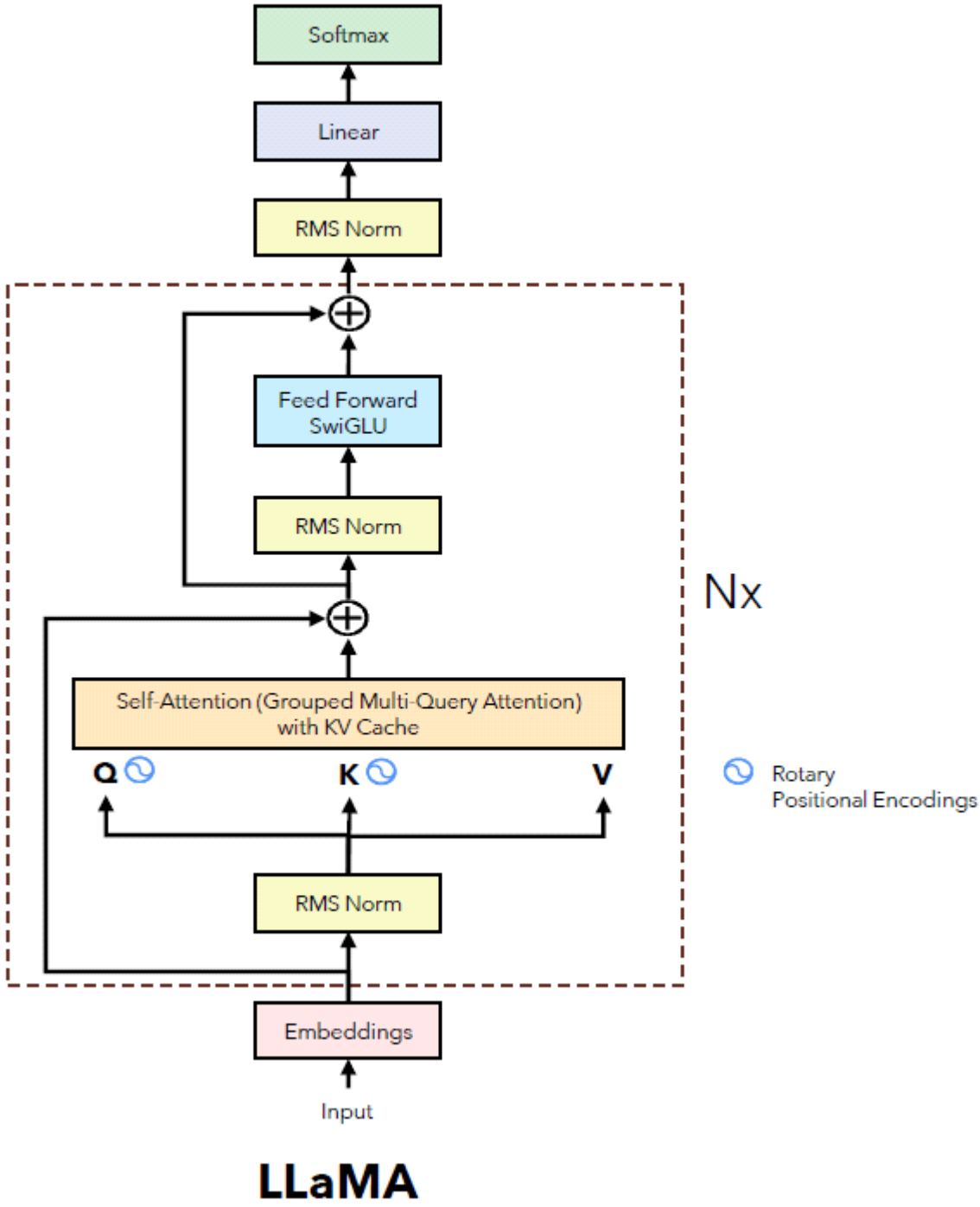
Llama 2

27 July 2025 22:12



RMS_Norm

27 July 2025 23:22

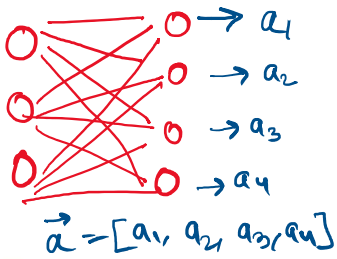
4 RMSNorm

A well-known explanation of the success of LayerNorm is its re-centering and re-scaling invariance property. The former enables the model to be insensitive to shift noises on both inputs and weights, and the latter keeps the output representations intact when both inputs and weights are randomly scaled. In this paper, we hypothesize that the re-scaling invariance is the reason for success of LayerNorm, rather than re-centering invariance.

We propose RMSNorm which only focuses on re-scaling invariance and regularizes the summed inputs simply according to the root mean square (RMS) statistic:

$$\bar{a}_i = \frac{a_i}{\text{RMS}(\mathbf{a})} g_i, \quad \text{where } \text{RMS}(\mathbf{a}) = \sqrt{\frac{1}{n}\sum_{i=1}^{n} a_i^2}. \tag{4}$$

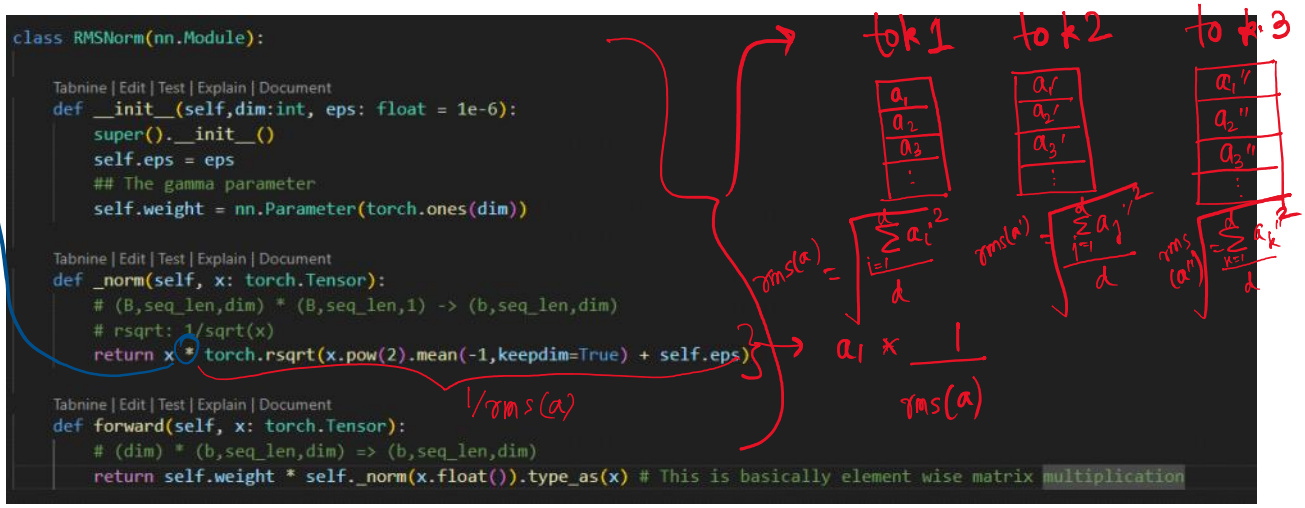
Intuitively, RMSNorm simplifies LayerNorm by totally removing the mean statistic in Eq. (3) at the cost of sacrificing the invariance that mean normalization affords. When the mean of summed inputs is zero, RMSNorm is exactly equal to LayerNorm. Although RMSNorm does not re-center



Just like Layer Normalization, we also have a learnable parameter **gamma** (**g** in the formula on the left) that is multiplied by the normalized values.

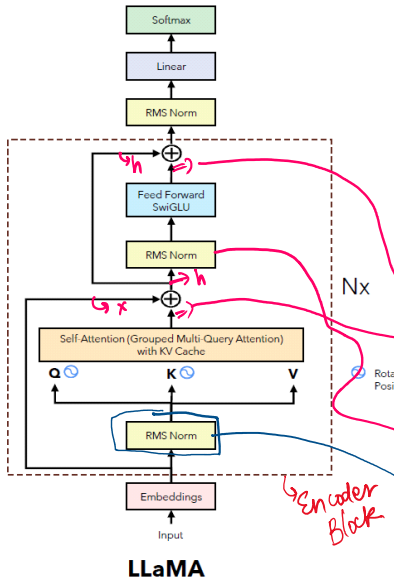
element wise multiplication.

```python
class RMSNorm(nn.Module):

    Tabnine | Edit | Test | Explain | Document
    def __init__(self,dim:int, eps: float = 1e-6):
        super().__init__()
        self.eps = eps
        ## The gamma parameter
        self.weight = nn.Parameter(torch.ones(dim))

    Tabnine | Edit | Test | Explain | Document
    def _norm(self, x: torch.Tensor):
        # (B,seq_len,dim) * (B,seq_len,1) -> (b,seq_len,dim)
        # rsqrt: 1/sqrt(x)
        return x * torch.rsqrt(x.pow(2).mean(-1,keepdim=True) + self.eps)

    Tabnine | Edit | Test | Explain | Document
    def forward(self, x: torch.Tensor):
        # (dim) * (b,seq_len,dim) => (b,seq_len,dim)
        return self.weight * self._norm(x.float()).type_as(x) # This is basically element wise matrix multiplication
```



Encoder Block

27 July 2025 23:53




```python
class EncoderBlock(nn.Module):

    Tabnine | Edit | Test | Fix | Explain | Document
    def __init__(self,args: ModelArgs):
        super().__init__()

        self.n_heads = args.n_heads
        self.dim = args.dim
        self.head_dim = args.dim   // args.n_heads

        self.attention = SelfAttention(args)
        self.feed_forward = FeedForward(args)

        # Normalization BEFORE the self.attention
        self.attention_norm = RMSNorm(args.dim,eps=args.norm_eps)

        # Normalization BEFORE the feed forward block
        self.ffn_norm = RMSNorm(args.dim, eps=args.norm_eps)

    Tabnine | Edit | Test | Explain | Document
    def forward(self, x: torch.Tensor, start_pos: int, freqs_complex: torch.Tensor):
        # (B,seq_len,dim) + (b, seq_len, dim) => (b, seq_len, dim)
        h = x + self.attention.forward(self.attention_norm(x), start_pos, freqs_complex)
        out = h + self.feed_forward.forward(self.ffn_norm(h))
        return out
```

RoPE

11 August 2025 10:41

$$\boldsymbol{R}^d_{\Theta,m}\boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ \vdots \\ x_{d-1} \\ x_d \end{pmatrix} \otimes \begin{pmatrix} \cos m\theta_1 \\ \cos m\theta_1 \\ \cos m\theta_2 \\ \cos m\theta_2 \\ \vdots \\ \cos m\theta_{d/2} \\ \cos m\theta_{d/2} \end{pmatrix} + \begin{pmatrix} -x_2 \\ x_1 \\ -x_4 \\ x_3 \\ \vdots \\ -x_{d-1} \\ x_d \end{pmatrix} \otimes \begin{pmatrix} \sin m\theta_1 \\ \sin m\theta_1 \\ \sin m\theta_2 \\ \sin m\theta_2 \\ \vdots \\ \sin m\theta_{d/2} \\ \sin m\theta_{d/2} \end{pmatrix}$$

How do we calculate this in code

1) $\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}$ ① reshape $\Rightarrow$ $\begin{bmatrix} [x_1 & x_2] \\ [x_3 & x_4] \end{bmatrix}$ ② $\Rightarrow$ convert to polar $\begin{bmatrix} x_1 + ix_2 \\ x_3 + ix_4 \end{bmatrix}$ ↓↓ x_complex

2) $m = [1, 2, 3, \cdots, seq\_len]$

$\theta = [\theta_1, \theta_2, \theta_3, \cdots, \theta_{d/2}]$

m (outer_product) $\theta \Rightarrow$

$$\begin{bmatrix} m_1\theta_1 & m_1\theta_2 & m_1\theta_3 & \cdots & m_1\theta_{d/2} \\ m_2\theta_1 & m_2\theta_2 & m_2\theta_3 & \cdots & m_2\theta_{d/2} \\ \vdots & & & & \end{bmatrix}$$

freqs_complex ↑

↓↓ Polar

$[[\cos(m_1\theta_1) + i\sin(m_1\theta_1)] \quad [\cos(m_1\theta_2) + i\sin(m_1\theta_2)] \cdots$

② element wise

(Doing for m_1)

(Doing for m_1)

3) x - complex $\otimes$ freqs_complex

$$\Rightarrow \begin{bmatrix} x_1 + i x_2 \\ x_3 + i x_4 \end{bmatrix} \otimes \begin{bmatrix} \overbrace{\cos(m_1\theta_1)}^{f_1} + i \overbrace{\sin(m_1\theta_1)}^{f_2} \\ \underbrace{\cos(m_1\theta_2)}_{f_3} + i \underbrace{\sin(m_1\theta_2)}_{f_4} \end{bmatrix}$$

$$(x_1 + i x_2)(f_1 + i f_2) = x_1 f_1 + i x_1 f_2 + i x_2 f_1 - x_2 f_2$$
$$= (x_1 f_1 - x_2 f_2) + i(x_1 f_2 + x_2 f_1)$$

$$\begin{bmatrix} x_1 f_1 - x_2 f_2 + i(x_1 f_2 + x_2 f_1) \\ x_3 f_3 - x_4 f_4 + i(x_3 f_4 + x_4 f_3) \end{bmatrix}$$

$\Downarrow$ (3) Convert to Real

$$\begin{bmatrix} [(x_1 f_1 - x_2 f_2) & x_1 f_2 + x_2 f_1] \\ [x_3 f_3 - x_4 f_4 & x_3 f_4 + x_4 f_3] \end{bmatrix}$$

$\Downarrow$ (4) Flatten

$$\begin{bmatrix} x_1 f_1 - x_2 f_2 \\ x_1 f_2 + x_2 f_1 \\ x_3 f_3 - x_4 f_4 \\ \vdots \end{bmatrix}$$

$\Rightarrow$ This is the matrix that we want.

$$\begin{bmatrix} x_3 f_3 - x_4 f_4 \\ x_3 f_4 + x_4 f_3 \end{bmatrix}$$

we want

Self Attention

August 11, 2025 11:45 PM

Rough ⇒ n_kv_heads (n_kv), q_heads (n_q), n_rep = n_q // n_kv

head_dim = dim // n_q

wq ⇒ (dim, n_q × head_dim)
wk ⇒ (dim, n_kv × head_dim)
wv ⇒ (dim, n_kv × head_dim)
wo ⇒ (n_q × head_dim, dim)

cache_k ⇒ (max_bs, max_seq_len, n_kv, head_dim)
cache_v ⇒ (max_bs, max_seq_len, n_kv, head_dim)

x → (b, seq_len, dim)

xq = wq(x) ⇒ (b, seq_len, dim) * (dim, dim)
⇒ (b, seq_len, dim)

xk = wk(x) ⇒ (b, seq_len, dim) * (dim, n_kv * head_dim)
⇒ (b, seq_len, n_kv * head_dim)

xv = wv(x) ⇒ (b, seq_len, dim) * (dim, n_kv * head_dim)
⇒ (b, seq_len, n_kv * head_dim)

xq ⇒ (b, seq_len, dim) → (b, seq_len, n_q, head_dim)

xk ⇒ (b, seq_len, n_kv * head_dim) → (b, seq_len, n_kv, head_dim)

xv ⇒ (b, seq_len, n_kv * head_dim) → (b, seq_len, n_kv, head_dim)

xq ⇒ RoPE(xq) (b, seq_len, n_q, head_dim)

x_k ⇒ RoPE(x_k) (b, seq_len, n_kv, head_dim)

cache_k[:b, start_pos : start_pos + seq_len] = xk

cache_v[:b, s_pos : s_pos + seq_len] = xv

keys = cache_k[:b, : s_pos + seq_len]

cache_v[:b, : s_pos + seq_len]

keys = cache_k[:b, : start_pos + seq_len]
values = cache_v[:b, : start_pos + seq_len]

Repetition of keys and values upto n_rep
bcz for every q, k & v will be same.

∴ keys = repeat_kv(keys, self.n_rep)
⇒ (b, seq_len, n_kv, head_dim) → (b, seq_len, n_q, head_dim)

values = ...
⇒ (b, seq_len, n_kv, head_dim) → (b, seq_len, n_q, head_dim)

xq = xq.transpose(1,2)
⇒ (b, seq_len, n_q, head_dim) → (b, n_q, seq_len, head_dim)

keys = transpose(1,2)
⇒ (b, seq_len, n_q, head_dim) → (b, n_q, seq_len, head_dim)

values = ...
⇒ (b, seq_len, n_q, head_dim) → (b, n_q, seq_len, head_dim)

xq * keys.transpose(2,3)
⇒ (b, n_q, seq_len, head_dim) × (b, n_q, head_dim, seq_len)

⇓

(b, n_q, seq_len, seq_len)

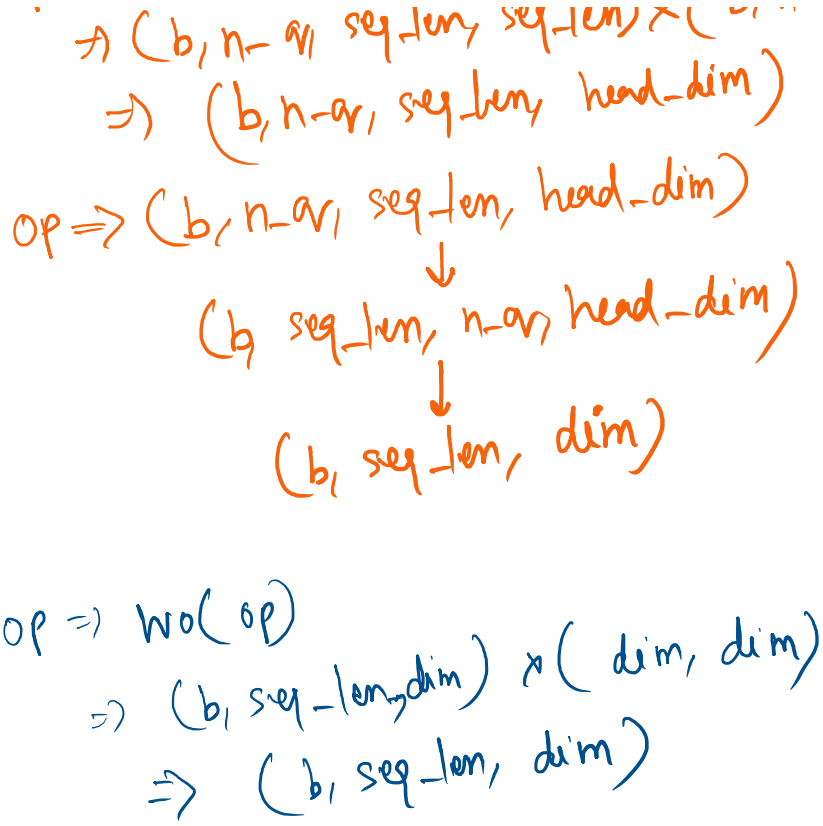
⇓

softmax

⇓

(b, n_q, seq_len, seq_len)

o/p ⇒ scores × values
⇒ (b, n_q, seq_len, seq_len) × (b, n_q, seq_len, head_dim)
⇒ (b, n_q, seq_len, head_dim)


(b, n_q, seq_len, seq_len) x (
(b, n_q, seq_len, head_dim)
op => (b, n_q, seq_len, head_dim)
(b, seq_len, n_q, head_dim)
(b, seq_len, dim)

op => wo(op)
(b, seq_len, dim) x (dim, dim)
(b, seq_len, dim)