Implementations of Large Language Models

Zenjie Li

6th January 2025

Contents

Chapter 1

Large Language Models

In this article, we explore the architecture and implementation of Large Language Models (LLMs), and Vision-Language Models, a subset of Multi-modal Models. We focus on the design and inference of these systems, briefly explaining training techniques and datasets. This chapter is dedicated to LLMs, with subsequent chapters delving into multi-modal models that integrate visual and textual inputs.

1.1 Architectural Overview

Figure 1.1 illustrates the typical architecture of a large language model (LLM), such as LLaMA [9] and Qwen [10, 15]. The process begins by converting the input into a sequence of embedding vectors. These embeddings then pass through multiple decoder layers, each consisting of a self-attention mechanism followed by a feed-forward network. After processing through the stack of decoder blocks, a linear transformation projects the output into a logits space, corresponding to the vocabulary. A sampling method is applied to these logits to determine the next token in the sequence. This iterative process repeats with the inclusion of each new token.

1.1.1 Essence of Self-Attention Mechanism

At the heart of LLMs lies the self-attention mechanism, which allows the model to dynamically weigh the relevance of each token in the input sequence relative to all others when making predictions. This enables the model to focus on pertinent parts of the input as it generates output.

In the self-attention layer, an input list of vectors $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\}$—where each vector $\mathbf{x}_i$ corresponds to a token from the input—is transformed into an output list of vectors $\mathbf{Y} = \{\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_n\}$. The transformation is guided by the attention weights calculated between every pair of input vectors.

For causal LLMs, the self-attention mechanism adheres to the sequential order of tokens, ensuring that a given vector can only attend to itself and preceding vectors in the sequence. For example, the first token's vector $\mathbf{x}_1$ can only consider itself, whereas the second token's vector $\mathbf{x}_2$ can take into account both $\mathbf{x}_1$ and itself.

Projection of Input Vectors

Each input vector $\mathbf{a}_i$ is projected into three separate spaces to obtain the corresponding query $\mathbf{q}_i$, key $\mathbf{k}_i$, and value $\mathbf{v}_i$ vectors:

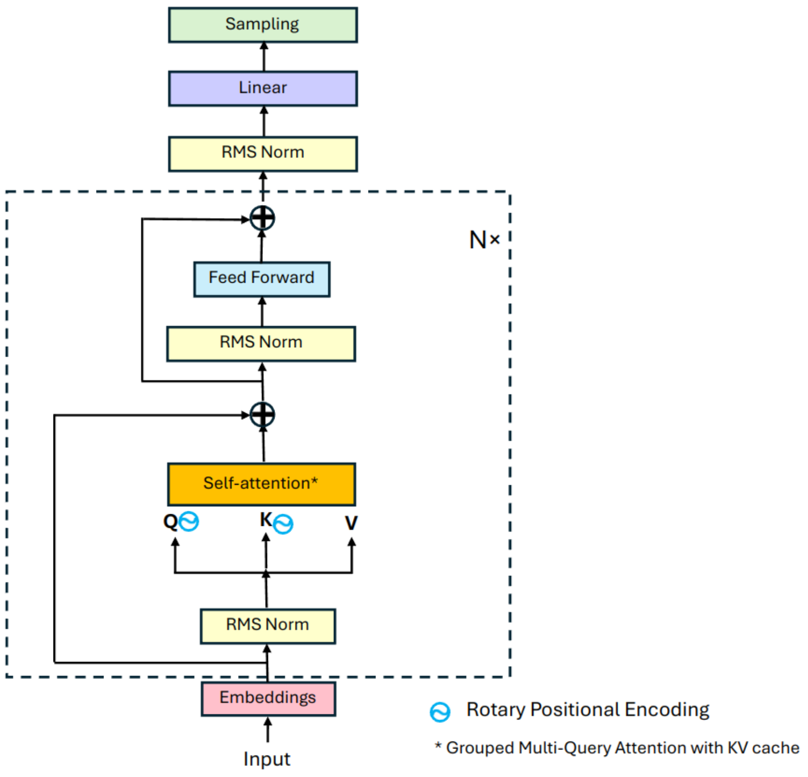

Fig. 1.1: Typical LLM architecture.

$$\begin{aligned} \mathbf{q}_i &= \mathbf{a}_i \mathbf{W}_Q \\ \mathbf{k}_i &= \mathbf{a}_i \mathbf{W}_K \\ \mathbf{v}_i &= \mathbf{a}_i \mathbf{W}_V \end{aligned} \tag{1.1}$$

where $\mathbf{W}_Q$, $\mathbf{W}_K$, and $\mathbf{W}_V$ are trainable projection matrices, and all vectors are expressed as row vectors by default in this article.

Positional Encoding with Rotary Embeddings

Rotary positional embeddings are applied to the query and key vectors after their linear projections, encoding relative positions essential for sequence-dependent tasks. Value vectors do not receive rotary embeddings. See Appendix A for implementation details.

Computation of Attention Weights

The attention weight from vector $\mathbf{v}_j$ to vector $\mathbf{v}_i$ is computed as follows:

$$\text{Attention}(\mathbf{q}_j, \mathbf{k}_i) = \frac{\exp(\mathbf{q}_j \cdot \mathbf{k}_i^T / \sqrt{d_h})}{\sum_{t=1}^{n} \exp(\mathbf{q}_j \cdot \mathbf{k}_t^T / \sqrt{d_h})} \tag{1.2}$$

where n in the input sequence length, $\mathbf{q}_j$ and $\mathbf{k}_i$ are the query and key vectors, respectively. d_h is key head size. For example, if there are 8 attention heads, d_h will be 1/8 of the

	Original Transformer	**LLaMA**
Self-attention Mechanism	Multi-head	Grouped multi-query
Input Embedding Size	Fixed	Varies with model size
Normalization	Layer normalization	RMS normalization
Positional Encoding	Fixed positional encoding	Rotary positional encoding
Activation Function	ReLU	SiLU

Table 1.1: Comparison between the original transformer and LLaMA Architectures. See Appendix B.1 for explanation of layer normalization, and Appendix B.2 for ReLU and SiLU activation functions.

embedding dimension. The dot product is scaled by the square root of d_h to stabilize gradients during training. The `softmax` function converts the raw attention scores into a probability distribution.

1.2 Model Implementation

The model implementation is detailed in the following.

1.2.1 Chat template and tokenization

A tokenizer converts splits a text into words or subwords, which then are converted to IDs through a look-up table. The most used tokenizer for causal lanuage models is Byte-Pair Encoding (BPE) which is used in LLama-3 serials, Qwen-2 serials and Phi-3 serials.

The $Huggingfacetokenizers$ library is written in Rust [1].

1.2.2 Architecture

The comparison with the original transformers is shown in Table 1.1.

The grouped multi-query, i.e. repeating some k and v heads for the q heads, achieves a balance between speed and quality. The implementation details for positional encoding, layer normalization, and activation functions are provided in the appendices.

1.2.3 Implementation Details

The following steps outline the inference process for the LLama model implemented in the `transformers` library from Hugging Face.

[1]https://github.com/huggingface/tokenizers

Step 1: Loading Pretrained Model and Tokenizer. Load the pretrained LLaMA model and tokenizer. For batch processing of input messages with varying lengths, padding is essential. Since LLaMA is a decoder-only architecture, it requires left padding, different from the default right padding in `AutoTokenizer`. Set `padding_side="left"` when instantiating the tokenizer. This step results in an instantiated and initialized tokenizer and model.

Step 2: Applying the Prompt Template. Transform the user's input text by applying a predefined template that includes system prompts, potential generation instructions, and the user's message, all separated by delimiter tokens. The formatted output may resemble:

```
<|begin_of_text|><|start_header_id|>user<|end_header_id|>\n\nHi
<|eot_id|>|start_header_id|>assistant<|end_header_id|>\n\n
```

Step 3: Tokenization. Tokenize the input text into tokens and convert each token into its corresponding ID from the vocabulary. Padding is added to the batch as necessary to ensure uniform sequence length. The result is a tensor of shape (b, n), where b is the batch size and n is the maximum sequence length in the batch.

Step 4: Embedding Generation. Generate embeddings for the input tokens using the embedding matrix of shape (n_v, d), where n_v is the vocabulary size and d is the embedding dimension. Input: input token IDs of the shape (b, n). Output: embeddings of the shape (b, n, d)

Step 5: Self-attention.

Input embeddings undergo RMS normalization. For a input tensor (vector) $\mathbf{a}$ with a shape $(d,)$, the RMS normalization is defined as $(\mathbf{a}/\sqrt{\mathbf{a}^2/n})\mathbf{W_{rms}}^T$. $\mathbf{W_{rms}}$ is the RMS normalization weight with the same shape as $\mathbf{a}$. See Appendix B.1 for details.

The normalization is followed by linear projections through W_Q, W_K and W_V matrices to produce Q, K and V (see Equation 1.1). In the case of the `llama-3.2B` model, the hidden size $d = 2048$, the head size $d_h = 64$ and the number of heads for k and v is 8.

After reshaping and transpose, the shapes are:

- Q: $(b, d/d_h, n, d_h)$
- K and V: $(b, n_{kvHeads}, n, d_h)$

In case of the `1b` model, the tensor shapes are:

- Q: $(b, 32, n, 64)$
- K and V: $(b, 8, n, 64)$

Rotary positional embeddings are then applied to the Q and K tensors to incorporate position information. These embeddings consist of a pair of tensors for cosine and sine components, each with a shape of $(1, n, d_h)$. To ensure that the key and value tensors match the dimensions of the query tensor, K and V are expanded along the second dimension (`dim=1`). In case of the `1b` model, they are repeated four times $(8 \times 4 = 32)$.

Subsequently, the attention weights are calculated using Equation1.2, and the resulting attention scores are used to compute the weighted sum of the value vectors, producing the hidden states. The initial shape of the hidden states is $(b, d/d_h, n, d_h)$, but after concatenating across all heads, it becomes (b, n, d).

A final linear transformation with the matrix W_o is applied to the concatenated hidden states, preserving the output shape as (b, n, d). Lastly, the original input embeddings are added to the transformed hidden states as part of the residual connection (see Figure 1.1), maintaining the same shape throughout.

Step 6: Feed-forward Network (FFN).

Apply another RMS normalization, followed by the FFN as described below:

$$H_{out} = [silu(\mathbf{H_{in}W_{gate}}) \odot \mathbf{W_{up}})]\mathbf{W_{down}}$$

where $\odot$ denotes element-wise multiplication, and the weight shapes are:

- W_{gate} and W_{up}: $(4d, d)$
- W_{down}: $(d, 4d)$.

It's important to note that in PyTorch, the shape of a weight matrix is defined as (in_features, out_features). The gating mechanism controls the contribution of W_{up} to the final output, allowing the model to modulate information flow.

The output of the FFN has the same shape as the input, (b, n, d), and is added to the tensor before the RMS norm as part of the residual connection (see Figure 1.1).

Step 7: Repeating Decoder Layers. Steps 5 and 6 together form a single `decoder layer`. This layer is repeated N times; for the `1B` model, it is repeated 16 times.

Step 8: Final RMS Normalization. Apply a final RMS normalization without changing the tensor's shape.

Step 9: Linear Transformation to Logits. A linear transformation maps the last dimension from d to n_v, the vocabulary size. The input shape is (b, n, d), and the output shape is (b, n, n_v), representing the logits for the next token prediction.

Step 10: Sampling the Next Token. Based on the logits, predict the next token. With greedy search, the token with the highest probability (highest logit) is selected.

Alternatively, sampling methods such as top-k or nucleus (top-p) sampling can be used to introduce some randomness in the selection process.

Step 11: Generating Additional Tokens. Return to **Step 4: Embedding Generation** and continue generating new tokens until predefined stop tokens are encountered. Decode the sequence of token IDs into a human-readable text string.

1.2.4 Parameter Analysis

Table 1.2 outlines the layers and parameters for typical LLMs. The specific configurations for the `LLaMA-3.2-1B` model and the `Qwen2.5-1.5B` model are presented in Table 1.3. The parameter values for those two models are detailed in Table 1.4.

Table 1.2: Model Layers and Parameters for typical LLMs (including LLaMA and Qwen). n_v: vocabulary size; d: embedding dimension (also called hidden size); $n_{kvheads}$: number of key/value heads; n_h: number of attention heads; n_{layers}: number of decoder blocks (self-attention + FFN); d_h: Head dimension, $d_h = d/n_h$

Layer name	**Repetition**	**Weight shape**	**# Parameters**
Embedding generation	1	(n_v, d)	$n_v \cdot d$
RMS norm	n_{layers}	$(d,)$	d
Attention layer	n_{layers}	W_Q: (d, d) W_K: $(d, n_{kvheads} \cdot d_h)$ W_V: $(d, n_{kvheads} \cdot d_h)$ W_o: (d, d)	$2d^2 + 2d \cdot n_{kvheads} \cdot d_h$ $d + 2n_{kvheads} \cdot d_h$ if Bias
RMS norm	n_{layers}	$(d,)$	d
MLP layer	n_{layers}	W_{gate}: (d_{inter}, d) W_{up}: (d_{inter}, d) W_{down}: (d, d_{inter})	$3d_{inter} \cdot$
RMS norm	1	$(d,)$	d
LM Head	1	(n_v, d)	$n_v \cdot d$

Table 1.3: Model configurations for the `LLaMA-3.2-1B` and `Qwen2.5-1.5B` Model.

Configuration	**LLaMA-3.2-1B**	**Qwen2.5-1.5B**
n_v	128,256	151,936
d	2048	1536
$n_{kvheads}$	8	2
d_{inter}	8192	8960
d_h	64	128

1.2.5 Implementation Enhancements

A few optimizations are essential for enabling real-time and interactive applications of LLMs, including Key-Value (KV) cache and the implementation of efficient attention masking.

1.2.5.1 Key-Value Cache

In the implementation of LLMs, the utilization of a Key-Value (KV) cache is a critical optimization technique. When generating a new token, instead of reprocessing the entire

Table 1.4: Model Layers and Parameters for the `LLaMA-3.2-1B` and `Qwen2.5-1.5B` Model. See 1.3 for model configurations.

Layer name	**LLaMA weight**	**LLaMA params**	**Qwen weight**	**Qwen params**
Embedding generation	$(128256, 2048)$	262,668,288	$(151936, 1536)$	233,373,696
RMS norm	$(2048,)$	2048×16	$(1536,)$	1536×28
Attention layer	W_Q: $(2048, 2048)$ W_K: $(512, 2048)$ W_V: $(512, 2048)$ W_o: $(2048, 2048)$	$10,485,760 \times 16$	W_Q: $(1536, 1536)$ $Bias_Q$: $(1536,)$ W_K: $(256, 1536)$ $Bias_K$: $(256,)$ W_V: $(256, 1536)$ $Bias_Q$: $(256,)$ W_o: $(1536, 1536)$	$5,507,072 \times 28$
RMS norm	$(2048,)$	2048×16	$(1536,)$	1536×28
MLP layer	W_{gate}: $(8192, 2048)$ W_{up}: $(8192, 2048)$ W_{down}: $(2048, 8192)$	$50,331,648 \times 16$	W_{gate}: $(8960, 1536)$ W_{up}: $(8960, 1536)$ W_{down}: $(1536, 8960)$	$41,287,680 \times 28$
RMS norm	$(2048,)$	2048	$(1536,)$	1536
LM Head	$(128265, 2048)$	262,668,288	$(151936, 1536)$	233,373,696
Total	-	1,498,482,688	-	1,777,088,000

sequence of tokens up to that point, one can leverage the KV cache to store the key (K) and value (V) matrices from previous computations.

Given the self-attention mechanism, where each token's representation is updated based on its interactions with all other tokens in the sequence, the KV cache specifically stores the K and V vectors associated with past tokens. For a newly generated token, only the K and V vectors corresponding to this token need to be computed and added to the cache. The attention scores are then calculated using the cached K and V values and the query (Q) vector of the new token, as shown in Equation 1.2.

It's important to distinguish between the K and V in the context of self-attention and the "keys" and "values" used in dictionary data structures.

For multi-turn dialogues or conversations, managing the KV cache requires careful handling. New conversation turns may introduce delimiter tokens and generation tokens so the new tokens are not simply appended to the end of the input sequence. For instance, in the chat template used by `Qwen2`, the input always ends with "`<|im_start|>assistant\n`". This means that tokens for the new conversation round are not necessarily the last tokens in the sequence. To address this, the system must implement logic to accurately identify the boundaries of new conversation segments and update the KV cache accordingly.

Unlike in single-round conversations [2], the management of the KV-cache is not handled by the `transformers` library and is addressed at the deployment platform level.

1.2.5.2 Efficient Attention Masking

Causal generation in LLMs ensures that each token can only attend to itself and the tokens that precede it, preventing future-looking attention. This is achieved through the application of an attention mask.

[2] https://huggingface.co/docs/transformers/en/kv_cache

The masking operation is executed by adding a mask matrix to the raw attention scores before applying the `softmax` function. The mask matrix has the same shape as the attention score matrix and contains $-\infty$ for positions that should be masked out and 0 elsewhere. The elements of the mask matrix MM are defined as follows:

$$M_{ij} = \begin{cases} 0 & \text{if } i \geq j \\ -\infty & \text{if } i < j \end{cases}$$

In practice, a very large negative number is used instead of $-\infty$ to maintain numerical stability. The resulting equation for computing the masked attention is given by:

$$\text{MaskedAttention}(Q, K, V, M) = \text{softmax}\left(\frac{QK^T + M}{\sqrt{d_h}}\right) V$$

Where M is the mask matrix. By adding the mask before the `softmax`, one avoids unnecessary multiplications by zero, leading to a more efficient computation.

Chapter 2

Multi-modal Large Language Models

In this chapter, we focus on Multi-modal Large Language Models (MLLMs), which integrate capabilities for processing both visual and textual data. These models are sometimes referred to as Vision-Language Models (VLMs). It is important to note that our discussion excludes certain specialized models such as CLIP and BLIP, which, while related, have distinct architectures and purposes. For the sake of simplicity, we will refer to the models under consideration as VLMs throughout this text.

Typically, a VLM architecture comprises three key components: a Large Language Model (LLM), a vision encoder and a projector (see Figure 2.1). The LLM component is responsible for understanding and generating human language, while the vision encoder processes images or video frames. The projector aligns the features extracted by the vision encoder to the LLM.

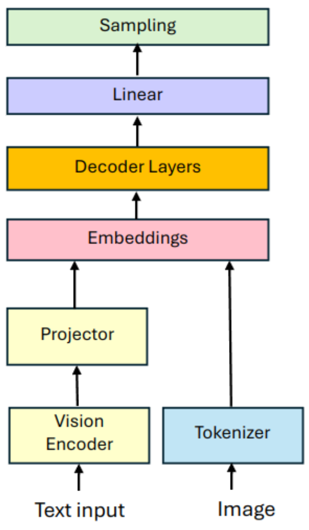

Fig. 2.1: VLM architecture. A decoder layer consists of a self-attention block and a feed-forward network (see Figure 1.1 for details).

2.1 Vision encoder

In this section, we explore the architecture of a typical vision encoder, which often utilizes the Vision Transformer (ViT) framework [6]. Figure 2.2 illustrates this architecture.

We detail the implementation process for the `Qwen2-VL` model, specifically its `2B` variant, when referring to concrete model parameters. However, the described principles are broadly applicable to other vision-language models.

Step 1: Apply Text Template.

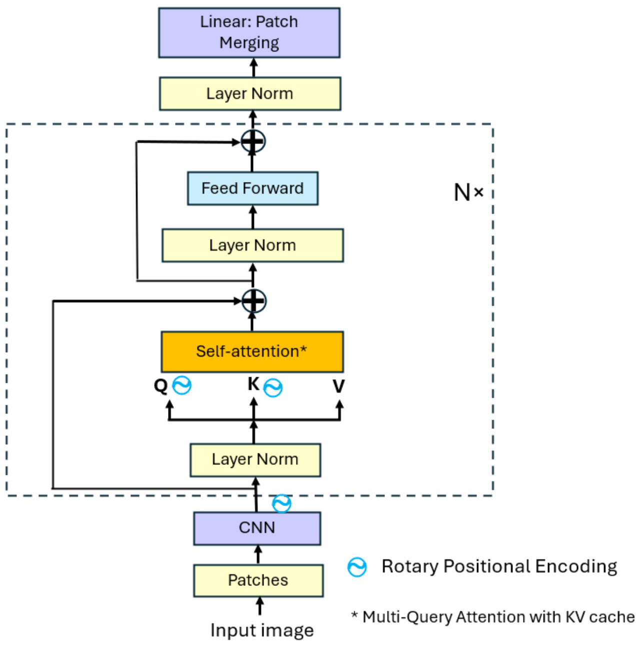

Fig. 2.2: A typical vision encoder architecture, where the encoder block (self-attention and MLP layers) is repeated N times. The patch merger serves as a projector, aligning the extracted visual features with the LLM.

Refer to Chapter 1 for detailed instructions. A dummy token is reserved as a placeholder for subsequent image insertion.

Step 2: Load and Pre-process the Image.

- Convert the image to RGB format if it isn't already.
- Resize the image such that both its height and width are divisible by the multiplication of the patch size d_{patch} and merge size n_{merge}. For the 2B model, the patch size $d_{patch} = 14$ and $n_{merge} = 2$.
- Normalize the image using predefined mean and standard deviation values.
- Transpose the image tensor to (C, H, W).

Step 3: Prepare Image Patches.

The image is divided into smaller patches. Key parameters include:

- Grid Sizes: Denoted by t_{grid} (temporal grid size), h_{grid} (grid size along height), and w_{grid} (grid size along width).
- Patch sizes: d_{t_patch} along the temporal axis, and d_{patch} along the height and width axes.

- Merge Size: n_{merge} defaults to 2, meaning two patches are merged into one along all three axes (temporal, height, and width).

The relation between the image height/width and the parameters above is

$$h = h_{grid} \times d_{patch}$$
$$w = w_{grid} \times d_{patch}$$

For an input image with shape (C, H, W):

- Temporal duplication: Duplicate and tile the image along the temporal axis to create a shape of $(2, C, H, W)$, where $t_{grid} = 1$ and $d_{t\_patch} = 2$.
- Tensor reshaping: Reshape the tensor to

$$(t_{grid}, d_{t\_patch}, C, h_{grid}/n_{merge}, n_{merge}, d_{patch}, w_{grid}/n_{merge}, n_{merge}, d_{patch}).$$

Example: An input image with dimensions $(3, 224, 224)$ (3 color channels) results in a reshaped tensor of

$$(1, 2, 3, 8, 2, 14, 8, 2, 14).$$

An image is divided into a grid of h_{grid} and w_{grid} patches, where each patch has dimensions of d_{patch}. These patches are further organized into groups of n_{merge}, facilitating their subsequent merging process.

Step 4: Transpose and reshape the image tensor.

The image tensor channels are permuted to facilitate subsequent convolution and encoding operations:

$$(t_{grid}, h_{grid}/n_{merge}, w_{grid}/n_{merge}, n_{merge}, n_{merge}, C, d_{t\_patch}, d_{patch}, d_{patch}).$$

Continuing the example, the tensor shape becomes

$$(1, 8, 8, 2, 2, 3, 14, 14, 14).$$

Step 5: Prepare image tokens.

Expand the token placeholders for the image. Initially, when we apply the chat template, we only left one placeholder for the image. Now we can compute the actual number of needed tokens. The actual number is

Expand the token placeholders for the image. Initially, only one placeholder was left when applying the text template. Calculate the actual number of required tokens:

$$\text{number of image tokens} = \frac{t_{grid} \cdot h_{grid} \cdot w_{grid}}{n_{merge}^2}$$

Tokenize both the text message and the image placeholder tokens.

Step 6: 3D convolution.

- Reshape the image tensor channels to

$$(t_{grid} \cdot h_{grid} \cdot w_{grid}, C, d_{t\_patch}, d_{patch}, d_{patch}).$$

- Apply a 3D convolution.

Convolution parameters:

- Input channels C: The color channel, always 3.
- Output channels d_{img}: The image embedding dimension, configured beforehand; for the `2B` model, this is 1280.
- Kernel: $(n_{t\_patch}, n_{patch}, n_{patch})$
- Stride: Equal to the kernel size, *i.e.*, $(n_{t\_patch}, n_{patch}, n_{patch})$

The resulting hidden states has a shape of (n_{img}, d_{img}) or, in our example, (256,1280). Here, n_{img} is the token number before merging.[1]

Step 7: Positional embedding.

To preserve the spatial structure of image patches, which is lost when they are flattened into a sequence, positional embeddings are used to provide positional information.

- Rotary Positional Embeddings (`RoPE`): Used in Models Like Qwen-VL: RoPE encodes relative positions by rotating embedding vectors using sine and cosine functions [13]. This dynamic approach allows the model to maintain relative position information even when absolute positions change, making it flexible for varying input sizes.
- Trainable Positional Embeddings: Used in Models Like InternVL: These embeddings are learned during training and added to the input patch embeddings. They can capture complex spatial patterns but require knowing the number of image tokens in advance.

Step 8: Self-attention.

[1]Some models, such as InternVL [4], incorporate a class token independent of the input image, apparently inspired by LLM classification tasks.

- Layer normalization without changing the shape.
- Project the input tensor from (n_{img}, d_{img}) to query (q), key (k), and value (v) tensors of shape (n_{img}, n_h, d_h). Parameters:
  - n_h: Number of attention heads (16 for the `2B` model).
  - d_h: Head dimension (80 for the `2B` model).

Rotary embeddings are applied to q and k to incorporate positional information. Transpose q, k, and v to (n_h, n, d_h) (see Eq. 1.1), compute multi-head self-attention, concatenate the heads, and transform again using W_o. The output shape is (n_{img}, d_{img}).

Step 9: Multi-Layer Perceptron (MLP)

- Apply another layer normalization.
- Pass through an MLP layer with weights:
  - W_1: Shape $(mlp\_ratio \cdot d_{img}, d_{img})$
  - W_2: Shape $(d_{img}, mlp\_ratio \cdot d_{img})$

In out example, W_1 is (5120,1280) and W_2 is (1280,5120). Both linear layers include bias terms. A *QuickGELUActivation* function is used between the linear operations. Add a residual connection by summing the result with the hidden states before this `MLP` layer. The output shape remains (n_{img}, d_{img}) or, in our example, $(256, 1280)$.

Step 10: Repeat Encoder Blocks.

Repeat the sequence of self-attention and `MLP` layers (Steps 8–9) for all blocks. The `2B` model contains 32 such blocks.

Step 11: Patch Merging.

- Layer normalization without altering the shape.
- Reshape the tensor to $(n_{img}/n_{merge}^2, n_{merge}^2 \cdot d_{img})$

The features for every patches are merged and concatenated, and they pass through two linear transformations:

For each set of $n_{merge} \times n_{merge}$ patches, the features from the four patches are merged and concatenated, then passed through two linear transformations:

- First transformation: Shape $(n_{merge}^2 \cdot d_{img}, n_{merge}^2 \cdot d_{img})$
- Second transformation: Shape $(d, n_{merge}^2 \cdot d_{img})$

Where d is the text token embedding size. Both transformations include bias terms. Use `GELU` activation between them. The final output shape is $(n_{img}/n^2_{merge}, d)$ or, in our example, $(64, 1536)$.

2.2 Parameter Analysis

The model configuration for a `Qwen2-VL-2B model` and `InternVL2-2B` are detailed in Table 2.1. Their parameters are detailed in Table 2.2.

Table 2.1: Configurations for the vision encoder and project components in the `Qwen2-VL-2B` and `InternVL2-2B` Model.

Configuration	**Description**	**Qwen2-VL-2B**	**InternVL2-2B**
d_{img}	Image embedding dimension (vision hidden size)	1280	1024
Class token	Whether to use for image encoder	No	Yes
Position embeddings	Whether trainable for image encoder	No	Yes
d	Token embedding dimension (LLM hidden size)	1536	2048
d_patch	Patch size along height and width	14	14
d_{t_patch}	Temporal patch size	2	1
n_{merge}	Number of patches along each dimension to be merged after encoder blocks	2	2
Encoder repetitions	Number of image encoder blocks	32	24
Scaling residual	Apply a factor when adding residual connection	No	Yes

Table 2.2: Model Layers and Parameters for vision encoder and project components in the `Qwen2-VL-2B` and `InternVL2-2B` Model. See 2.1 for model configurations.

Layer name	**Qwen2-VL weight/bias**	**Qwen2-VL params**	**InternVL2 weight**	**InternVL2 params**
Convolution	$(1280, 3, 2, 14, 14)$	1,505,280	Weight: $(1024, 3, 14, 14)$ Bias: $(1024,)$	603,136
Class embedding	None	0	(1, 1024)	1024
Positional embedding	None	0	(1025, 1024)	1,049,600
Layer norm	weight: $(1280,)$ bias $(1280,)$	2560×32	Weight: $(1024,)$ Bias $(1024,)$	2048×24
Attention layer	W_{qkv}: $(3840, 1280)$ $Bias_{qkv}$: $(3840,)$ W_o: $(1280, 1280)$ $Bias_O$: $(1280,)$	$6,558,720 \times 32$	W_{qkv}: $(3072, 1024)$ $Bias_{qkv}$: $(3072,)$ W_o: $(1024, 1024)$ $Bias_O$: $(1024,)$ Residual factor: $(1024,)$	$4,198,400 \times 24$
Residual scale	None	0	$(1024,)$	1024×24
Layer norm	Weight: $(1280,)$ Bias $(1280,)$	2560×32	Weight: $(1024,)$ Bias $(1024,)$	2048×24
MLP layer	W_1: $(5120, 1280)$ $Bias_1$: $(5120,)$ W_2: $(1280, 5120)$ $Bias_2$: $(1280,)$	13,113,600×32	W_1: $(4096, 1024)$ $Bias_1$: $(4096,)$ W_2: $(1024, 4096)$ $Bias_2$: $(1024,)$ Residual factor: $(1024,)$	$8,393,728 \times 24$
Residual scale	None	0	$(1024,)$	1024×24
Layer norm	weight: $(1280,)$ bias $(1280,)$	2560	weight: $(4096,)$ bias: $(4096,)$	8192
Merger MLP	$W_{merger1}$: $(5120, 5120)$ $Bias_{merger1}$: $(5120,)$ $W_{merger2}$: $(1536, 5120)$ $Bias_{merger2}$: $(1536,)$	34,085,376	$W_{merger1}$: $(2048, 4096)$ $Bias_{merger1}$: $(2048,)$ $W_{merger2}$: $(2048, 2048)$ $Bias_{merger2}$: $(2048,)$	12,587,008
Total		665,271,296		316,607,488
Total incl. LLM		2,442,359,296		2,205,754,368

Chapter 3

Training of Large Language Models

This chapter delves into the training process of large language models (LLMs), using Microsoft Phi-3 [1] as a case study. The methodologies discussed herein, however, can be broadly applied to the training of various LLMs.

3.1 Pre-training: Establishing a Robust Foundation

It is crucial to maximize the utility of these models' limited capacity by ensuring that they are trained on high-quality, relevant data. Two primary types of datasets are commonly employed: web-based data that has been filtered using LLM-based techniques, and synthetic datasets generated by LLMs themselves.

The pre-training phase is typically segmented into multiple stages, each with its own objectives. For Phi-3, the initial stage aims to impart the model with general knowledge and enhance its understanding of language. This is followed by a subsequent stage that focuses on refining the model's reasoning abilities.

3.2 Post-training: Specialization and Ethical Alignment

Post-training involves two critical stages: Supervised Fine-Tuning (SFT) and Direct Preference Optimization (DPO). SFT leverages high-quality, labeled datasets to boost the model's performance in specific domains or tasks. Meanwhile, DPO is designed to align the model's responses with user preferences (inluding appropriate chat format) and to ensure adherence to Responsible AI principles.

3.2.1 Fine-tuning with Low-Rank Adaptation

Low-Rank Adaptation (LoRA) is an effective technique for fine-tuning LLMs that addresses some of the challenges associated with traditional fine-tuning methods. In conventional fine-tuning, all parameters of the pre-trained model are updated, which can lead to overfitting and require substantial computational resources. By contrast, LoRA modifies only a small subset of the parameters, specifically those related to the low-rank matrices A and B, where $W = W_0 + BA^T$ represents the adapted weight matrix, with W_0 being the original weights.

Let us consider a layer in the LLM with input $x \in \mathbb{R}^d$ and output $y \in \mathbb{R}^{d'}$. The original transformation can be represented as:

$$y = W_0 x,$$

where $W_0 \in \mathbb{R}^{d' \times d}$ is the original weight matrix. With LoRA, this transformation becomes:

$$y = (W_0 + BA^T)x,$$

where $A \in \mathbb{R}^{r \times d}$ and $B \in \mathbb{R}^{d' \times r}$ are the low-rank matrices, and $r << \min(d, d')$ is the rank of the adaptation. The matrices A and B are learned during the fine-tuning process, while the original weights W_0 remain frozen.

The advantages are as follows:

- It reduces the risk of overfitting.
- It decreases computational cost and memory usage, enabling the fine-tuning of very large models even on consumer-grade hardware.

Additionally, because the original weights W_0 are not altered, the fine-tuned model can retain much of the knowledge it has acquired during pre-training. This property is particularly beneficial when working with domain-specific data that is limited in size.

3.3 Fine-Tuning with DeepSpeed

Training large-scale models, such as LLMs, can be computationally intensive and require significant memory resources. The `DeepSpeed` library offers an efficient and scalable solution to fine-tune LLMs on consumer-grade hardware.

Zero Redundancy Optimizer

The DeepSpeed library provides the Zero Redundancy Optimizer (ZeRO), which can be configured in different stages to optimize memory usage and parallelism. Instead of replicating the optimizer states, gradients, and parameters on every GPU, ZeRO partitions them so that each GPU holds only a portion of the full model state.

In ZeRO Stage 2, the optimizer states (such as momentum and variance in Adam) are partitioned across GPUs. Each GPU is responsible for updating only a subset of the model's parameters, and the corresponding optimizer states are stored only on the GPUs that own those parameters. During the backward pass, gradients are aggregated across all GPUs, and then each GPU updates its owned parameters using its local optimizer states.

In ZeRO Stage 3, in addition to the optimizer states, the gradients and parameters are divided among the GPUs. During training, communication between GPUs is required to gather the necessary information for the forward and backward passes, as well as for parameter updates.

CPU Offloading

In CPU offloading, by offloading certain components of the model to the CPU, the overall memory footprint can be further reduced.

In ZeRO Stage 2, CPU offloading can be used to move the optimizer states from the GPU to the CPU after each update. During the next iteration, the optimizer states are transferred back to the GPU as needed.

ZeRO Stage 3 takes CPU offloading by offloading also the gradients and parameters to the CPU. However, this comes at the cost of increased data transfer times between the CPU and GPU.

Best practices

To ensure efficient and effective fine-tuning with `DeepSpeed`, consider the following best practices:

- **Start with a Small Batch Size:** Begin with a smaller batch size and gradually increase it as you monitor memory usage and training stability.
- **Use Mixed Precision:** Enable mixed precision training (`fp16`) to reduce memory consumption and speed up training.
- **Leverage Gradient Accumulation:** If your GPU memory is limited, use gradient accumulation to simulate a larger batch size without increasing memory usage.
- **Optimize Communication:** Use ZeRO stages 2 or 3 to offload optimizer states and partition model parameters.

Bibliography

[1] Abdin, M., Aneja, J., Awadalla, H., Awadallah, A., Awan, A.A., Bach, N., Bahree, A., Bakhtiari, A., Bao, J., Behl, H., Benhaim, A., Bilenko, M., Bjorck, J., Bubeck, S., Cai, M., Cai, Q., Chaudhary, V., Chen, D., Chen, D., Chen, W., Chen, Y.C., Chen, Y.L., Cheng, H., Chopra, P., Dai, X., Dixon, M., Eldan, R., Fragoso, V., Gao, J., Gao, M., Gao, M., Garg, A., Giorno, A.D., Goswami, A., Gunasekar, S., Haider, E., Hao, J., Hewett, R.J., Hu, W., Huynh, J., Iter, D., Jacobs, S.A., Javaheripi, M., Jin, X., Karampatziakis, N., Kauffmann, P., Khademi, M., Kim, D., Kim, Y.J., Kurilenko, L., Lee, J.R., Lee, Y.T., Li, Y., Li, Y., Liang, C., Liden, L., Lin, X., Lin, Z., Liu, C., Liu, L., Liu, M., Liu, W., Liu, X., Luo, C., Madan, P., Mahmoudzadeh, A., Majercak, D., Mazzola, M., Mendes, C.C.T., Mitra, A., Modi, H., Nguyen, A., Norick, B., Patra, B., Perez-Becker, D., Portet, T., Pryzant, R., Qin, H., Radmilac, M., Ren, L., de Rosa, G., Rosset, C., Roy, S., Ruwase, O., Saarikivi, O., Saied, A., Salim, A., Santacroce, M., Shah, S., Shang, N., Sharma, H., Shen, Y., Shukla, S., Song, X., Tanaka, M., Tupini, A., Vaddamanu, P., Wang, C., Wang, G., Wang, L., Wang, S., Wang, X., Wang, Y., Ward, R., Wen, W., Witte, P., Wu, H., Wu, X., Wyatt, M., Xiao, B., Xu, C., Xu, J., Xu, W., Xue, J., Yadav, S., Yang, F., Yang, J., Yang, Y., Yang, Z., Yu, D., Yuan, L., Zhang, C., Zhang, C., Zhang, J., Zhang, L.L., Zhang, Y., Zhang, Y., Zhang, Y., Zhou, X.: Phi-3 technical report: A highly capable language model locally on your phone (2024), https://arxiv.org/abs/2404.14219

[2] Ba, J.L., Kiros, J.R., Hinton, G.E.: Layer normalization. arXiv preprint arXiv:1607.06450 (2016)

[3] Bai, J., Bai, S., Yang, S., Wang, S., Tan, S., Wang, P., Lin, J., Zhou, C., Zhou, J.: Qwen-vl: A versatile vision-language model for understanding, localization, text reading, and beyond. arXiv preprint arXiv:2308.12966 (2023)

[4] Chen, Z., Wu, J., Wang, W., Su, W., Chen, G., Xing, S., Zhong, M., Zhang, Q., Zhu, X., Lu, L., et al.: Internvl: Scaling up vision foundation models and aligning for generic visual-linguistic tasks. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. pp. 24185–24198 (2024)

[5] Devlin, J., Chang, M.W., Lee, K., Toutanova, K.: Bert: Pre-training of deep bidirectional transformers for language understanding (2019), https://arxiv.org/abs/1810.04805

[6] Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., et al.: An image is worth 16x16 words: Transformers for image recognition at scale. arXiv preprint arXiv:2010.11929 (2020)

[7] Glorot, X., Bordes, A., Bengio, Y.: Deep sparse rectifier neural networks. In: Proceedings of the fourteenth international conference on artificial intelligence and statistics. pp. 315–323 (2011)

[8] Hendrycks, D., Gimpel, K.: Gaussian error linear units (gelus). arXiv preprint arXiv:1606.08415 (2016)

[9] Llama Team, AI @ Meta1: The llama 3 herd of models (July 2024), https://ai.meta.com/research/publications/the-llama-3-herd-of-models/

[10] Qwen Team: Qwen2.5: A party of foundation models (September 2024), https://qwenlm.github.io/blog/qwen2.5/

[11] Radford, A., Kim, J.W., Hallacy, C., Ramesh, A., Goh, G., Agarwal, S., Sastry, G., Askell, A., Mishkin, P., Chen, J., et al.: Learning transferable visual models from natural language supervision. International Conference on Machine Learning pp. 8748–8763 (2021)

[12] Ramachandran, P., Zoph, B., Le, Q.V.: Searching for activation functions. arXiv preprint arXiv:1710.05941 (2017)

[13] Sun, Y., Zhang, M., Zhou, D., Li, J., Qin, T., Liu, T.Y.: Roformer: Enhanced transformer with rotary position embedding. arXiv preprint arXiv:2104.09864 (2021)

[14] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L., Polosukhin, I.: Attention is all you need (2023), https://arxiv.org/abs/1706.03762

[15] Yang, A., Yang, B., Hui, B., Zheng, B., Yu, B., Zhou, C., Li, C., Li, C., Liu, D., Huang, F., Dong, G., Wei, H., Lin, H., Tang, J., Wang, J., Yang, J., Tu, J., Zhang, J., Ma, J., Xu, J., Zhou, J., Bai, J., He, J., Lin, J., Dang, K., Lu, K., Chen, K., Yang, K., Li, M., Xue, M., Ni, N., Zhang, P., Wang, P., Peng, R., Men, R., Gao, R., Lin, R., Wang, S., Bai, S., Tan, S., Zhu, T., Li, T., Liu, T., Ge, W., Deng, X., Zhou, X., Ren, X., Zhang, X., Wei, X., Ren, X., Fan, Y., Yao, Y., Zhang, Y., Wan, Y., Chu, Y., Liu, Y., Cui, Z., Zhang, Z., Fan, Z.: Qwen2 technical report. arXiv preprint arXiv:2407.10671 (2024)

[16] Zhang, B., Sennrich, R.: Root mean square layer normalization (2019), https://arxiv.org/abs/1910.07467

Appendix A

Positional Embeddings

In transformer-based language models, token embeddings do not inherently carry information about a token's position within a sentence. To address this limitation, positional embeddings are introduced to encode the sequence order.

A.1 Absolute Positional Embedding

The original Transformer architecture [14] employs absolute positional embeddings, where each position in the sequence is associated with a unique embedding. For the i-th position, the embedding components are defined as:

$$\begin{aligned} P_{i,2t} &= \sin\left(\frac{i}{10000^{2t/d}}\right) \\ P_{i,2t+1} &= \cos\left(\frac{i}{10000^{2t/d}}\right) \end{aligned}$$

Here, t denotes the index within the embedding vector, and d represents the embedding dimension or hidden size. This method encodes the absolute position but does not account for the relative distances between tokens, which can be a significant drawback for certain tasks.

A.2 Learnable Positional Embedding

An alternative approach involves learnable positional embeddings, where the position-specific vectors are optimized during training. This strategy provides greater flexibility and can adapt to the specific characteristics of the training data. It has been adopted by models like BERT [5] and InternVL [4]. However, it requires a predefined maximum sequence length and predefined embedding dimension, limiting its applicability in scenarios with varying input lengths.

A.3 Rotary Positional Embedding

A more recent advancement is the Rotary Positional Embedding (RoPE) mechanism [13], which has gained popularity in modern large language models (LLMs) such as LLaMA-3 [9] and vision-language models (VLMs) like Qwen-VL [3]. RoPE effectively captures the relative positions of tokens by applying rotations to the query and key vectors in the attention mechanism.

Consider two tokens at positions m and n. When computing the attention score between the m-th query vector and the n-th key vector, RoPE ensures that the resulting dot product reflects the difference in their positions. This is achieved through the use of rotation matrices, which have the property that the product of two rotations corresponds to a single rotation by the sum of the angles.

Simplified Example with 2D Embeddings

For illustration, let's assume an embedding dimension of 2. Given a rotation matrix $R(\alpha)$, we have:

$$\begin{aligned} R(\alpha)^T &= R(-\alpha) \\ R(\alpha)R(\beta) &= R(\alpha+\beta) \end{aligned}$$

If we apply a rotation to two 2D embedding vectors $\mathbf{q}_m$ and $\mathbf{k}_n$,

$$\begin{aligned} \mathbf{q}'_m &= \mathbf{q}_m R(m\theta) \\ \mathbf{k}'_n &= \mathbf{k}_n R(n\theta) \end{aligned}$$

where θ is a fixed angle independent of the token positions, the attention computation becomes:

$$\begin{aligned} \mathbf{q}'_m \mathbf{k}'^T_n &= \mathbf{q}_m R(m\theta) R(n\theta)^T \mathbf{k}_n^T \\ &= \mathbf{q}_m R(m\theta) R(-n\theta) \mathbf{k}_n^T \\ &= \mathbf{q}_m R(m\theta - n\theta) \mathbf{k}_n^T \end{aligned}$$

This results in a rotation by the angle $(m-n)\theta$, thereby encoding the relative position of the tokens in the computed attention scores.

Geometrically, the rotation above can be understood as rotating around an angle $m\theta$ and rotate in the opposite direction around $n\theta$, so in the end we rotate $m\theta - n\theta$. Thus we obtain the relative positional information in the computed attention.

Computation of rotation operation

For a 2D embedding vector $\mathbf{q}$ with elements q_0 and q_1, the rotated vector $\mathbf{q}'$ is given by:

$$q'_0 = q_0\, cos(m\theta_0) - q_1\, sin(m\theta_0)$$
$$q'_1 = q_1\, cos(m\theta_0) + q_1\, sin(m\theta_0)$$

Extension to Higher Dimensions

In practice, the embedding dimension d is much larger than 2 and typically even. We can extend the rotation to higher dimensions by treating the embedding as a concatenation of $d/2$ 2D vectors. Each pair of elements (q_{2i-1}, q_{2i}) undergoes a rotation by an angle $\theta_i = 10000^{-2i/d}$, where $i \in [1, 2, ..., d/2]$.

The transformed embedding vector $\mathbf{q}'$ can be expressed as:

$$\begin{pmatrix} q'_1 \\ q'_2 \\ q'_3 \\ q'_4 \\ \vdots \\ q'_{d-1} \\ q'_d \end{pmatrix} = \begin{pmatrix} q_1 \\ q_2 \\ q_3 \\ q_4 \\ \vdots \\ q_{d-1} \\ q_d \end{pmatrix} \odot \begin{pmatrix} \cos m\theta_1 \\ \cos m\theta_1 \\ \cos m\theta_2 \\ \cos m\theta_2 \\ \vdots \\ \cos m\theta_{d/2} \\ \cos m\theta_{d/2} \end{pmatrix} + \begin{pmatrix} -q_2 \\ q_1 \\ -q_4 \\ q_3 \\ \vdots \\ -q_d \\ q_{d-1} \end{pmatrix} \odot \begin{pmatrix} \sin m\theta_1 \\ \sin m\theta_1 \\ \sin m\theta_2 \\ \sin m\theta_2 \\ \vdots \\ \sin m\theta_{d/2} \\ \sin m\theta_{d/2} \end{pmatrix} \tag{A.1}$$

Implementation Considerations

In practical implementations, such as in PyTorch, the RoPE algorithm optimizes element pairing for rotations to avoid altering the order of elements. Instead of pairing consecutive elements like (q_i, q_{i+1}), the implementation pairs each element q_i with its counterpart $q_{i+d/2}$, i.e., q_1 with $q_{d/2+1}$, q_2 with $q_{d/2+2}$, and so on.

This optimization leverages the fact that the elements within a hidden state do not carry an intrinsic order; they are permutation-invariant. Therefore, reordering the pairs does not affect the functionality of the RoPE mechanism while maintaining computational efficiency.

Appendix B

Normalization and Activation

Two key components that significantly influence the training dynamics and ultimate capabilities of deep-learning models are *layer normalization* and *activation functions*. This appendix delves into these concepts, focusing on their implementations within the context of large language models (LLMs).

B.1 Layer Normalization

Layer Normalization was introduced by Ba et al. in [2] as a technique to mitigate the internal covariate shift within neural networks, thereby accelerating training. Unlike Batch Normalization, which operates over batches of data and is commonly applied in convolutional neural networks (CNNs) for vision tasks, Layer Normalization acts on the last dimension of the input tensor, making it suitable for models with varying input sizes, such as those used in natural language processing.

Given a tensor $\mathbf{x}$, Layer Normalization computes the normalized output $\mathbf{y}$ as follows:

$$\mathrm{LayerNorm}(\mathbf{x}) = \gamma \cdot \frac{\mathbf{x} - \mu}{\sigma + \epsilon} + \beta$$

where μ and σ are the mean and standard deviation of $\mathbf{x}$ computed over the last dimension, γ and β are learnable affine transformation parameters that allow per-channel scaling and shifting, and ϵ is a small constant added to the denominator for numerical stability. This normalization process can be seen as a form of standardization from statistics.

Layer Normalization is a crucial component in the architecture of the original Transformer model [14].

RMS Layer Normalization

While Layer Normalization offers significant benefits, its computational cost can be high due to the calculation of the mean and variance. Zhang et al. [16] observed that the scaling operation is more critical than the shifting for many applications. They proposed RMS Layer Normalization (RMSNorm), a simplified variant that omits the mean subtraction and only scales the input tensor by its root-mean-square (RMS):

$$\mathrm{RMSNorm}(\mathbf{x}) = \frac{\mathbf{x}}{\sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2 + \epsilon}} \tag{B.1}$$

where n is the number of elements in the tensor $\mathbf{x}$ along the specified dimension, and ϵ is again a small constant for numerical stability. RMSNorm reduces the model's complexity and speeds up computation without significantly compromising performance.

RMSNorm has gained popularity in large language models (LLMs) such as LLaMA-3 [9] and Qwen2 [15], where efficiency and scalability are paramount.

B.2 Activation Functions

Activation functions introduce non-linearity into the model. In the context of LLMs and Deep Learning models, several activation functions have gained prominence due to their efficiency and performance. Among these, ReLU, SiLU, and GELU are widely adopted. Moreover, some vision models, such as Qwen-VL [3], utilize QuickGELU, a variant of GELU, for the vision part of the architecture.

Different activation functions are plotted in Figure B.1.

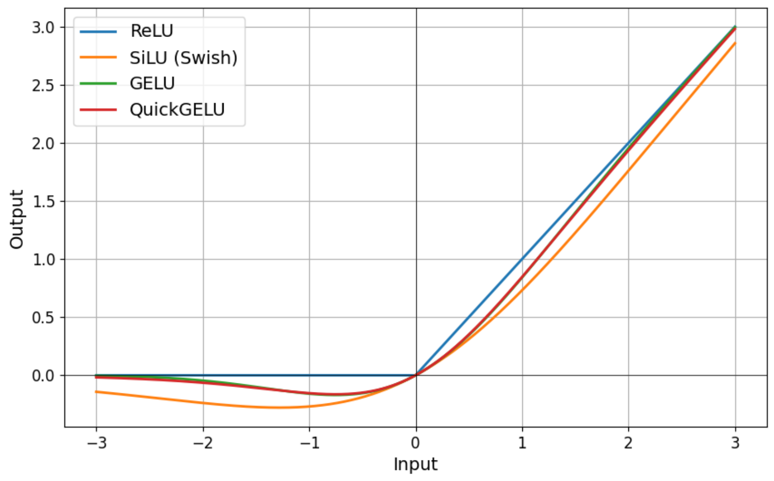

Fig. B.1: Comparison of Activation Functions - ReLU, SiLU, GELU, and QuickGELU.

ReLU (Rectified Linear Unit)

The ReLU function is one of the most commonly used activation functions in neural networks. It is defined as:

$$f(x) = \max(0, x)$$

This simple function outputs the input directly if it is positive, otherwise, it outputs zero. Due to the constant gradient of 1 for all positive inputs, ReLU helps to mitigate the vanishing gradient problem, which is common in networks with many layers [7]. However, it can lead to the "dying ReLU" problem, where neurons become inactive and always output zero.

SiLU (Sigmoid Linear Unit)

Also known as the Swish function, SiLU is a smooth, non-monotonic function. It is defined as:

$$f(x) = x \cdot \sigma(x) = \frac{x}{1 + e^{-x}}$$

where $\sigma(x)$ is the sigmoid function. Unlike ReLU, SiLU is differentiable everywhere and can output negative values for negative inputs, rather than always outputting zero. By incorporating self-gating, i.e., multiplying the input by its sigmoid, it allows for more flexible feature learning compared to ReLU [12].

GELU (Gaussian Error Linear Unit)

GELU is another non-linear activation function that is closely related to the cumulative distribution function of a Gaussian [8]. It is defined as:

$$f(x) = x \cdot \Phi(x)$$

where $\Phi(x)$ is the standard Gaussian cumulative distribution function:

$$\Phi(x) = \frac{1}{2}\left[1 + \text{erf}\left(\frac{x}{\sqrt{2}}\right)\right]$$

and erf is the error function[1]. The GELU activation function is computationally intensive, but it can be approximated to improve computational efficiency.

QuickGELU

In some vision models, such as Qwen-VL [3], a computationally efficient approximation of GELU called QuickGELU is used:

$$f(x) = x \cdot \sigma(1.702x)$$

[1] $\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$

where $\sigma(x)$ is the sigmoid function. The factor 1.702 is chosen to approximate the shape of the GELU function, making QuickGELU faster to compute while still retaining its desirable characteristics [11].

Appendix C

Sampling Strategies in LLM Generation

LLMs predicts the next word in a sequence based on a probability distribution over the vocabulary. To modulate the randomness of this process, three main sampling strategies are commonly employed: *temperature scaling*, *top-k sampling*, and *top-p (nucleus) sampling.*

At the end of this appendix, we briefly introduce beam search, which is not commonly used in practice.

Temperature Scaling

The `temperature` parameter adjusts the sharpness or flatness of the probability distribution. A lower temperature makes the model more confident but less creative.

Let y_i denote the score (logit) for the i-th word, and let t be the temperature. The adjusted probability for the i-th word is calculated as follows:

$$p_i(t) = \frac{e^{y_i/t}}{\sum_{v \in V} e^{y_v/t}}$$

where V represents the vocabulary set. When $t = 1$, the probabilities remain unchanged. As $t < 1$, the distribution becomes sharper, increasing the difference between high and low probabilities. Specifically, as t infinitely approaches zero, the sampling becomes like greedy decoding, where only the token with the highest probability is selected. Conversely, as $t > 1$, the distribution flattens, making the choice more uniform. The temperature should always be positive, typically within the range $(0, 2]$.

Top-k Sampling

Top-k sampling limits the sampling space to only the top k most likely next words. After adjusting the probabilities with the temperature, one selects the top k words with the highest probabilities. When $k = 1$, this method is equivalent to greedy search, selecting the most probable word at each step.

Top-p (Nucleus) Sampling

Top-p sampling, also known as nucleus sampling, selects the smallest set of words whose cumulative probability exceeds a threshold p. This approach dynamically adjusts the number of considered words, potentially leading to more diverse outputs compared to fixed top-k sampling.

Given a sorted list of probabilities $p_1 \geq p_2 \geq ... \geq p_{|V|}$, one finds the smallest index j such that the cumulative probability sum meets or exceeds top_p:

$$\sum_{m=1}^{j} p_m \geq top_p$$

The set of selected words consists of the first j words.

Final Sampling

After applying a combination of the above sampling methods, the probabilities of the selected words are re-normalized to ensure they sum to 1. During the final sampling step, a token is chosen according to its adjusted probability.

Beam Search

The *beam search* algorithm is a heuristic search method used to find the most probable sequence of words by exploring multiple potential sequences at each step.

The process can be summarized as follows:

- **Initialization:** Begin with an initial set of top k highest-probability tokens, forming the beam.
- **Expansion:** For each token in the current beam, generate possible continuations by predicting the next token based on the context provided by the preceding tokens. This results in a larger pool of candidate sequences.
- **Pruning:** Evaluate the likelihood of all candidate sequences and retain only the top k sequences that have the highest cumulative probability. Less promising candidates are discarded to control computational complexity.
- **Iteration:** Repeat the expansion and pruning steps until a termination condition is met.

By evaluating multiple hypotheses simultaneously, beam search can possibly yields more fluent and contextually appropriate sentences than just select the single best word at each step. However, this approach comes with increased computational cost. There is a risk that the optimal sequence might not be among the top k choices during an early stage of the search, potentially leading to its premature exclusion from further consideration.