtener la salida.

• Proyección y Residual: La salida de atención se proyecta mediante una multiplicación matricial y se suma con la entrada original (conexión residual).

vi. Cross-Attention
[enlace al código fuente]

• Similar al self-attention, pero con esta diferencia: K y V provienen del codificador de audio (kv_cross)

• Token-level Timestamps (opcional): Si dtw_token_timestamps está habilitado, se aplican operaciones adicionales para mapear la atención a niveles de tiempo por token.

Finalmente se retorna el grafo

—----------

e. ***Relación entre los 4 grafos de Whisper***

Cada grafo tiene un rol específico en el proceso de inferencia, y su relación se basa en la transferencia de representaciones entre etapas, permitiendo que el modelo capture y procese la información.

El proceso comienza con el grafo de convolución, que toma la señal de audio en forma de espectrograma Mel y la transforma en una representación inicial mediante convoluciones 1D y activaciones GELU. Este grafo prepara la entrada para las etapas posteriores, generando una representación intermedia del audio que se almacena en un tensor llamado embd_conv. Esta salida sirve como entrada para el grafo del encoder.

El grafo del encoder toma la representación generada por la convolución y la procesa a través de múltiples capas de atención multi-cabezal y redes feed-forward. Aquí, se añaden codificaciones posicionales para preservar la información temporal del audio, y se aplican operaciones de autoatención para capturar relaciones complejas en los datos. La salida del encoder es una representación enriquecida del audio, que se almacena en un tensor y se utiliza como entrada para el grafo de cross-attention.

El grafo de cross-attention actúa como un puente entre el encoder y el decoder. Aquí, la representación del audio generada por el encoder se combina con la representación del texto en el decoder. Este grafo calcula las matrices Key y Value a partir de la salida del encoder, permitiendo que el decoder enfoque su atención en las partes relevantes del audio mientras genera el texto. La salida de este grafo es importante para que el decoder pueda alinear la información del audio con la generación de texto.

Finalmente, el grafo del decoder toma la representación del texto y la combina con la información del audio proporcionada por el grafo de cross-attention. A través de múltiples capas de autoatención y atención cruzada, el decoder genera una secuencia de tokens que representa la transcripción o traducción del audio. Este grafo también incluye operaciones de normalización, proyección y conexiones residuales para asegurar que la generación de texto sea precisa y coherente.

—----------

D. *DeepSeek*

a. ***Versión 1*** [enlace al código fuente]

La función build_deepseek() construye el grafo de cómputo para el modelo DeepSeek utilizando la estructura ggml_cgraph.

i. Inicialización y Preprocesamiento

Se inicializa el grafo (ggml_cgraph) y se definen variables como el número de tokens y la dimensionalidad de los embeddings.

Los tokens de entrada se convierten en embeddings mediante llm_build_inp_embd().

Se gestionan las posiciones de los tokens y se crea una máscara de atención.

ii. Capa por Capa

Normalización: Se aplica normalización RMS a la entrada de cada capa.

Autoatención: Se calculan las matrices Q (Consulta), K (Clave) y V (Valor), y se aplica RoPE para modelar el orden de las secuencias.

Mixture of Experts (MoE): En las capas finales, se utiliza MoE para activar dinámicamente subredes especializadas, combinadas con un experto compartido que siempre está activo.

Conexiones Residuales: Se suman las entradas originales a las salidas de cada capa para mejorar el flujo de información.

iii. Capa Final y Salida

Se aplica una normalización final y se proyecta la salida mediante una capa lineal (lm_head).

El grafo se expande con ggml_build_forward_expand() para incluir todas las operaciones necesarias.

b. *Versión 2* [enlace al código fuente]

La función build_deepseek2() construye el grafo de cómputo para el modelo DeepSeek-V2, que introduce mejoras significativas sobre su predecesor, especialmente en la eficiencia y escalabilidad.

Se ajustan dinámicamente los factores de escalado para RoPE y atención, lo que mejora la eficiencia en contextos largos.

Se reduce el tamaño del KV cache mediante técnicas de compresión, optimizando el uso de memoria y ancho de banda.

Se introduce un mecanismo de enrutamiento más eficiente para los expertos, evitando el colapso del enrutamiento y mejorando la especialización.

i. Inicialización y Preprocesamiento

Se inicializa el grafo (ggml_cgraph) y se definen variables necesarias, como el número de tokens y parámetros de escalado para RoPE y atención.

Los tokens de entrada se convierten en embeddings mediante llm_build_inp_embd().

Se gestionan las posiciones de los tokens y se crea una máscara de atención.

ii. Capa por Capa

Normalización: Se aplica normalización RMS a la entrada de cada capa.

Autoatención Optimizada:
- Se calculan las matrices Q (Consulta), K (Clave) y V (Valor) utilizando técnicas de compresión de bajo rango para reducir el tamaño del KV cache.
- Se aplica RoPE a las porciones de Q y K que requieren codificación posicional.
- Se combinan las porciones con y sin RoPE para formar las matrices finales de Q y K.

Mixture of Experts (MoE): En las capas finales, se utiliza MoE para activar dinámicamente subredes especializadas, combinadas con un experto compartido que siempre está activo.

Conexiones Residuales: Se suman las entradas originales a las salidas de cada capa para mejorar el flujo de información.

iii. Capa Final y Salida

Se aplica una normalización final y se proyecta la salida mediante una capa lineal (lm_head).

El grafo se expande con ggml_build_forward_expand() para incluir todas las operaciones necesarias.

III. SEMEJANZAS Y DIFERENCIAS

Los modelos LLAMA 2, ViT, Whisper y DeepSeek comparten una estructura basada en transformadores, pero cada uno está diseñado para tareas específicas, lo que introduce diferencias en la representación de sus grafos.

Los cuatro modelos comparten una base común utilizando mecanismos de atención, normalización, conexiones residuales y codificaciones posicionales.

Las diferencias principales radican en el tipo de entrada, el preprocesamiento inicial y el uso de técnicas específicas como cross-attention (Whisper), MLA (DeepSeek) o tokens de clase (ViT). Estas adaptaciones permiten que cada modelo sea efectivo en su dominio específico.

A. Semejanzas

- Atención Multi-Cabezal:

Los modelos emplean mecanismos de atención multi-cabezal para capturar relaciones en los datos.

En todos los casos, se calculan las matrices de Consulta (Q), Clave (K) y Valor (V), y se aplica una función softmax para obtener los pesos de atención.

- Conexiones Residuales:

Los cuatro modelos utilizan conexiones residuales para combinar la salida de una capa con su entrada original.

- Normalización:

En todos los modelos se aplica normalización de capa (LayerNorm o RMSNorm) para estabilizar las activaciones y mejorar la convergencia.

- Redes Feed-Forward:

Los cuatro modelos incluyen redes feed-forward (FFN) después de las operaciones de atención.

- Codificaciones Posicionales:

Los modelos utilizan codificaciones posicionales para preservar la información de orden en las secuencias:

- LLAMA 2 y DeepSeek: Orden de tokens en el texto.
- ViT: Orden de parches en la imagen.
- Whisper: Orden temporal en el audio.

B. Diferencias

- Estructura del Grafo:

LLAMA 2: El grafo se centra en procesar secuencias de texto, con un enfoque en la autoatención y el uso de RoPE.

ViT: El grafo se centra en procesar imágenes, con un enfoque en la autoatención aplicada a los parches de la imagen.

Whisper: El grafo está dividido en cuatro etapas (convolución, encoder, cross-attention y decoder), donde la cross-attention conecta la representación del audio con la generación de texto.

DeepSeek: El grafo se centra en procesar secuencias de texto, pero introduce MLA para optimizar la atención en contextos largos y utiliza Mixture of Experts (MoE) para mejorar la eficiencia.

- Tipo de Entrada:

LLAMA 2 y DeepSeek: Procesa tokens de texto, que son secuencias de palabras o subpalabras.

ViT: Procesa imágenes, que se dividen en parches (patches) y se transforman en secuencias de embeddings.

Whisper: Procesa señales de audio, representadas como espectrogramas Mel, que se convierten en secuencias de embeddings.

- Preprocesamiento Inicial:

LLAMA 2: Convierte los tokens en embeddings mediante una capa de embeddings.

ViT: Divide la imagen en parches y aplica una convolución 2D para generar los patch embeddings.

Whisper: Aplica convoluciones 1D al espectrograma Mel para generar una representación intermedia (embd_conv).

DeepSeek: Convierte los tokens en embeddings, pero utiliza técnicas de compresión de bajo rango para optimizar el KV cache.

- Cross-Attention:

Whisper: Utiliza cross-attention para combinar la representación del audio (proveniente del encoder) con la generación de texto en el decoder.

LLAMA 2, ViT y DeepSeek: No utilizan cross-attention, ya que no necesitan combinar dos modalidades diferentes (audio y texto).

- Token de Clase:

ViT: Incluye un token de clase (clasificación) que resume la información global de la imagen. Este token se utiliza para la clasificación final.

LLAMA 2, Whisper y DeepSeek: No utilizan un token de clase, ya que su salida es una secuencia de tokens (texto o transcripción).

- Máscaras de Atención:

LLAMA 2 y Whisper: Utilizan máscaras de atención para evitar que los tokens atiendan a posiciones futuras (atención causal).

ViT: No utiliza máscaras de atención causal, ya que los parches de la imagen no tienen un orden secuencial estricto.

DeepSeek: Utiliza máscaras de atención causal, pero optimiza el proceso mediante MLA para reducir el tamaño del KV cache.

Tabla de Semejanzas y Diferencias

Aspecto/Criterio	***LLAMA 2***	***ViT***	***Whisper***	***DeepSeek***
Tipo de Entrada	Tokens de texto	Imágenes (parches)	Señales de audio (espectrogramas)	Tokens de texto
Preprocesamiento	Embeddings de tokens	Convolución 2D (patch embeddings)	Convolución 1D (embd_conv)	Embeddings con compresión de bajo rango
Atención	Autoatención	Autoatención	Autoatención + Cross-Attention	Autoatención con MLA
Conexiones Residuales	Sí	Sí	Sí	Sí
Normalización	RMSNorm	LayerNorm	LayerNorm	RMSNorm
Codificaciones Posicionales	Rotary (RoPE)	Posicional	Posicional	RoPE mejorado
Token de Clase	No	Sí	No	No
Máscaras de Atención	Causal	No causal	Causal	Causal optimizada con MLA

IV. OPORTUNIDADES PARA EL REUSO DE CÓDIGO

Debido a que los modelos comparten una base común en su arquitectura, existen varias oportunidades para reutilizar código y crear un estándar para la construcción de grafos

- Módulo de Atención Multi-Cabezal

Reutilización: El mecanismo de atención multi-cabezal es común en los cuatro modelos. Se puede crear una función genérica que calcule las matrices Q (Consulta), K (Clave) y V (Valor), aplique la función softmax y realice la multiplicación de matrices para obtener la salida de atención.

Beneficio: Este módulo puede ser parametrizado para adaptarse a diferentes dimensiones de embeddings, número de cabezas de atención y tipos de máscaras (causal o no causal). Además, puede incluir optimizaciones como Multi-Head Latent

Attention (MLA) de DeepSeek para mejorar la eficiencia en contextos largos.

- Conexiones Residuales

Reutilización: Las conexiones residuales son utilizadas en todos los modelos para combinar la salida de una capa con su entrada original. Se puede implementar una función genérica que sume dos tensores y maneje la dimensionalidad automáticamente.

Beneficio: Este componente es independiente del tipo de entrada (texto, imágenes o audio) y puede ser reutilizado en cualquier modelo basado en transformadores.

- Normalización de Capa

Reutilización: La normalización de capa (LayerNorm o RMSNorm) es un componente común en los cuatro modelos. Se puede crear una función genérica que aplique la normalización y permita elegir entre diferentes tipos (LayerNorm, RMSNorm, etc.).

Beneficio: Este módulo puede ser reutilizado en cualquier modelo que requiera normalización, independientemente del tipo de datos.

- Red Feed-Forward (FFN)

Reutilización: Las redes feed-forward son utilizadas en todos los modelos después de las operaciones de atención. Se puede implementar una función genérica que incluya capas lineales y activaciones con la opción de elegir el tipo de activación (como RELU, GELU, SWIGLU, etc).

Beneficio: Este componente es modular y puede ser reutilizado en cualquier modelo basado en transformadores. Además, puede extenderse para soportar Mixture of Experts (MoE), como en DeepSeek.

- Codificaciones Posicionales

Reutilización: Los cuatro modelos utilizan codificaciones posicionales para preservar la información de orden en las secuencias. Se puede crear una función genérica que genere codificaciones posicionales basadas en el tipo de entrada (texto, imágenes o audio).

Beneficio: Este módulo puede ser adaptado para diferentes tipos de secuencias (tokens, parches o pasos temporales) y puede incluir variantes como Rotary Positional Embeddings (RoPE) de LLAMA 2 y DeepSeek.

- Inicialización del Grafo de Cómputo

Reutilización: La inicialización del grafo de cómputo (creación de la estructura ggml_cgraph, gestión de memoria, etc.) es un proceso común en los cuatro modelos. Se puede crear una función genérica que maneje la inicialización del grafo y la asignación de tensores.

Beneficio: Este componente es independiente del tipo de modelo y puede ser reutilizado en cualquier implementación de grafos.

V. CÓMO MANEJAR LAS DIFERENCIAS

Existen diferencias significativas en el tipo de entrada, preprocesamiento y estructura del grafo. Para manejar estas diferencias y crear un estándar para la construcción de grafos de cómputo, se pueden implementar algunas estrategias:

- Estructura del grafo modular

Estrategia: Diseñar el grafo de cómputo como una serie de módulos interconectados, donde cada módulo representa una etapa del proceso (embeddings, atención, FFN, etc.). Esto permite que los módulos específicos de cada modelo sean intercambiables.

Beneficio: Facilita la creación de un estándar para la construcción de grafos, ya que los módulos pueden ser reutilizados y adaptados según sea necesario. Además, permite la integración de innovaciones como MLA y MoE de DeepSeek sin afectar la estructura general.

- Modularización del preprocesamiento

Estrategia: Crear módulos específicos para el preprocesamiento de cada tipo de entrada:

• Texto: Módulo para convertir tokens en embeddings.

• Imágenes: Módulo para dividir imágenes en parches y aplicar convoluciones 2D.

• Audio: Módulo para procesar espectrogramas Mel con convoluciones 1D.

Beneficio: Estos módulos pueden ser intercambiables y adaptarse a diferentes tipos de datos sin afectar el resto del grafo.

- Uso de interfaces genéricas

Estrategia: Definir interfaces genéricas para los componentes que varían entre modelos, como:

• Atención: Una interfaz que permita implementar tanto autoatención como cross-attention.

• Codificaciones Posicionales: Una interfaz que permita generar codificaciones para texto, imágenes o audio.

• Mixture of Experts (MoE): Una interfaz que permita activar dinámicamente subredes especializadas, como en DeepSeek.

Beneficio: Esto permite que los componentes específicos de cada modelo sean implementados de manera flexible, sin romper la estructura general del grafo.

- Parámetros configurables

Estrategia: Utilizar parámetros configurables para adaptar el grafo a diferentes modelos:

• Tipo de Atención: Especificar si se usa autoatención, cross-attention o MLA (Multi-Head Latent Attention).

• Tipo de Normalización: Elegir entre LayerNorm, RMSNorm, etc.

• Tipo de máscara: Especificar si se usa una máscara causal o no causal.

• Tipo de MoE: Activar o desactivar Mixture of Experts.

Beneficio: Esto permite que el mismo grafo sea utilizado en diferentes modelos con solo ajustar los parámetros.

- Componentes específicos para diferencias

Estrategia: Implementar componentes específicos para manejar las diferencias principales entre los modelos:

• Token de Clase (ViT): Un módulo que agregue y procese el token de clase en el caso de ViT.

• Cross-Attention (Whisper): Un módulo que implemente cross-attention para combinar la representación del audio con la generación de texto.

• RoPE (LLAMA 2 y DeepSeek): Un módulo que aplique Rotary Positional Embeddings para modelar el orden en las secuencias de texto.

• MLA (DeepSeek): Un módulo que implemente Multi-Head Latent Attention para optimizar el KV cache en contextos largos.

Beneficio: Estos componentes pueden ser activados o desactivados según el modelo, sin afectar la estructura general del grafo.

VI. CONCLUSIÓN

Con base en la reutilización de código y el manejo de las diferencias se puede decir que es posible la creación de un estándar para la construcción de grafos de cómputo gracias a las semejanzas entre los modelos. Mediante la utilización de componentes comunes y la implementación de estrategias para manejar las diferencias se puede desarrollar un grafo modular y flexible que se adapte a diferentes tipos de modelos.

VII. REFERENCIAS

1. Meta, "Model Cards and Prompt Formats: Meta Llama 2," [En línea]. Disponible: https://www.llama.com/docs/model-cards-and-prompt-formats/other-models#meta-llama-2. [Accedido: 15-Ene-2024].
2. Meta, "Llama 2," [En línea]. Disponible: https://www.llama.com/llama2/. [Accedido: 15-Ene-2024].
3. IBM, "Llama 2," [En línea]. Disponible: https://www.ibm.com/es-es/topics/llama-2. [Accedido: 15-Ene-2024].
4. Meta, "Llama 2: Open Foundation and Fine-Tuned Chat Models," arXiv preprint arXiv:2307.09288, 2023. [En línea]. Disponible: https://arxiv.org/pdf/2307.09288. [Accedido: 15-Ene-2024].
5. Viso.ai, "Vision Transformer (ViT)," [En línea]. Disponible: https://viso.ai/deep-learning/vision-transformer-vit/. [Accedido: 18-Ene-2024].
6. A. Dosovitskiy et al., "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale," arXiv preprint arXiv:2010.11929, 2020. [En línea]. Disponible: https://arxiv.org/pdf/2010.11929. [Accedido: 22-Ene-2024].
7. Google Research, "Vision Transformer," [En línea]. Disponible: https://github.com/google-research/vision_transformer. [Accedido: 25-Ene-2024].
8. OpenAI, "Whisper: Robust Speech Recognition via Large-Scale Weak Supervision," [En línea]. Disponible: https://cdn.openai.com/papers/whisper.pdf. [Accedido: 28-Ene-2024].
9. OpenAI, "Whisper," [En línea]. Disponible: https://github.com/openai/whisper. [Accedido: 28-Ene-2024].
10. G. Gerganov, "whisper.cpp," [En línea]. Disponible: https://github.com/ggerganov/whisper.cpp. [Accedido: 28-Ene-2024].
11. OpenAI, "Whisper," [En línea]. Disponible: https://openai.com/index/whisper/. [Accedido: 1-Feb-2024].
12. OpenAI, "Whisper: Robust Speech Recognition via Large-Scale Weak Supervision," arXiv preprint arXiv:2212.04356, 2022. [En línea]. Disponible: https://arxiv.org/pdf/2212.04356. [Accedido: 5-Feb-2024].
13. NVIDIA, "Whisper ASR GGUF," [En línea]. Disponible: https://catalog.ngc.nvidia.com/orgs/nvidia/teams/nvigisdk/models/whisper_asr_gguf. [Accedido: 8-Feb-2024].
14. OpenAI, "Whisper Model Card," [En línea]. Disponible: https://github.com/openai/whisper/blob/main/model-card.md. [Accedido: 12-Feb-2024].
15. Hugging Face, "Introduction to GGML," [En línea]. Disponible: https://huggingface.co/blog/introduction-to-ggml. [Accedido: 15-Feb-2024].
16. Labellerr, "Large Language Models," [En línea]. Disponible: https://www.labellerr.com/blog/tag/large-language-models/. [Accedido: 18-Feb-2024].
17. I. Isamu, "Understanding Graph Machine Learning in the Era of Large Language Models (LLMs)," Medium, 2023. [En

línea]. Disponible: https://isamu-website.medium.com/understanding-graph-machine-learning-in-the-era-of-large-language-models-llms-dce2fd3f3af4. [Accedido: 22-Feb-2024].
18. P. Jin, "Awesome Language Model on Graphs," [En línea]. Disponible: https://github.com/PeterGriffinJin/Awesome-Language-Model-on-Graphs. [Accedido: 25-Feb-2024].
19. NVIDIA, "Efficiently Scale LLM Training Across a Large GPU Cluster with Alpa and Ray," [En línea]. Disponible: https://developer.nvidia.com/blog/efficiently-scale-llm-training-across-a-large-gpu-cluster-with-alpa-and-ray/. [Accedido: 28-Feb-2024].
20. S. Taghado, "vit.cpp," [En línea]. Disponible: https://github.com/staghado/vit.cpp. [Accedido: 28-Feb-2024].
21. G. Gerganov, "llama.cpp Discussions," [En línea]. Disponible: https://github.com/ggerganov/llama.cpp/discussions/4531. [Accedido: 28-Feb-2024].
22. G. Gerganov, "GGML Tips & Tricks," [En línea]. Disponible: https://github.com/ggerganov/llama.cpp/wiki/GGML-Tips-&-Tricks. [Accedido: 28-Feb-2024].
23. G. Gerganov, "llama.cpp Issues," [En línea]. Disponible: https://github.com/ggerganov/llama.cpp/issues/589. [Accedido: 28-Feb-2024].
24. Towards AI, "Llama Explained," [En línea]. Disponible: https://pub.towardsai.net/llama-explained-a70e71e706e9. [Accedido: 28-Feb-2024].
25. G. Gerganov, "whisper.cpp," [En línea]. Disponible: https://github.com/ggerganov/whisper.cpp/blob/master/src/whisper.cpp. [Accedido: 28-Feb-2024].
26. OpenAI, "Whisper," [En línea]. Disponible: https://github.com/openai/whisper. [Accedido: 28-Feb-2024].
27. Unsloth, "DeepSeek-R1-GGUF," [En línea]. Disponible: .https://huggingface.co/unsloth/DeepSeek-R1-GGUF. [Accedido: 3-Mar-2025].
28. TechTarget, "DeepSeek explained: Everything you need to know," [En línea]. Disponible: https://www.techtarget.com/whatis/feature/DeepSeek-explained-Everything-you-need-to-know. [Accedido: 3-Mar-2025].
29. BBC News, "DeepSeek: What you need to know," [En línea]. Disponible: https://www.bbc.com/news/articles/c5yv5976z9po. [Accedido: 3-Mar-2025].
30. GeeksforGeeks, "DeepSeek-R1: Technical overview of its architecture and innovations," [En línea]. Disponible: https://www.geeksforgeeks.org/deepseek-r1-technical-overview-of-its-architecture-and-innovations/. [Accedido: 3-Mar-2025].
31. S. Wadekar, "DeepSeek-R1 model architecture," [En línea]. Disponible: https://shaktiwadekar.medium.com/deepseek-r1-model-architecture-853fefac7050. [Accedido: 3-Mar-2025].
32. Epoch AI, "How has DeepSeek improved the Transformer architecture?" [En línea]. Disponible: https://epoch.ai/gradient-updates/how-has-deepseek-improved-the-transformer-architecture. [Accedido: 3-Mar-2025].