1 Linear Algebra

1.1 Column wise decomposition.

Any matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$ can be decomposed into the sum of its columns:

$$\mathbf{A} = \sum_{i=1}^{n} \mathbf{A}_{:j} e_j^\top, \tag{1}$$

where e_j are standard basis vectors of $\mathbb{R}^n$. Notice that this is a rank 1 decomposition.

1.2 Row wise decomposition.

Any matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$ can be decomposed into the sum of its rows:

$$\mathbf{A} = \sum_{i=1}^{m} e_i \mathbf{A}_{i:}^\top, \tag{2}$$

where e_i are standard basis vectors of $\mathbb{R}^m$. Notice that this is a rank 1 decomposition.

2 LLM Training

2.1 Scaling the logits after LLM head.

We usually apply RMS norm to normalize (along the last dimension E, where it stands for model dimension, B means batch size and L means sequence length) the tensor $\mathbf{X} \in \mathbb{R}^{B\times L\times E}$ we feed into LLM head, and obtain the corresponding logits l. After RMS normalization, each tensor corresponding to the token $x_i \in \mathbb{R}^E$ will then have $\mathrm{RMS}(x_t) = 1$. Now notice that for each coordinate $x_{i,t}, t \in [E]$, treating as a random variable, its variance is given by

$$\mathrm{Var}\left(x_{i,t}\right) = \mathbb{E}\left[x_{i,t}^2\right] - \left(\mathbb{E}\left[x_{i,t}\right]\right)^2, \tag{3}$$

and if it is zero-mean (or small), then $\mathrm{Var}\left(x_{i,t}\right) \simeq \mathbb{E}\left[x_{i,t}^2\right]$, which is to say that second moment reflects the variance.

The next step is to use the empirical observation that for linear layers, hidden vectors tend to be approximatedly rotation-invariant (isotropic), i.e., each coordinate behaves like the others, so we can use the second moment over the coordinate in a token to replace the actual second moment. And the former, is given by

$$\mathrm{Var}\left(x_{i,t}\right) \simeq \frac{1}{E}\sum_{t=1}^{E} x_{i,t} = 1. \tag{4}$$

Now we start to consider the logits, which is generated by

$$l_{j,i} = w_j^\top x_i = \sum_{t=1}^{E} w_{j,t} x_{i,t}.$$

If we assume each weight entry $w_{j,t}$ are i.i.d. with variance σ^2 the logits variance is give by

$$\mathrm{Var}\left(l_{j,i}\right) = \sum_{t=1}^{E} \sigma^2\,\mathrm{Var}\left(x_{t,i}\right) \simeq E\sigma^2.$$

So the standard deviation $\sim \sqrt{E}$. To ensure that logits do not scale with the model dimension, we scale it by $\sqrt{E}$.

3 Attention

3.1 Multihead Self Attention (MHA)

Consider an input tensor $\mathbf{X} \in \mathbb{R}^{B \times L \times d_{\text{model}}}$ to an attention layer, where B is the batch size, L is the sequence length, and d_{model} is the model dimension.

1. The first step involves computing queries, keys, and values. We have three matrices, $\mathbf{W}_q$, $\mathbf{W}_k$, and $\mathbf{W}_v \in \mathbb{R}^{d_{\text{model}} \times d_{\text{model}}}$, and simultaneously perform the following operations

$$\mathbf{Q} = \mathbf{X}\mathbf{W}_q; \quad \mathbf{K} = \mathbf{X}\mathbf{W}_k; \quad \mathbf{V} = \mathbf{X}\mathbf{W}_v. \tag{5}$$

   These operations are vectorized, meaning that for each sequence b in the batch of size B, we do

$$\mathbf{Q}_b = \mathbf{X}_b\mathbf{W}_q \quad \forall b \in [B].$$

   $\mathbf{W}_q$, $\mathbf{W}_k$, and $\mathbf{W}_v$ are trainable parameters shared across the entire batch. The resulting $\mathbf{Q}$, $\mathbf{K}$, and $\mathbf{V}$ have the shape $\mathbb{R}^{B \times L \times d_{\text{model}}}$.

2. Next, for multihead attention, we reshape $\mathbf{Q}$, $\mathbf{K}$, and $\mathbf{V}$ from shape $\mathbb{R}^{B \times L \times d_{\text{model}}}$ into $\mathbb{R}^{B \times H \times L \times d_{\text{head}}}$, where H is the number of attention heads and d_{head} is the dimension of each head. To achieve this, we first divide d_{model} into H heads, resulting in shapes of $\mathbb{R}^{B \times L \times H \times d_{\text{head}}}$. Then we rearrange into $\mathbb{R}^{B \times H \times L \times d_{\text{head}}}$. Conceptually, each head uses a subset of dimensions from d_{model} to compute scores between queries and keys along the sequence dimension L. We will use the following notations $\mathbf{Q}_h, \mathbf{K}_h, \mathbf{V}_h$ to denote the per head tensor in $R^{B \times 1 \times L \times d_{\text{head}}}$ for each head $h \in [H]$.

3. In the next step, we perform the attention calculation:

$$\begin{aligned} \mathbf{S}_h &:= \text{Scores}_h(\mathbf{Q}_h, \mathbf{K}_h) = \frac{\mathbf{Q}_h \mathbf{K}_h^\top}{\sqrt{d_{\text{head}}}} + \mathbf{M} \\ \mathbf{A}_h &:= \text{Attention}_h(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\mathbf{S}_h\right)\mathbf{V}_h, \quad \forall h \in [H]. \end{aligned}$$

   The scaled multiplication of $\mathbf{Q}_h$ and $\mathbf{K}_h^\top$ is vectorized, resulting in $\mathbf{S}_h \in \mathbb{R}^{B \times 1 \times L \times L}$ and $\mathbf{S} \in \mathbb{R}^{B \times H \times L \times L}$.[1] Optionally, we could use a mask matrix to mask out certain tokens, an example would be the causual self attention. To stabilize the gradients, we element-wise divide raw scores by $\sqrt{d_{\text{head}}}$. This scaling choice can be justified because each element of $\mathbf{Q}_h\mathbf{K}_h^\top$ represents a dot product between vectors of dimension d_{model}. The variance of this dot product scales as $\text{Var}\left(\langle q_h, k_h \rangle\right) \sim d_{\text{model}}\sigma_q^2\sigma_k^2$. Since variance scales quadratically, we divide by $\sqrt{d_{\text{head}}}$. The softmax operation turns the scores after masking into probabilities, along the last dimension.[2] Imagine $z = [z_1, \ldots, z_L] \in \mathbb{R}^L$ is a row vector, then essentially, softmax defines the operation:

$$\sigma(z)_i := \frac{e^{z_i}}{\sum_{j=1}^L e^{z_j}}. \tag{6}$$

   Sometimes we use a numerically stable version to replace it

$$\tilde{\sigma}(z) := \frac{e^{z_i - \max(z)}}{\sum_{j=1}^L e^{z_j - \max(z)}}. \tag{7}$$

   It is worth mentioning that in single head attention (scaled dot product), the complexity of computation is $\mathcal{O}\left(BL^2 d_{\text{model}}\right)$, while for multihead attention, it is the same since we do $\mathcal{O}\left(H \times BL^2 d_{\text{head}}\right) = \mathcal{O}\left(BL^2 d_{\text{model}}\right)$.

[1]Here $\mathbf{S}$ is the stack of $\mathbf{S}_h$ along dimension H.

[2]This is to say that for each $L \times L$ matrix, we softmax every row.

4. Finally, we concatenate and mix attention outputs from all heads. Concatenation involves first transposing $\mathbf{A}_h$ to $\mathbb{R}^{B\times L\times H\times d_{\text{head}}}$ and then merging the last two dimensions into $\mathbf{A}\in\mathbb{R}^{B\times L\times d_{\text{model}}}$. This concatenated result is projected using a matrix $\mathbf{W}_O$, as follows:

$$\text{MHA}(\mathbf{X}) = \mathbf{A}\mathbf{W}_O. \tag{8}$$

The final output retains the shape $\mathbb{R}^{B\times L\times d_{\text{model}}}$.

There a bunch of reasons why we are using multi heads instead of scaled dot product attention.

- **Diversity of learned attention patterns**: Each head learns different attention patterns in parallel. A single attention head computes only one set of attention scores.
- **Subspace specialization**: Instead of operating in d_{model}, each head projects to a lower dimension subspace d_{head}. This suggests that each head operates in a distinct feature subspace.
- **Improved gradient flow and representation mixing**: Independent paths improve gradient flow and richness of learned representations.

Notice that the compuatational cost are the **SAME**!

3.2 Multi Query Attention (MQA)

In MQA, different heads have its own query, but share the same key and value. Specifically, for a head h, we have

$$\begin{aligned}\mathbf{S}_h &:= \text{Scores}(\mathbf{Q}_h,\mathbf{K}) = \frac{\mathbf{Q}_h\mathbf{K}^\top}{\sqrt{d_{\text{head}}}} + \mathbf{M}\\ \mathbf{A}_h &:= \text{Attention}_h(\mathbf{Q}_h,\mathbf{K},\mathbf{V}) = \text{softmax}(\mathbf{S}_h)\mathbf{V}, \quad \forall h\in[H].\end{aligned}$$

This means that for each head h, we have a separate $\mathbf{Q}_h\in\mathbb{R}^{B\times 1\times L\times d_{\text{head}}}$ and shared $\mathbf{K},\mathbf{V}\in\mathbb{R}^{B\times 1\times L\times d_{\text{head}}}$.

Compared to standrad MHA, MQA has the following features:

- **Reduced parameter count**: Each head has its own query only, shared key and value.
- **Smaller activation size (memory usage)**: Now $\mathbf{K},\mathbf{V}\in\mathbb{R}^{B\times 1\times L\times d_{\text{head}}}$, so the activation size is smaller.
- **Reduced KV cache (fast inferencing)**: For transformer based models such as GPT, we generate text one token at a time. To avoid recomputing attention over all previous tokens on every step, we cache k, v (key and value vectors) for all previously seen tokens. Specifically, in standard MHA, for each layer and token generated, we neeed $2BHLd_{\text{head}}$ for cached $\mathbf{K},\mathbf{V}$. In MQA, we share $\mathbf{K}$ and $\mathbf{V}$ so that the cost becomes $2BLd_{\text{head}}$.
- **Minimal accuracy loss.** Used in GPT-3.5, PaLM, LLaMA, etc.

3.3 KV cache

The motivation for KV caching is to enable efficient inference — both in terms of compute time and memory bandwidth. At inference time only, autoregressive models input a sequence of tokens $\{x_t,\ldots,x_{t+L-1}\}$ to generate the next token x_{t+L}. To avoid recomputing key and value vectors for all previous tokens every time, we cache the k, v pairs corresponding to the tokens $x_t\ldots,x_{t+L-1}$ in the forward pass. Then, when generating x_{t+L+1}, we can reuse the vectors for cached $x_{t+1},\ldots,x_{t+L-1}$ and we only need to compute k, v for x_{t+L}. This mechanism is known as the **KV cache**.

3.4 Grouped Query Attention (GQA)

GQA is like an interpolation between MQA and MHA, where we ask groups of heads to share $\mathbf{K}, \mathbf{V}$. Specifically, let $g(h)$ be a function that maps a head h to its corrsponding group index, then we have

$$\begin{aligned}\mathbf{S}_h &:= \text{Scores}(\mathbf{Q}_h, \mathbf{K}_{g(h)}) = \frac{\mathbf{Q}_h \mathbf{K}_{g(h)}^\top}{\sqrt{d_{\text{head}}}} + \mathbf{M} \\ \mathbf{A}_h &:= \text{Attention}_h(\mathbf{Q}_h, \mathbf{K}_{g(h)}, \mathbf{V}_{g(h)}) = \text{softmax}(\mathbf{S}_h)\mathbf{V}_{g(h)}, \quad \forall h \in [H].\end{aligned}$$

Benefits:

1. Less memory than MHA.
2. Flexible compute/memory tradeoff by controlling the number of k, v heads.

It is used in LLaMA 2 and Mistral.

3.5 Multihead Latent Attention (MLA)

Before we go into details, we need to first differentiate between self attention and cross attention.

- Self attention is the case when Q, K, V comes from the same input sequence. Its typically used in encoder blocks of BERT, GPT, LLaMA, etc, and decoder blocks in GPT, T5, etc.
- Cross attention refers to the case when Q comes from one sequence but K, V comes from another sequence. It is typically used in the case that decoder attends to encoder outputs (T5, BART), and the case of vision language models where text attends to image, and perceiver-style latent attention.

We can formulate cross attention in the following way: Let $\mathbf{Z} \in \mathbb{R}^{B \times M \times d_{\text{model}}}$ (expanded from $\mathbb{R}^{1 \times M \times d_{\text{model}}}$.) be a target sequence (queries) and $\mathbf{X} \in \mathbb{R}^{B \times L \times d_{\text{model}}}$ be a source sequence (keys and values),

$$\mathbf{Q} = \mathbf{Z}\mathbf{W}_q; \quad \mathbf{K} = \mathbf{X}\mathbf{W}_k; \quad \mathbf{V} = \mathbf{X}\mathbf{W}_v. \tag{9}$$

Notice that M could be different than L. In the case of cross-attention with a latent array, we often have, $M << L$, which significantly reduces computational cost by avoiding full self-attention over the entire input sequence. We then compute the attention scores and softmax:

$$\mathbf{S}_h := \text{Scores}(\mathbf{Q}_h, \mathbf{K}_h) = \frac{\mathbf{Q}_h \mathbf{K}_h^\top}{\sqrt{d_{\text{head}}}} \tag{10}$$

$$\mathbf{A}_h := \text{Attention}_h(\mathbf{Q}_h, \mathbf{K}_h, \mathbf{V}_h) = \text{softmax}(\mathbf{S}_h)\mathbf{V}_h, \quad \forall h \in [H]. \tag{11}$$

Notice that in this case $\mathbf{S}_h \in \mathbb{R}^{B \times 1 \times M \times L}$ and $\mathbf{A}_h \in \mathbb{R}^{B \times 1 \times M \times d_{\text{head}}}$ for head $h \in [H]$. After concatenation, we result in $\mathbf{A} \in \mathbb{R}^{B, M \times d_{\text{model}}}$, which is like we are focusing on a smaller sequence. In MLA, latent vector it self is a learnable parameter and shared accross a batch. These latents act like information bottleneck that extract useful features from the long input $\mathbf{X} \in \mathbb{R}^{B \times L \times d_{\text{model}}}$.

3.6 Latent Transformer Block

This is a key design of Perceiver (2021, DeepMind), Set Transformer and efficient transformers for long inputs (e.g., audio, video, documents). Essentially it can be viewed as cross attention followed by latent self-attention. Mathematically speaking, we are giving a vector $\mathbf{X} \in \mathbb{R}^{B \times L \times d_{\text{model}}}$. First we are using the latent vector $\mathbf{Z} \in \mathbb{R}^{1\times M\times d_{\text{model}}}$ (expanded accross batch dimension.) and the input $\mathbf{X} \in \mathbb{R}^{B\times L\times d_{\text{model}}}$, we have

$$\mathbf{A}_1 = \text{Attention}(\mathbf{X}_q = \mathbf{Z}, \mathbf{X}_k = \mathbf{X}, \mathbf{X}_v = \mathbf{X}),$$

which essentially asks "What should I learn from all of you tokens?". After this step: each latent now contains information extracted from the input. Notice that now $\mathbf{A}_1 \in \mathbb{R}^{B\times M\times d_{\text{model}}}$ is a compressed representation of $\mathbf{X}$, extracted by the latent array. Then we do normal self attention on the latent variable:

$$\mathbf{A}_2 = \text{Attention}(\mathbf{X}_q = \mathbf{A}_1, \mathbf{X}_k = \mathbf{A}_1, \mathbf{X}_v = \mathbf{A}_1)$$

where each latent vector is allowed to look at other latents, share what they learned and refine itself.

```
for each block:
  z = z + CrossAttention(q ← z, k ← x, v ← x)
  z = z + SelfAttention(q ← z, k ← z, v ← z)
```

Before we actually feed the $\mathbf{Z}$ and $\mathbf{X}$ into the attention block and the final feed forward layer, we first do normlization (LayerNorm in the case of my code).

3.7 Pre- and Post- Normlization

In general, there are two ways of doing layer normlization, Post-LN and Pre-LN. In the original implementation of transformer, post-LN is used. However, pre-LN has become the modern default, which is used in GPT-2/3/4, T5, LLaMA, PaLM, Perceiver, etc. The benefits of using pre-LN includes the follows:

- **(Help gradient flow & increasing training stability)**: In a deep stack, residual paths carry the untouched signal forward. With Pre-LN, those residual paths also carry unit-variance, zero-mean activations (because they are already normalized). That keeps gradients well-scaled and prevents the exploding / vanishing issues that appeared when stacking 24 - 100+ layers with Post-LN. Empirically, Pre-LN lets you train hundreds (even thousands) of layers with a stable learning rate schedule, whereas Post-LN often needed warm-up tricks or gradient clipping.
- **(Easier optimization of very long sequences)**: Cross-entropy loss is applied after the final LayerNorm. With Post-LN every sub-layer's output is renormalized, the network must constantly "undo" those shifts. Pre-LN leaves the residual branch untouched, so the model can accumulate information across time steps or tokens without repeatedly rescaling it.
- **(Faster convergence)**: Many ablations show 1.3 - 1.5x faster convergence for GPT/T5 style models when switching from Post-LN $\rightarrow$ Pre-LN. This is because every tensor that flows straight down the stack (both forward activations and backward gradients through the residual skip) has mean 0 and variance 1, which helps stabilize second-moment estimate quickly for Adam.
- **(Safer with half-precision / mixed-precision)**: Normalizing before the high-variance matrix multiplications keeps activations in a narrower numeric range, reducing overflow/underflow risk in FP16/BF16 training.

LayerNorm: Mathematically speaking, consider an input $\mathbf{X}$ in the space $\mathbb{R}^{B\times L\times d_{\text{model}}}$, for the l-th token in the b-th batch

$$\mathbf{L}[b,l,:] = \text{LayerNorm}(\mathbf{X}[b,l,:]) = \gamma \cdot \frac{X[b,l,:] - \mu_{n,l}}{\sqrt{\sigma_{b,l}^2 + \epsilon}} + \beta,$$

where

$$\mu_{b,l} = \frac{1}{d_{\text{model}}} \sum_{i=1}^{d_{\text{model}}} \mathbf{X}[b,l,i], \quad \sigma^2_{b,l} = \frac{1}{d_{\text{model}}} \sum_{i=1}^{d_{\text{model}}} \left(\mathbf{X}[b,l,i] - \mu_{b,l}\right)^2,$$

$\gamma, \beta \in \mathbb{R}^{d_{\text{model}}}$ are learned shift and scale vectors. Notice that statistics are computed per sample, per position, no batch coupling, so the network behaves the same in training and inference and is robust to batch-size 1. Its benefits includes:

- **(Zero-mean, unit-var features).** Keeps dot-products in a predictable range, resulting in stable softmax gradients.
- **(Identical behaviour in training / inferencing).** Important for autoregressive generation where batch size changes.
- **(Works with any sequence length).** No running-average statistics needed.

In a simpler form, layernorm can be written as

$$\mathbf{L} = \gamma \odot \frac{\mathbf{X} - \mu}{\sqrt{\sigma + \epsilon}} + \beta,$$

where all operations are elementwise. Notice that $\gamma, \beta \in \mathbb{R}^{d_{\text{model}}}$ are learnable parameters.

3.8 Dropout

During Training: This refers to During training we randomly set a fraction of the sub-layer output activations to zero and scale the survivors. Specifically, for $\mathbf{X} \in \mathbb{R}^{B \times L \times d_{\text{model}}}$ and the drop out rate $0 < p < 1$, we sample a binary traning mask $\mathbf{M} \in \{0,1\}^{B \times L \times d_{\text{model}}}$, using the Bernoulli($q$) distribution where $q = 1 - p$, we then apply

$$\widehat{\mathbf{X}} = \frac{1}{q}\mathbf{X} \odot \mathbf{M}.$$

q is referred to as the keep probability, and we are trying to make $\mathbb{E}\left[\widehat{\mathbf{X}}\right] = \mathbf{X}$.

During Inferencing: At inference time the mask is removed and $\mathbf{X}$ passes through unchanged.
Code:

```
torch.nn.Dropout(p=p)
```

3.9 DCA

This one seems to be less well known, on medical image tasks.

3.10 Linear Attention

3.11 Sparse Attention

4 Normalizations

Besides the layer norm mentioned in the previous chapter, there are other norms used quite often.

4.1 BatchNorm

Batch norm is rarely used in language modeling especially in modern architectures. It is not sequence or position aware, and it requires consistent statistics over a batch, with variable-length sequences in language modeling, the batch statistics can be unstable and unreliable. Furthermore, it mixes information from each sequences, but ideally we want to keep it separate. However, it is great for vision tasks

Mathematically, given an input $\mathbf{X} \in \mathbb{R}^{B\times d}$, for each feature j, we have

$$\mu_j = \frac{1}{B}\sum_{i=1}^{B}\mathbf{X}[i,j], \quad \sigma_j^2 = \frac{1}{B}\sum_{i=1}^{B}(\mathbf{X}[i,j]-\mu_j)^2.$$

Then each value is normlized in a way that

$$\hat{X}[i,j] = \gamma[j]\frac{X[i,j]-\mu_j}{\sqrt{\sigma_j^2+\epsilon}} + \beta[j],$$

where $\gamma, \beta \in \mathbb{R}^d$ are learnable parameters per feature / channel.

If we are given an input 2D/Convolutional data $\mathbf{X} \in \mathbb{R}^{B\times C\times H\times W}$, we have

$$\mu_c = \frac{1}{BHW}\sum_{b,h,w}\mathbf{X}[b,c,h,w], \quad \sigma_c^2 = \frac{1}{BHW}\sum_{b,h,w}(\mathbf{X}[b,c,h,w]-\mu_c)^2,$$

basically, we are computing mean and variance per channel. Notice that **LayerNorm is still the default in most Vision Transformer (ViT-style) backbones**. In CNN, sometimes GroupNorm replaces BatchNorm because it's batch-size independent and mixes well with vision features.

4.2 RMSNorm

Its name is Root Mean Square Layer Normalization, which has become popular in some recent LLMs, especially as a lighter and sometimes more numerically stable alternative to LayerNorm. It normalizes only by root mean square of the features[3] instead of mean and variance. Given $X \in \mathbb{R}^{B\times L\times d_{\text{model}}}$

$$\text{RMS}_{b,l} = \sqrt{\frac{1}{d_{\text{model}}}\sum_{i=1}^{d_{\text{model}}}\mathbf{X}[b,l,i]^2 + \epsilon}$$

Then, each element in $\mathbf{X}$ is normalized by

$$\hat{X}[b,l,i] = \frac{\mathbf{X}[b,l,i]}{\text{RMS}_{b,l}}\cdot\gamma_i, \qquad \forall i \in [d],$$

where γ_i is the i-th component of the learnable scaling factor $\gamma \in \mathbb{R}^{d_{\text{model}}}$.

LLaMA (and most of Meta-Llama 1/2/3 checkpoints) swap the original LayerNorm for pre-norm RMSNorm.

Below is a complete list of RMSNorm's features.

- **(Rescales by RMS only & scaling only)**.
- **(Compute & Memory efficiency)**. 30 % cheaper per norm op; overall 2-6 % faster end-to-end in large LLMs.

[3]Features in this context refer to the elements in dimension d_{model}

- **(Numerically stable)**. Works well with very deep pre-norm Transformers.

When does it shine: (1) Ultra-deep LLMs, (2) Inference-first or edge deployments, (3) Pre-norm architectures.

RMSNorm really does skip the "subtract-the-mean" step, so a single normalization needs one less reduction-operation and a bit less memory traffic (removing the mean saves one vector reduction, one broadcast, and one add.). This skip would not affect training stability, because pre-norm residuals absorb the offset: in modern transformers RMSNorm sits before each residual branch, any mean shift can be compensated by the next linear layer's bias. To see this: consider the following procedure, (assuming $x \in \mathbb{R}^d$)

$$\boxed{x} \xrightarrow{\text{RMSNorm}} \boxed{y = \gamma \cdot \frac{x}{\|x\|_{\text{RMS}}}} \xrightarrow{\mathbf{W},b} \boxed{u = \mathbf{W}y + b} \rightarrow \boxed{z = x + u}$$

y indeed does not have mean zero, denote it as μ_y, since the next operation is affine, we can rewrite it as

$$u = \mathbf{W}(y - \mu_y \mathbf{1}) + (b + \mathbf{W}\mu_y \mathbf{1}),$$

which means that it is equivalent to adjusting the bias. During training, back-prop will simply nudge b so the network learns whatever overall shift is optimal, it does not care where that shift originates. After that u is added to the original x, but the drift in μ_y will not accumulate unchecked because: (1) The residual path still carries the original, unshifted activations, (2) The next block starts with another RMSNorm, which rescales its input (including any offset) back to a controlled RMS before new computations begin.

5 Positional Embedding

6 Precisions

Here is a summary provided by Chat-GPT o3.

Format	**Bits**	**Exp/Mant**	**Dynamic range**†	**ULP@1**	**Typical use**
FP32	32	8/23	1.2×10^{-38}-3.4×10^{38}	2^{-23}	Master weights, optimiser
FP16 (IEEE)	16	5/10	6.1×10^{-5}-6.6×10^{4}	2^{-10}	Fwd/Bwd on Volta & T4
bfloat16	16	8/7	1.2×10^{-38}-3.4×10^{38}	2^{-7}	Default on A100/H100, TPU
TF32*	19	8/10	same as FP32	2^{-10}	GEMMs on Ampere

Table 1: Mid-/high-precision dtypes widely used during LLM training.†Smallest positive *normal* value to largest finite.*TF32 is a compute mode; tensors stored as FP32.

Format	**Bits**	**Exp/Mant**	**Dynamic range**†	**ULP@1**	**Typical use**
FP8-E4M3	8	4/3	9.2×10^{-8}-4.5×10^{2}	2^{-3}	Research train / fast inf. (H100)
FP8-E5M2	8	5/2	3.0×10^{-10}-5.7×10^{4}	2^{-2}	Wider range variant (H100)
INT8 / INT4	8/4	—	±127 / ±7	1 / 1/16	Post-training inference quant.

Table 2: Very low-precision formats used for efficient training research (FP8) or deployment quantisation (INT).

A floating point number in IEEE-style formats is stored as:

```
Sign bit | Exponent bits | Mantissa (fraction) bits
```

1. **Sign bit**: (1 bit), 0 positive and 1 negative.
2. **Exponent bits**: control the scale (powers of 2). More exponent bits means wider dynamic range.
3. **Mantissa bits**: control the precision (how many distinct numbers you can represent between powers of 2). More mantissa bits means smaller ULP[4] (more precise).

A number is typically constructed in this way:

$$(-1)^{\text{sign}} \times (1.\text{mantissa bits}) \times 2^{\text{exponent}-\text{bias}}.$$

- **FP32** is used as the safe baseline in deep learning, it is almost always used for the master weigts[5] and optimizer state because it is numerically stable.
- **FP16** is used in forward/backward pass to save memory in Volta(Nvidia V100)/T4 mixed-precision training. Prone to overflow/underflow unless scaled (hence loss scaling).
- **Loss scaling** refers to the practice we multiply our loss by a large constant S before back propagation: $L_{\text{scaled}} = S \cdot L$, this scales up gradients which avoids gradient underflow. However, after computing the gradients, we divide them by S before updating the weights to restore correct magnitude. NVIDIA's AMP (Automatic Mixed Precision) does dynamic loss scaling, adjusting S automatically to prevent overflow.
- **bfloat16** is Brain floating point came from Google Brain, whose exp bits are same as **FP32**, but the mantissa bits are much fewer, this avoids the over/under-flow problem in **BF16**. This comes at a price of lower precision, but it is the default choice on A100 and H100.

  FP16 on Volta/T4 is fast but fragile, and requires loss scaling, which is why bfloat16 became popular. It enjoys the same range as FP32, but no scaling headaches.

  Its advantage includes: (i) no losss scaling needed to handle with over/under flow, (ii) half the memory of **FP32**, (iii) speed up in matmul on A100/H100, (iv) can used as a drop in for **FP32**, because the dynamic range is the same.

  For downsides: (a) precisiob loss, (b) optimizers often still keep master weights in **FP32** to avoid cumulative rounding error, (c) numerically sensitive operations such as softmax/ normalization, kernel still uses **FP32** internally.
- **TF32**: NVIDIA introduced with Ampere GPUs (A100, RTX 3000 series), default in cuBLAS/cuDNN matmul on Ampere if you pass FP32 inputs. FP32's size and range, FP16's precision.
- **FP8-E4M3, FP8-E5M2**: We can now tell directly from their names that FP8-E4M3 has a narrower dynamic range at a higher precision. For FP8-E4M3, ULP@1 is $\frac{1}{2^3} = 0.125$ and for FP8-E5M2, the ULP@1 becomes $\frac{1}{2^2} = 0.25$ which is very coarse.

  E4M3: good for weights and activations that stay within a moderate range.

  E5M2: good for gradients or loss-related values that can swing wildly in magnitude.

[4] Unit in the Last Place: The smallest possible difference between two representable floating-point numbers around a given value

[5] Master weight refers to the full-precision copy of the model's parameter that we keep during mixed-precision training.

How they (FP8) are used on H100: It can be applied both for training and inferencing.

During **training:**

1. Keep master weights in FP32 (like with FP16/bfloat16 training).
2. Cast activations/gradients to FP8 for GEMMs inside the forward/backward pass.
3. Apply per-tensor or per-channel **scaling** to map values into FP8's limited range.
4. Often mix E4M3 for forward activations, E5M2 for backward gradients.

During **Inferencing:** Post-training quantization to FP8 for ultra-fast inference with minimal memory footprint.

Scaling is not optional: FP8's numeric range is so tiny that, without actively scaling tensors before casting to FP8, we'll either hit overflow or underflow constantly. The fix is that we multiply the tensor by a scaling factor S before FP8 conversion,

$$x_{\text{scaled}} = x \times S.$$

We may choose S so that $\max x_{\text{scaled}}$ fits nicely into FP8's max normal value, and we store S as a separate FP32 number. We can later undo the the scaling after computation:

$$y = y_{\text{scaled}} \times S^{-1}.$$

This keeps numbers inside FP8's safe zone while still preserving the original magnitude relationship.

Per channel vs. per tensor: Often, there are **per channel** scaling and **per tensor** scaling. Let us image that we are doing a fully-connected layer, where we have this weight matrix $\mathbf{W} \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$. Per channel scaling means that we have a scaling factor for each output feature (each row), so there will be d_{out} scalars, while for per tensor scaling, we only have 1 scaling factor. In this sense, per tensor scaling is simpler, but will be suboptimal is some channels are loud and the others are quiet, while per channel scaling has better precisions, especially in convolutional layers or transformers where variance differs accross each head / filter.

Automation on H100: Notice that there is automated scaling on NVIDIA H100, which tracks running max values for each tensor, and picks E4M3 (more precision) for forward activations, E5M2 (more range) for backward gradients. The engine applies scaling transparently, so we mostly see FP8 without manually tuning the scaling factor S.

As an example of scaling, let us consider the previous example of fully connected layer, where we are expected to do $\mathbf{O} = \mathbf{W}\mathbf{X}$, originally, both $\mathbf{W}$ and $\mathbf{X}$ are FP32 (master weights and full precision activations). Before GEMM[6], we independently scaled them to FP8 (to fit the range) by

$$\mathbf{X}_{\text{scaled}} = \mathbf{X} \cdot S_{\mathbf{X}}; \qquad \mathbf{W}_{\text{scaled}} = \mathbf{W} \cdot S_{\mathbf{W}},$$

then we performed the matmul in FP8,

$$\mathbf{O}_{\text{scaled}} = \text{FP8GEMM}(\mathbf{W}_{\text{scaled}}, \mathbf{X}_{\text{scaled}}).$$

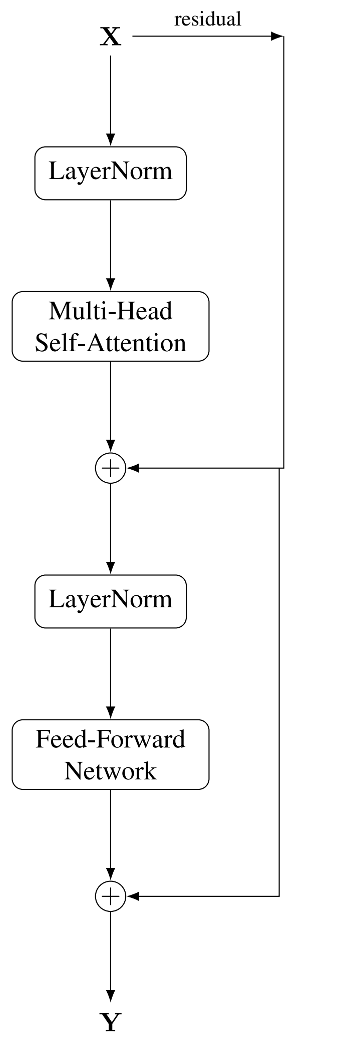
To restore the original magnitude, we take advantage of the scaling factor $S_{\mathbf{X}}, S_{\mathbf{W}}$ who are FP32,

$$\mathbf{O} = \mathbf{O}_{\text{scaled}} \cdot \frac{1}{S_{\mathbf{X}} S_{\mathbf{W}}}.$$

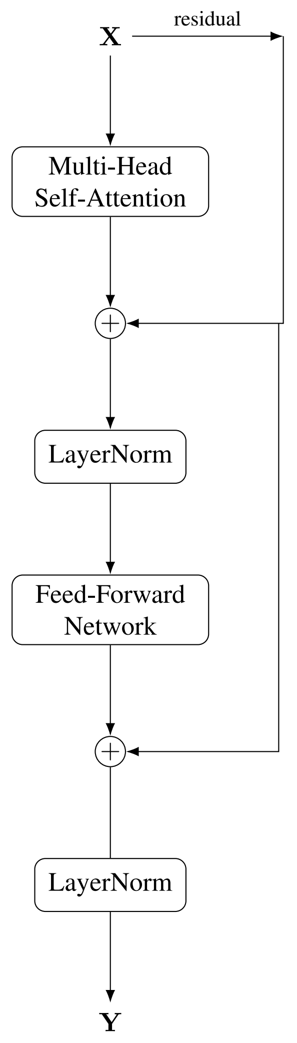
[6]GEneral Matrix-Matrix Multiplication, in the form of $\mathbf{D} = \alpha \cdot \mathbf{AB} + \beta\mathbf{C}$, where α, β are scalars. This is the workhorse of modern deep learning.

7 Encoder and Decoder Structure

Encoder Typically, encodes looks like:



Pre-LayerNorm Transformer block



Post-LayerNorm Transformer block

As we have discussed before, the Pre-LayerNorm fashion is the current state of the art as its training is more stable, as well as its convergence. It helps gradient flow, and deep model stability, does not requries learning rate warm up and works better with long context. It do has downside though, Pre-LN has a weaker normalizing effect so sometimes people add an extra final LayerNorm after the last layer for output stability ("final LN" in GPT models).

Decoder As a further explanation besides the figure: this $\mathbf{M}$ is the matrix of encoder hidden states produced by the last encoder layer. It is what the decoder attends to in cross attention. Let us image that an input $\mathbf{X} \in \mathbb{R}^{B\times L_{\text{enc}}\times d_{\text{model}}}$ is fed into the encoder blocks. After embedding and positional encoding, we have

$$\mathbf{H}^0 \in \mathbb{R}^{B\times L_{\text{enc}}\times d_{\text{model}}}.$$

Each encoder will do

$$\mathbf{H}^{k+1} = \text{Encoder}(\mathbf{H}^k) \in \mathbb{R}^{B\times L_{\text{enc}}\times d_{\text{model}}}, \quad \forall k \in [N_{\text{enc}}],$$

And $\mathbf{M}$ is exactly $\mathbf{M} = \mathbf{H}^{N_{\text{enc}}}$. As a result, for the cross attention layer: $\mathbf{K}, \mathbf{V} \in \mathbb{R}^{B \times L_{\text{enc}} \times d_{\text{model}}}$ and $\mathbf{Q} \in \mathbb{R}^{B \times L_{\text{dec}} \times d_{\text{model}}}$, so the output of decoder block will always be $\mathbb{R}^{B \times L_{\text{dec}} \times d_{\text{model}}}$.

The two types of blocks are different:

- Encoder: Self attention only, no masking, aiming to build a contextual representation of the entire source sequence.
- Decoder: Masked self attention, which prevents seeing future tokens so that the generation is autoregressive. Cross attention, which lets each target position attend to the encoder's output $\mathbf{M}$ (source context) The purpose overall is to generate the target sequence one step at a time, with access to both past target tokens and the entire source sequence.

What happens if we are using one type?

- Encoder: leaking future target tokens during training (no causal mask), so the model wouldn't learn autoregressive generation.
- Decoder: useless masked self-attention in the encoder that blocks half the context for no reason, waste compute on a cross-attention sublayer when no encoder memory exists yet.

The original seq2seq transformer: Things happens in two phases,

```
Source tokens ---> [Encoder stack] ----------------> M (memory)
                                          |
                                          v
Shifted target tokens ------> [Decoder stack] ---> Output logits
```

Encoder runs once on the full source, decoder runs once (training) or incrementally (inference), always starting with shifted target embeddings as its own input stream. The input of encoder is the source sequence $x = [x_1, x_2, ..., x_{L_{\text{enc}}}]$, after embedding we get $\mathbf{X}$, while the input of the decoder is the target sequence y so far, but shifted ($\hat{y}$) so that the model predicts the next token.

1. We take the original gold target sequence $y = [y_1, y_2, ..., y_{L_{\text{dec}}}]$.
2. Shift right by one and add the <BOS> special token, and obtain $\hat{y} = [< BOS >, y_1, ..., y_{L_{\text{dec}}-1}]$, we then embed to get $\hat{\mathbf{Y}}$ ans use it as an input to the decoder blocks.

7.1 Encoder only models

BERT, RoBERTa, DeBERTa, ELECTRA, Sentence-BERT (SBERT).

The architecture mostly are: • Stacks of encoder blocks only, • Full self attention, • Input sequence length stay fixed, • output contextualized embeddings for every token.

Purpose: Understand text: classification, regression, retrieval, token-level labeling (NER, POS tagging, QA span prediction). In those cases the model needs bidirectional context: token sees both left and right neighbors.

For many NLP tasks, we already have the full text and just need to analyze it, not generate it.

Advantages:

- **Better context capture:** every token attends to all others.
- **More efficient training** for non-generative tasks (no need to autoregress).

- **Easier fine-tuning** for classification tasks: just take the [CLS] embedding[7].

7.2 Decoder only models

GPT family (GPT-2, GPT-3, GPT-4, LLaMA, Mistral, Falcon, etc.); BLOOM, OPT, Pythia; Code generation models (CodeLLaMA, StarCoder)

Architecture: • Stack of decoder blocks only • Causal self-attention (mask future positions) • **No cross-attention** • Input length = current sequence length during generation.

Purpose: Generate text: language modeling, code generation; model learns

$$P(\text{next token} \mid \text{prev tokens})$$

Advantages:

- **Simpler**: do not need an encoder, since we aim to predict the next token given the past. Same architecture works for both pretraining (predict next token) and inference (sample next token)
- **Massive scalability**: can ingest any text corpus, no need for aligned parallel data.
- **Flexible prompts**: can condition on arbitrary text in-context.

7.3 Encoder-decoder models

often on seq2seq task, requires paired data[8].

As a summary:

Model type	**Pros**	**Cons**
Encoder-only	• Strong understanding with bidirectional context • Great for classification, retrieval, and embedding learning	• Not suited for generative tasks
Decoder-only	• Simpler architecture • Easy to train generatively • Highly flexible for any prompt-based generation	• Lacks bidirectional context • Weaker for pure understanding tasks without adaptation
Encoder–decoder	• Best for sequence-to-sequence tasks (translation, summarization, speech-to-text)	• Requires paired data • Heavier compute at inference (two stacks)

Table 3: Comparison of encoder-only, decoder-only, and encoder–decoder Transformer architectures.

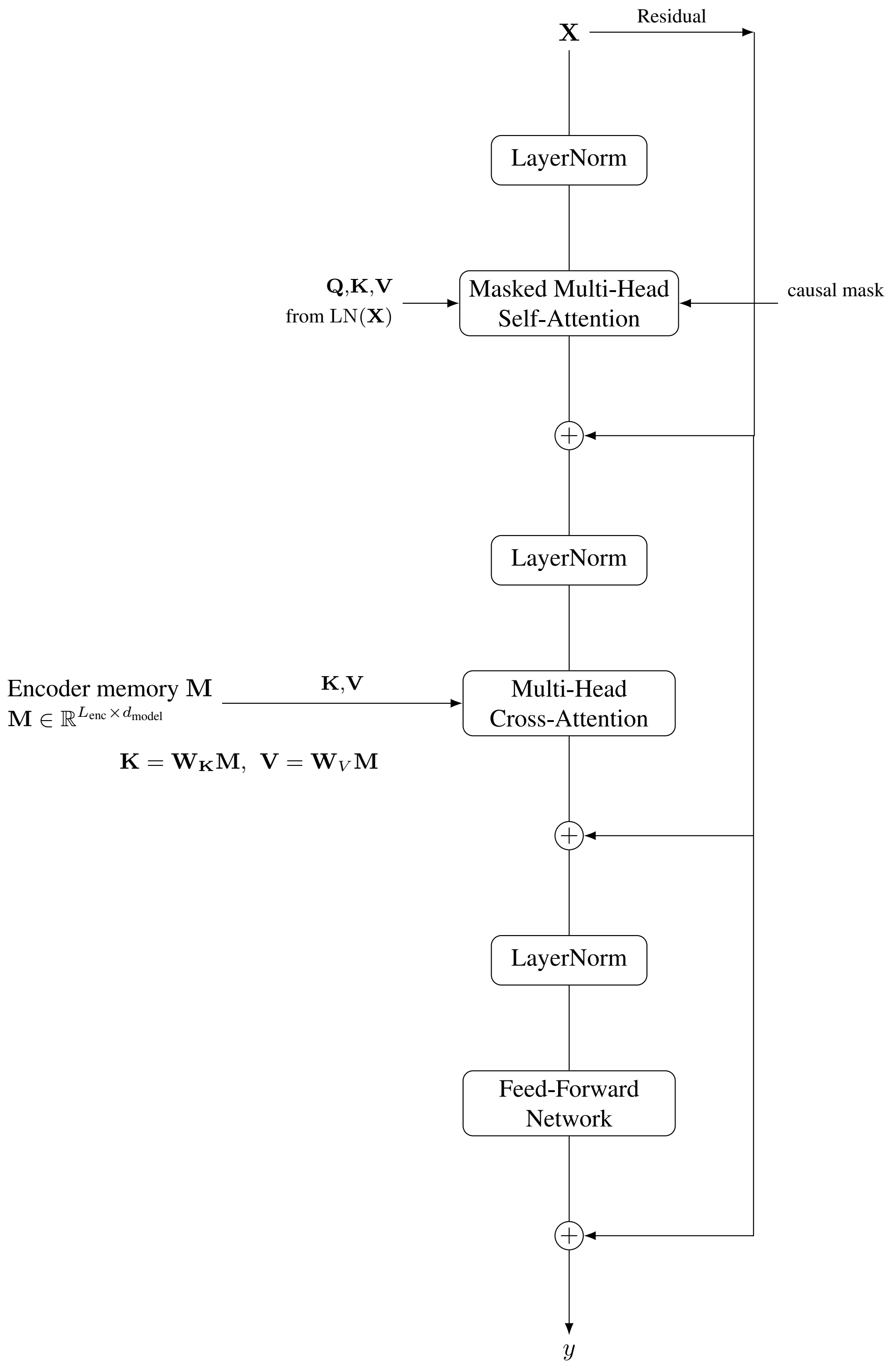
7.4 The Linear Layers

FFNs (Feed Forward Neural networks) are a part of the encoder, decoder block, there are several design choices here.

[7] In BERT and similar models, we prepend a special token [CLS] ("classification") to the start of every input sequence before feeding it to the encoder. [CLS] has its own trainable embedding vector in the model's vocabulary, just like any word. It is treated as position 0 in the sequence and goes through all encoder layers along with the other tokens. After the final encoder layer, [CLS] has a contextualized vector $h \in \mathbb{R}^{d_{\text{model}}}$, which encodes information from the entire sequence (thanks to self-attention).

[8] An example is that machine translation needs paired data: Source: "I like apples", Target: "J'aime les pommes".

8 Computing the Number of Parameters



$\mathbf{X}$
Residual
LayerNorm
$\mathbf{Q}$,$\mathbf{K}$,$\mathbf{V}$
from $\mathrm{LN}(\mathbf{X})$
Masked Multi-Head Self-Attention
causal mask
LayerNorm
Encoder memory $\mathbf{M}$
$\mathbf{M} \in \mathbb{R}^{L_{\text{enc}} \times d_{\text{model}}}$
$\mathbf{K}$,$\mathbf{V}$
$\mathbf{K} = \mathbf{W}_{\mathbf{K}}\mathbf{M},\ \mathbf{V} = \mathbf{W}_{V}\mathbf{M}$
Multi-Head Cross-Attention
LayerNorm
Feed-Forward Network
y

Figure 1: Pre-LN Transformer **Decoder** block. Each sublayer consumes a LayerNormed input. The masked self-attention applies a *causal mask* (no future tokens). Cross-attention queries come from the decoder stream, while keys/values are derived from the encoder memory $\mathbf{M}$ (encoder hidden states: $\mathbf{K} = \mathbf{W}_{\mathbf{K}}\mathbf{M}$, $\mathbf{V} = \mathbf{W}_{\mathbf{V}}\mathbf{M}$).