Transformers and Large Language Models

Bach Nguyen
Arizona State University

March 10, 2025

Abstract

This report provides an overview of the transformer architecture and its application in building large language models. It covers the fundamental principles and key mechanisms.

Contents

1 Background

This is a report on how transformers and large language models work. This report's motivation came from my personal project on fine-tuning and benchmarking model DeepSeek-R1-Distill-Llama-8B on a supervised medical dataset using Low Rank Adaptation that you can take a look at this GitHub repository.

Being curious about how inputs are fed into the feed-forward neural network, and how using only 16 adapters to 8 different layers could increase the accuracy performance, I decided to find out more about the architecture. This report only focuses on non-reasoning models, such as Qwen, Llama 3, DeepSeek V3, etc. Reasoning models can be more complicated because they involve Reinforcement Learning, ...

2 Transformers

2.1 Overview

Transformers are indeed a neural network architecture designed to handle sequential data. More specifically, they translate an input sequence into an output sequence. Therefore, a transformer model learns the context of those sequential inputs to produce appropriate outputs.

Unlike recurrent neural networks (RNNs) that process the tokens sequentially, transformers use "attention mechanism" to simultaneously process the whole context. This enables faster, more efficient training and better handling of complex concepts.

2.2 Tokenizer

Although the *tokenizer* does **not** belong to the transformers itself, it preprocesses raw texts into transformers' readable language: **token IDs**, which are numerical representations of raw text. The tokenizer converts sequences of text into chunks (tokens).

On top of that, before calling the model's inference, we always use the tokenizer before feeding the inputs into our models. This may return a tensor of token IDs before being fed into the transformers.

```
inputs = tokenizer([prompt_style.format(question, " ")], return_tensors="pt").to("cuda")
```

2.3 Encoder-Decoder Structure

Note: After the tokenizer, the *encoder* takes your input tokens from texts, images, etc. and turns them into matrix representations (or tensors), while the *decoder* takes those matrix representations to generate an output. Both encoder and decoder are stacked with the same number of layers (but different sub-layers).

Encoder: Each layer may include 2 sub-layers

- Self-Attention Mechanism
- Feed-Forward Neural Network

Decoder: Each layer may include 3 sub-layers

- Masked Self-Attention
- Encoder-Decoder Attention
- Feed-Forward Neural Network

3 Encoder Workflow

3.1 Input Embeddings

Input Embeddings are the very first step of the encoder, where token IDs are treated like indices in an embedding layer. This layer maps each token ID to a vector, which captures semantic information and is then processed by the transformer model.

3.2 Positional Encoding

Again, Transformers are different from Recurrent Neural Networks (RNNs), where all tokens are processed simultaneously (not sequentially). The model cannot determine the order of tokens without *positional encoding*, which is used to provide information about the tokens' positions.

There are two main methods of positional encoding:

- Sinusoidal Positional Encoding: Uses sine and cosine functions of frequencies to encode positions.
- Leaned Positional Encoding: The vectors of tokens are learned to determine the positions.

3.3 Encoder Layers

3.3.1 Self-Attention Mechanism

The self-attention mechanism may be the most important part of a transformer model, as it allows the model to weigh the relevance of each token in the input sequence relative to every other token. For each token ID, three vectors are computed:

- **Query (Q)**: Represents the act that a token can be used to compare with all other tokens.
- **Key (K)**: Represents the characteristics of each word (or token).
- **Value (V)**: Values associated with keys are used to compute the attention score. The stronger connections queries and keys are, the higher attention scores they have.

The attention scores, which represent how well the queries can match the keys, are computed by taking the dot product of the query with all keys, scaling the result by the square root of the key dimension d_k, and applying the softmax function to obtain a probability distribution. This can be mathematically expressed as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V$$

This operation allows each token to aggregate information from the entire sequence. In practice, transformers employ *multi-head attention* where the input vectors are split into multiple subspaces (heads). Each head computes its own self-attention, and their outputs are concatenated and linearly transformed to form the final output. This enables the model to capture diverse contextual relationships within the data.

3.3.2 Feed-Forward Neural Network

Unlike the attention mechanism, which considers the inter-token relationships, the activation function allows the model to capture non-linear relationships and enhance the features.

A typical FFN consists of nothing but weights and biases with a non-linear activation function (commonly ReLU):

$$\text{F}(x) = W_2 \cdot \text{ReLU}(W_1 \cdot x + b_1) + b_2$$

where:

- x is the input vector (token representation).
- W_1 and W_2 are weight matrices.
- b_1 and b_2 are bias vectors.

To promote stable training and effective gradient flow, **residual connections** and **layer normalization** are typically applied around both the self-attention and feed-forward sub-layers.

In summary, the self-attention mechanism and the feed-forward neural network work within each encoder layer to capture global dependencies across tokens and then to improve these representations through non-linear transformations.

Remark: Outputs of the encoder are matrices of token IDs that have already well captured the context of the sequence. They are then fed into the decoder.

4 Decoder Workflow

4.1 Embeddings and Positional Encoding

Similar to the encoder, the decoder begins by converting token IDs of the target sequence into dense vector embeddings. To retain the order information, positional encoding is added to these

embeddings. This combination enables the model to incorporate both semantic meaning and token positions.

4.2 Masked Self-Attention

The decoder uses a masked self-attention mechanism in its first sub-layer. Unlike the encoder's self-attention, this layer prevents positions from attending to subsequent tokens. This masking is crucial during training, as it ensures that the prediction for a token depends only on known outputs (tokens before it). The masking is typically implemented by adding a large negative value (commonly $-\infty$) to the positions in the attention score matrix corresponding to future unknown tokens before applying the softmax function.

4.3 Encoder-Decoder Attention

After the masked self-attention, the decoder incorporates an encoder-decoder attention sub-layer. In this step, the decoder's current representations (queries) attend to the encoder's output (keys and values). This mechanism enables the decoder to integrate context from the input sequence into its generation process, ensuring that the output is closely aligned with the input data.

4.4 Feed-Forward Neural Network

Each decoder layer also includes a position-wise feed-forward neural network (FFN), similar to the encoder, working in the same way as it does in the encoder. Residual connections and layer normalization are also used around the masked self-attention, encoder-decoder attention sub-layers, and the feed-forward neural network to stabilize training.

The output of the transformer decoder is typically a sequence of high-dimensional vectors, which encapsulate both the information from the previously generated tokens (masked self-attention) and the context from the encoder (encoder-decoder attention).

5 Large Language Models (LLMs)

5.1 How Transformers Models are built

Below is the piece code of running inference for DeepSeek-R1 8B Models, which is based on the fundamental transformer model.

```
outputs = model.generate(
    input_ids = inputs.input_ids,  # tokenized input question
    max_new_tokens = 1000,         # limit the output to 1000 tokens max
    attention_mask = inputs.attention_mask,
    use_cache = True               # repeated questions give quick answer
)
```

```
# Decode the output into human-readable text
response = tokenizer.batch_decode(outputs)
```

Based on the code, after the encoder, the model takes in the input IDs and goes through the Masked Attention process to predict the next tokens. After all, we have to translate the vectors of outputs provided by the decoder in to human-readable texts.

5.2 Fine-Tune LLMs using Low-Rank Adaptation

LoRA (Low-Rank Adaptation) is a fine-tuning method designed to adapt large pre-trained models using low-cost resources. Instead of training and updating all the parameters in the LLMs (at least 1B to even hundreds of billion parameters), we can apply some adapters (matrices) into specific layers but not change the original parameters. The adapters are the only things that get updated when we start training, which reduces cost and training time.

On my project of fine-tuning a large language model, I applied 16 adapters to these 7 layers:

- **"q_proj", "k_proj", and "v_proj"** are the layers that I mentioned in the *Attention Mechanism* in both the encoder and decoder (they can be masked or unmasked).
- **"o_proj"** is also used in the *Attention Mechanism* to take outputs from multiple attention heads and apply linear transformations (matrix multiplication) to ensure that the combined information from all heads is combined into a coherent representation.
- **"gate_proj", "up_proj", and "down_proj"** are parts of the Feed-Forward Neural Networks to better process the representations provided by *Attention Mechanism*.

References

[1] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30, Long Beach, CA, USA, 2017.

[1]