How Large Language Model Works in the Prediction Stage

Zhe Li

In this note, we will step by step explain how large language model (using small 15M parameters Llama network [1] as example) works in the prediction stage. The Figure 1 shows the overall structure of this 15M parameters network, which is composed of a stack of 6 of what inside of the red dash rectangle.

[1] https://github.com/karpathy/llama2.c

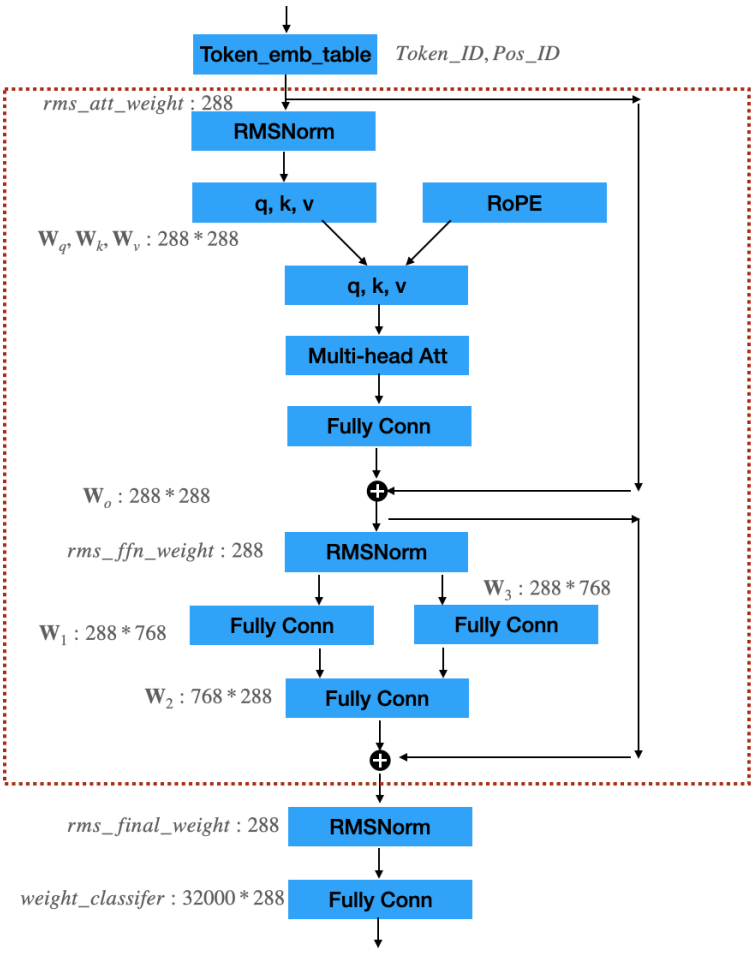

Figure 1: Model Structure

1 Token embedding

Assuming that there are totally $32,000$ distinct tokens in large language model's dictionary. Each token is represented by one embedding vector $\mathbf{e}$, which is learned during the training stage. In this note, we assume that we have already learned a well-trained embedding weight table, whose size will be $32,000 * 288$. Here $32,000$ and 288 are predefined hyperparameter. Each token is represented by a 288-dimension vector.

$$\begin{bmatrix} . & \cdots & \mathbf{e}_0 & \cdots & . \\ . & \cdots & \mathbf{e}_1 & \cdots & . \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ . & \cdots & \mathbf{e}_{31999} & \cdots & . \end{bmatrix}$$

Given the very first input token (or randomized generated the first token) with Token ID i and Position ID 0 (since we start from position 0). From the above token embedding weight table, easily get the corresponding embedding vector $\mathbf{e}_i$ by extracting the i^{th} row of the above table.

2 Root Mean Square (RMS) Normalization

Name	Symbol	Size
RMS Normalization weight	$\mathbf{w}$	288

Table 1: The Learned Parameter in the RMS Normalization Layer

Once we have the embedding vector $\mathbf{e}_i$, we can apply root mean square normalization[2]. Assuming that we have root mean square normalization weight $\mathbf{w}$, which is also 288-dimension.

$$\hat{\mathbf{e}}_i = \frac{\mathbf{e}_i}{RMS(\mathbf{e})}\mathbf{w}_i, \text{ Where } RMS(\mathbf{e}) = \sqrt{\sum_{i=0}^{287} \mathbf{e}_i^2} \tag{1}$$

After normalization, we have normalized embedding vector $\hat{\mathbf{e}}_i$.

3 Attention: Query, Key and Value

Name	Symbol	Size
query	$\mathbf{W}_q$	$288 * 288$
key	$\mathbf{W}_k$	$288 * 288$
value	$\mathbf{W}_v$	$288 * 288$

Table 2: The Learned Parameter in the Attention Layer

Assuming that we have Query, Key and Value parameter matrix $\mathbf{W}_q, \mathbf{W}_k, \mathbf{W}_v$, those sizes are 288∗288. We can compute query, key, and value vector corresponding to this normalized embedding vector (this token) by

$$\begin{aligned} \mathbf{q} &= \mathbf{W}_q * \hat{\mathbf{e}}_i \\ \mathbf{k} &= \mathbf{W}_k * \hat{\mathbf{e}}_i \\ \mathbf{v} &= \mathbf{W}_v * \hat{\mathbf{e}}_i \end{aligned} \tag{2}$$

[2] https://arxiv.org/pdf/1910.07467

where $\mathbf{q} \in \mathbf{R}^{288}, \mathbf{k} \in \mathbf{R}^{288}, \mathbf{v} \in \mathbf{R}^{288}$. So far we assume that attention is single head. If the above is multiple-head attention, it is same computation-wise. However, in the following stage, we divided query vector $\mathbf{q}$, key vector $\mathbf{k}$ and value vector $\mathbf{v}$ into different group. For example, if the number of head is 6, we will treat query vector $\mathbf{q}$ in 6 groups as

$$\begin{bmatrix} q_0 \\ \vdots \\ \vdots \\ q_{47} \end{bmatrix} \begin{bmatrix} q_{48} \\ \vdots \\ \vdots \\ q_{95} \end{bmatrix} \cdots \begin{bmatrix} q_{240} \\ \vdots \\ \vdots \\ q_{287} \end{bmatrix} \tag{3}$$

We can apply same idea to key vector $\mathbf{k}$ and value vector $\mathbf{v}$. Each group represents the different semantic, comparable to different channels in convolutional layer.

4 Position Embedding

We apply Rotary Position Embedding (RoPE)[3] to query vector $\mathbf{q}$ and key vector $\mathbf{k}$ to encode relative position information.

Name	Symbol	Size
real	fcr	$256 * (288/6/2)$
image	fci	$256 * (288/6/2)$

Table 3: The Learned Parameter in RoPE Layer

Where 256 is the defined length of the sequence, 288 is the dimension of embedding vector, and 6 is the number of head for attention. So each position in sequence, we have different real and imagine parameter. For 6 different heads, they share the same real and imagine parameters. Taking the following as example and assuming that we grab the corresponding fcr and fci vector

$$\begin{bmatrix} q_0 \\ \vdots \\ \vdots \\ q_{47} \end{bmatrix} \begin{bmatrix} fcr_0 \\ \vdots \\ fcr_{23} \end{bmatrix} \begin{bmatrix} fci_0 \\ \vdots \\ fci_{23} \end{bmatrix} \tag{4}$$

then we can compute the position-embedded query vector by

$$\begin{aligned} \hat{q}_i &= q_i * fcr_{i//2} - q_{i+1} * fci_{i//2} \\ \hat{q}_{i+1} &= q_i * fci_{i//2} + q_{i+1} * fcr_{i//2} \end{aligned} \tag{5}$$

[3] https://arxiv.org/pdf/2104.09864

the above can be written as matrix format as

$$\begin{bmatrix} \hat{q}_i \\ \hat{q}_{i+1} \end{bmatrix} = \begin{bmatrix} fcr_{i//2} & -fci_{i//2} \\ fci_{i//2} & fcr_{i//2} \end{bmatrix} \begin{bmatrix} q_i \\ q_{i+1} \end{bmatrix} \tag{6}$$

More concretely,

$$\begin{bmatrix} \hat{q}_0 \\ \hat{q}_1 \end{bmatrix} = \begin{bmatrix} fcr_0 & -fci_0 \\ fci_0 & fcr_0 \end{bmatrix} \begin{bmatrix} q_0 \\ q_1 \end{bmatrix} \tag{7}$$

write it in the big matrix-vector multiplication format, we have:

$$\begin{bmatrix} \hat{q}_0 \\ \hat{q}_1 \\ \hat{q}_2 \\ \hat{q}_3 \\ \vdots \\ \hat{q}_{46} \\ \hat{q}_{47} \end{bmatrix} = \begin{bmatrix} fcr_0 & -fci_0 & 0 & 0 & \cdots & 0 & 0 \\ fci_0 & fcr_0 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & fcr_1 & -fci_1 & \cdots & 0 & 0 \\ 0 & 0 & fci_1 & fcr_1 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & fcr_{23} & -fci_{23} \\ 0 & 0 & 0 & 0 & \cdots & fci_{23} & fcr_{23} \end{bmatrix} \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \\ \vdots \\ q_{46} \\ q_{47} \end{bmatrix} \tag{8}$$

We can apply same idea on the key vector $\mathbf{k}$ to compute the position-embedded key vector. One thing that we need to add is how to have fcr and fci parameter. After this step we have the position-embedded query vector $\hat{\mathbf{q}}$ and key vector $\hat{\mathbf{k}}$.

5 Key-Value Cache

Save the position-embedded key vector $\hat{\mathbf{k}}$ and value vector $\mathbf{v}$ at this pos to key-value cache. At the beginning, Key-Value Cache is empty, after we save the first position-embedded key vector $\hat{\mathbf{k}}$ and value vector $\mathbf{v}$, key-Value Cache is

$$\begin{bmatrix} \cdots & \hat{\mathbf{k}} & \cdots \\ \vdots & \vdots & \vdots \\ \cdots & . & \cdots \end{bmatrix} \begin{bmatrix} \cdots & \mathbf{v} & \cdots \\ \vdots & \vdots & \vdots \\ \cdots & . & \cdots \end{bmatrix} \tag{9}$$

For each head, compute the output of attention by

$$\begin{bmatrix} o_0 \\ o_1 \\ \vdots \\ o_{47} \end{bmatrix} = \left(\begin{bmatrix} \hat{q}_0 \\ \hat{q}_1 \\ \vdots \\ \hat{q}_{47} \end{bmatrix}^T \cdot \begin{bmatrix} \hat{k}_0 \\ \hat{k}_1 \\ \vdots \\ \hat{k}_{47} \end{bmatrix} \right) \begin{bmatrix} v_0 \\ v_1 \\ \vdots \\ v_{47} \end{bmatrix} \tag{10}$$

Note that

- The product $\hat{\mathbf{q}}$ and $\hat{\mathbf{k}}$ is the attention value

- Concatenate 6 head to get attention output $\mathbf{o}$ vector with size 288 dimension
- Most importantly, finally attention vector $\mathbf{o}$ is the summation of the position-embedded query vector with all previous 256 sequence length key vector and value vector, that is the reason why Key-Value Cache is necessary to reduce re-compute key vector and value vector.

6 Layers After Attention

so far we have output vector $\mathbf{o}$ of attention layer. Following attention layer, there is fully connect layer with parameters $\mathbf{W}_o$ with size $288 * 288$

$$\begin{aligned}
&\mathbf{x} = \mathbf{W}_o * \mathbf{o}, \text{ Fully Connected Layer} \\
&\mathbf{x} = \mathbf{x} + \mathbf{e}, \text{ Residual Layer} \\
&\mathbf{x}_n = RMSNorm(\mathbf{x}), \text{ RMS Normalization Layer} \\
&\mathbf{x}_1 = \mathbf{W}_1 * \mathbf{x}_n, \text{ Fully Connected Layer with } \mathbf{W}_1 \\
&\mathbf{x}_3 = \mathbf{W}_3 * \mathbf{x}_n, \text{ Fully Connected Layer with } \mathbf{W}_3 \\
&\mathbf{x}_2 = \mathbf{x}_1 + \mathbf{x}_3 \\
&\mathbf{x}_2 = \mathbf{W}_2 * \mathbf{x}_2, \text{ Fully Connected Layer with } \mathbf{W}_2 \\
&\mathbf{x}_2 = \mathbf{x}_2 + \mathbf{x}, \text{ Residual Layer}
\end{aligned} \tag{11}$$

7 The Final Layers for Predicting Next Token

After $n_layers = 6$ repeated multi-head attention blocks, there is RMS Normalization layer and final token classifier layer. Assuming the output after 6 repeated multi-head attention block is $\mathbf{x}$, we apply the root mean square normalization as same as in Section 2.

$$\mathbf{x} = RMSNorm(\mathbf{x}), \text{ RMS Normalization Layer} \tag{12}$$

The final token classifier is to predict next token. For the vector $\mathbf{x}$, We apply fully connected layer operation. There are two options on fully connected layer weights: 1) shared the weights from token embedding layer; 2) re-learned weight for this layer. The reason why the first option works is that the weights n the classifier layer indeed are the embedding representation. The advantage of this option is that it will reduce the significant amount of parameters, which will be seen in later section.

$$\mathbf{x}_o = \mathbf{W}\mathbf{x}, \text{ Full Connected layer, } \mathbf{W} \in \mathbf{R}^{32,000*288}, \mathbf{x} \in \mathbf{R}^{288} \tag{13}$$

then we can apply softmax on this output vector $\mathbf{x}_o \in \mathbf{R}^{32,000}$ to compute the probability vector of next token and predict the next token. We omit to present how softmax works here.

8 The Hype Parameters

Although in the above sections, we have mentioned some hype parameters, here we list them all in the table 4 as reference.

Parameter Name	Symbol	Value	Note
Token size	$token\_size$	$32,000$	
Sequence length	seq_len	256	
Token Embedding Dim	dim	288	
Hidden Dim	$hidden\_dim$	768	
Num of repeat block	n_layers	6	
Num of heads in multiple-head attention	n_heads	6	

Table 4: The Hype Parameters

9 The Learned Parameters

Although in the above section, we have mentioned learned parameters, we summarize them in the table as reference.

Param Name	Size	Num of Params
Token embedding table	$Token\_size * dim$	$32,000 * 288 = 9,216,000$
rms att weight	$n\_layer * dim$	$6 * 288 = 1,728$
$\mathbf{W}_q$	$n\_layer * dim * dim$	$6 * 288 * 288 = 497,664$
$\mathbf{W}_k$	$n\_layer * dim * dim$	$6 * 288 * 288 = 497,664$
$\mathbf{W}_v$	$n\_layer * dim * dim$	$6 * 288 * 288 = 497,664$
$\mathbf{W}_o$	$n\_layer * dim * dim$	$6 * 288 * 288 = 497,664$
rms_ffn_weight	$n\_layer * dim$	$6 * 288 = 1,728$
$\mathbf{W}_1$	$n\_layer * dim * hidden\_dim$	$6 * 288 * 768 = 1,327,104$
$\mathbf{W}_2$	$n\_layer * hidden\_dim * dim$	$6 * 768 * 288 = 1,327,104$
$\mathbf{W}_3$	$n\_layer * dim * hidden\_dim$	$6 * 288 * 768 = 1,327,104$
rms_final_weight	dim	288
$freq\_cis\_real$	$seq\_len * (dim/n\_heads)/2$	$256 * (288/6)/2 = 6,144$
$freq\_cis\_imag$	$seq\_len * (dim/n\_heads)/2$	$256 * (288/6)/2 = 6,144$
$weight\_classifier$	$Token\_size * dim$	$32,000 * 288 = 9,216,000(Shared)$
Total		$15,204,000$

Table 5: The Learned Parameters

As we can see from the table 5, parameters in the token embedding layer is almost 60% of all parameters. Generally speaking, in large language model, the large the number

of tokens in the directionally, the stronger the express ability of the model is. For the estimation, token size is around $30K$ to $60K$ and embedding dimension is around 256 to 1024 in the ballpark, thus, only embedding table will be in the rough range of 7.6M to 61M parameters along. That is the reason why a language model is large.