Appendices

A Proof for Lemma 1.

Given a linear transformation $\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x} + \boldsymbol{b}, \boldsymbol{x} \in \mathbb{R}^n, \boldsymbol{A} \in \mathbb{R}^{m\times n}, \boldsymbol{b}, \boldsymbol{y} \in \mathbb{R}^m$, we have

$$\begin{aligned}
\boldsymbol{y} &= \boldsymbol{A}\boldsymbol{x} + \boldsymbol{b} && (7)\\
&= (\boldsymbol{A} - k\mathbf{1}\mathbf{1}^T\boldsymbol{A})\boldsymbol{x} + k\mathbf{1}\mathbf{1}^T\boldsymbol{A}\boldsymbol{x} + (\boldsymbol{b} - \mu(\boldsymbol{b})\mathbf{1}) + \mu(\boldsymbol{b})\mathbf{1} && (8)\\
&= (\boldsymbol{A} - k\mathbf{1}\mathbf{1}^T\boldsymbol{A})\boldsymbol{x} + (\boldsymbol{b} - \mu(\boldsymbol{b})\mathbf{1}) + k(\mathbf{1}^T\boldsymbol{A}\boldsymbol{x})\mathbf{1} + \mu(\boldsymbol{b})\mathbf{1} && (9)\\
&= (\boldsymbol{A} - k\mathbf{1}\mathbf{1}^T\boldsymbol{A})\boldsymbol{x} + (\boldsymbol{b} - \mu(\boldsymbol{b})\mathbf{1}) + (k\mathbf{1}^T\boldsymbol{A}\boldsymbol{x} + \mu(\boldsymbol{b}))\mathbf{1} && (10)\\
&= \hat{\boldsymbol{A}}\boldsymbol{x} + \hat{\boldsymbol{b}} + f(\boldsymbol{x}, k)\mathbf{1} && (11)
\end{aligned}$$

where $\hat{\boldsymbol{A}} = \boldsymbol{A} - k\mathbf{1}\mathbf{1}^T\boldsymbol{A}, \hat{\boldsymbol{b}} = \boldsymbol{b} - \mu(\boldsymbol{b})\mathbf{1}, f(\boldsymbol{x}, k) = k\mathbf{1}^T\boldsymbol{A}\boldsymbol{x} + \mu(\boldsymbol{b})$.

If $k = 1/m$, then we obtain

$$\begin{aligned}
\mu(\hat{\boldsymbol{A}}\boldsymbol{x}) &= \frac{1}{m}\mathbf{1}^T(\boldsymbol{A} - \frac{1}{m}\mathbf{1}\mathbf{1}^T\boldsymbol{A})\boldsymbol{x} = \frac{1}{m}(\mathbf{1}^T\boldsymbol{A} - \mathbf{1}^T\boldsymbol{A})\boldsymbol{x} = 0 && (12)\\
\mu(\boldsymbol{y}) &= \mu(\hat{\boldsymbol{A}}\boldsymbol{x}) + \mu(\hat{\boldsymbol{b}}) + \mu(f(\boldsymbol{x}, k = 1/m)\mathbf{1}) && (13)\\
&= 0 + 0 + f(\boldsymbol{x}, k = 1/m) && (14)\\
&= \frac{1}{m}\mathbf{1}^T\boldsymbol{A}\boldsymbol{x} + \mu(\boldsymbol{b}) && (15)
\end{aligned}$$

The $\hat{\boldsymbol{A}}$ is the recentered matrix of $\boldsymbol{A}$, and all its column vectors have zero-mean. We decompose the output into two parts.

- The first part $\hat{\boldsymbol{A}}\boldsymbol{x} + \hat{\boldsymbol{b}} = \boldsymbol{y} - \mu(\boldsymbol{y})\mathbf{1}$, with zero mean, is another linear transformation with $\hat{\boldsymbol{A}} = \boldsymbol{A} - \frac{1}{m}\mathbf{1}\mathbf{1}^T\boldsymbol{A}, \hat{\boldsymbol{b}} = \boldsymbol{b} - \mu(\boldsymbol{b})\mathbf{1}$.
- The second part corresponds to the mean information $\mu(\boldsymbol{y})\mathbf{1} = (\frac{1}{m}\mathbf{1}^T\boldsymbol{A}\boldsymbol{x} + \mu(\boldsymbol{b}))\mathbf{1}$.

B Post-LN Transformers

Different from the Pre-LN Transformers, the Post-LN Transformers have the following blocks.

$$\boldsymbol{x}_{l+1} = \text{LN}(\boldsymbol{x}_l + \mathcal{F}_l(\boldsymbol{x}_l)),\ l = 0, 1, ..., L-1, \tag{16}$$

Layer normalization is on the main branch instead of the beginning of residual branches. We can keep a zero-mean branch on the main branch without impact on the functionality.

$$\begin{aligned}
\boldsymbol{x}_{l+1} &= \text{LN}(\boldsymbol{x}_l + \mathcal{F}_l(\boldsymbol{x}_l)) && (17)\\
&= \text{LN}((\boldsymbol{x}_l - \mu(\boldsymbol{x}_l)\mathbf{1}) + (\mathcal{F}_l(\boldsymbol{x}_l) - \mu(\mathcal{F}_l(\boldsymbol{x}_l))\mathbf{1})) && (18)\\
&= \text{LN}(\hat{\boldsymbol{x}}_l + \hat{\mathcal{F}}_l(\boldsymbol{x}_l)) && (19)\\
&= \text{RMSNorm}(\hat{\boldsymbol{x}}_l + \hat{\mathcal{F}}_l(\boldsymbol{x}_l)) && (20)
\end{aligned}$$

For the residual branch $\mathcal{F}_l$, we can apply the same method in Pre-LN Transformer. We can modify the output linear projection to obtain $\hat{\mathcal{F}}_l$, which will generate the zero-mean part of the original result.

The recentering operation $\hat{\boldsymbol{x}}_l = \boldsymbol{x}_l - \mu(\boldsymbol{x}_l)\mathbf{1}$ requires extra computation. If elementwise affine transformation is disabled in LayerNorm, $\boldsymbol{x}_l$ is the output of a normalization such that $\mu(\boldsymbol{x}_l) = 0$ and $\hat{\boldsymbol{x}}_l = \boldsymbol{x}_l$. If the transformation is enabled, $\boldsymbol{x}_l$ is not guaranteed zero-mean such that the explicit recentering is necessary.

C Experiments

C.1 Implementation of normalization

We have provided our implementation with JAX and PyTorch in the supplementary material. The reported results are based on the following implementations.

For JAX, we use the APIs of LayerNorm and RMSNorm in the flax library. For PyTorch, we use the implementations of LayerNorm and RMSNorm from NVIDIA's apex extension [4]. For CRMSNorm, we use our own customized implementations. We also provide our customized implementation of LayerNorm, RMSNorm.

We notice that there are lots of APIs for the standard LayerNorm and RMSNorm. For example, PyTorch has provided the official LayerNorm API but lacks of RMSNorm implementation. These different implementations are mixed. We do not find one implementation that is dominant over others for all the cases. For instance, `torch.nn.LayerNorm` is usually faster than apex's one when the input vectors are small in inference, while it is slower than the apex's one when the input vectors are large. PyTorch's official implementation is also slower than apex's for training.

C.2 Extended Experiments in ViT

Name	Dimension	Depth	Heads	MLP Dimension
Tiny-16	192	12	3	192×4
Small-16	384	12	6	384×4
Base-16	768	12	12	768×4
Large-16	1024	24	16	1024×4
Huge-14	1280	32	16	1280×4
Giant-14	1664	48	16	8192

Table 2: ViTs with different sizes. The number in the model name is the patch size.

	no norm	**Pre-LN**	**Pre-RMS**	**Pre-CRMS**
PyTorch, single A100, amp	0.8567	1.000	0.9699	0.9783
amp $\rightarrow$ float32	0.9353	1.000	0.9850	0.9951
single A100 $\rightarrow$ 16-thread CPU	0.8697	1.000	0.9012	0.8857
PyTorch $\rightarrow$ JAX	0.9610	1.000	0.9873	1.0005

Table 3: Normalized inference time of ViT.

Table 2 list the architecture parameters of Vision Transformer. We first measure the **inference time**. We sweep these 6 ViTs with 6 batch sizes (1, 4, 16, 64, 256, 1024) and collect the medians of these 36 data points. We report the average of these 36 experiments in Table 3. We conduct inference on a single A100 with automatic mixed precision (amp) in PyTorch. We further change the precision (disabling the amp), computation platforms (16 threads in AMD EPYC 7742 CPUs), and machine learning frameworks (JAX).

C.3 Numerical Issue

The theoretical arithmetic equivalence cannot be fully translated into equality in the practical numerical computation if we use floating numbers. An intuitive example is that $\mu(\boldsymbol{x}+\boldsymbol{y}) = \mu(\boldsymbol{x}) + \mu(vy)$ always holds for any vectors $\boldsymbol{x}, \boldsymbol{y}$. However, if these two vectors are represented as (low precision) floating numbers, this equality is not guaranteed in real-world numerical computation. It is possible that these small discrepancies may be accumulated and enlarged in the large models, further degrading the numerical stability. In our proposed method, we cannot ensure the exactly zero-mean in the main branch numerically.

The numerical issue is a common problem in machine learning. A typical example is operator reordering and layer fusion. PyTorch provides a related API officially, named `torch.ao.quantization.fuse_modules`. We can fuse the convolution layer and its following batch normalization layer to simplify the computation. These two layers are separate in training and can be fused to accelerate the inference. The fusion does not break the arithmetic equivalence but changes the numerical results. In spite of the numerical difference, the fusion usually has a neutral impact on task-related performance, such as classification accuracy, even in large models. Fine-tuning or calibration may be helpful in case there is severe performance degradation.

[4] https://github.com/NVIDIA/apex

Our proposed methods encounter a similar issue as layer fusion since we modify partial parameters. In our experiments, we can convert the pre-trained Pre-LN ViT-H/14 into Pre-(C)RMS variants without any accuracy change on the ImageNet validation dataset. Actually, we observe that replacing PyTorch's official LayerNorm implementation with the apex's one may have a larger impact on the model performance.