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 Multithreading

Per-thread scratch buffers (aka thread-local workspaces) are a very common pattern to cut allocation overhead and lock contention in multi-threaded code. In this way we do not create lots of short-lived temporaries, and have better cache locality and to avoid false sharing on shared buffers.

3 LLM Training

3.1 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 $\sqrt{E}$.

4 Attention

4.1 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 $\sqrt{d_{\text{head}}}$.

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

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 $\text{Var}(\langle q_h, k_h\rangle) \sim d_{\text{model}}\sigma_q^2\sigma_k^2$. 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 $\mathcal{O}\left(H \times BL^2 d_{\text{head}}\right) = \mathcal{O}\left(BL^2 d_{\text{model}}\right)$.

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 computational cost are the **SAME**!

4.2 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:

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

- **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.

4.3 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**.

4.4 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

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

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.

4.5 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 $\mathbf{X} \in \mathbb{R}^{B\times L\times d_{\text{model}}}$.

4.6 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).

4.7 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

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

$\gamma, \beta \in \mathbb{R}^{d_{\text{model}}}$ 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.

4.8 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)
```

4.9 DCA

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

4.10 Linear Attention

Let us first look at the asymptotic time complexity and memory complexity of attention.

Operation	**Complexity**
Projections $\mathbf{X} \mapsto (\mathbf{Q},\mathbf{K},\mathbf{V})$	$\mathcal{O}\left(BLd_{\text{model}}^2\right)$
Score computation $\mathbf{QK}^\top$ (per head)	$\mathcal{O}\left(BHL^2 d_{\text{head}}\right)$
Softmax over scores	$\mathcal{O}\left(BHL^2\right)$
Weighted sum $\mathrm{softmax}(\mathbf{QK}^\top)\mathbf{V}$	$\mathcal{O}\left(BHL^2 d_{\text{head}}\right)$
Output projection (concat heads $\to d_{\text{model}}$)	$\mathcal{O}\left(BLd_{\text{model}}^2\right)$

Table 1: Computation complexity of the main steps in a standard self-attention block. Here B = batch size, L = sequence length, H = number of heads, d_{head} = head dimension, and $d_{\text{model}} = H \cdot d_{\text{head}}$.

For naive self-attention, we can see that as sequence length becomes longer, the dominating term would be $\mathcal{O}(BHL^2 d_{\text{head}})$, which becomes $\mathcal{O}(BHL_q L_k d_{\text{head}})$, which is quadratic in L.

Memory consumption:

- Store $\mathbf{Q}, \mathbf{K}$ and $\mathbf{V}$: $\mathcal{O}(BLd_{\text{model}})$
- Store attention scores/probs: $\mathcal{O}(BHL^2)$
- Output activations for backprob add similar terms.
- By FlashAttention / PyTorch SDPA (streaming): we keep compute the same $\mathcal{O}(BHL^2 d_{\text{head}})$ but reduce memory from $\mathcal{O}(BHL^2)$ to roughly $\mathcal{O}(BLd_{\text{model}})$ by not materializing the full $L \times L$ matrix.

Scenario	Time (leading term)	Memory (leading term)
Self-attention (train)	$O(BHL^2 d_{\text{head}})$	$O(BHL^2)$
Cross-attention	$O(BHL_q L_k d_{\text{head}})$	$O(BHL_q L_k)$
With FlashAttention / SDPA	same time	$O(BLd_{\text{model}})$
Autoregressive decoding (per token at step t)	$O(Htd_{\text{head}})$	cache $O(Ld_{\text{model}})$

Table 2: Asymptotic compute and memory complexity for different attention scenarios. Here B = batch size, L = sequence length (train), H = number of heads, d_{head} = per-head dimension, $d_{\text{model}} = H \cdot d_{\text{head}}$, L_q/L_k = query/key lengths for cross-attention, and t = current decoding step in autoregressive inference. Autoregressive decoding is with KV cache.

Linear attention: The linear attention is introduced to avoid the quadratic dependence using attention kernels which allow us to obtain linear dependence. Note that the attention of token i is given by

$$\text{attn}_i = \frac{\sum_{j=1}^{L} \exp\left(\frac{\mathbf{q}_i^\top \mathbf{k}_j}{\sqrt{d_{\text{head}}}}\right) \mathbf{v}_j}{\sum_{j=1}^{L} \exp\left(\frac{\mathbf{q}_i^\top \mathbf{k}_j}{\sqrt{d_{\text{head}}}}\right)}.$$

To compute it, we need to perform $\mathbf{QK}^\top$ and apply softmax rowwise. We introduce the kernel trick just as we did in SVM, imagine we can write

$$\exp\left(\frac{\mathbf{q}_i^\top \mathbf{k}_j}{\sqrt{d_{\text{head}}}}\right) \approx \phi(\mathbf{q}_i)^\top \phi(\mathbf{k}_j),$$

for some feature map $\phi : \mathbb{R}^{d_{\text{head}}} \mapsto \mathbb{R}^{d_\phi}$ with non-negative outputs, then we have

$$\text{Numerator:} \qquad \sum_{j=1}^{L} \phi(\mathbf{q}_i)^\top \phi(\mathbf{k}_j) \mathbf{v}_j^\top = \left(\sum_{j=1}^{L} \mathbf{v}_j \phi(\mathbf{k}_j)^\top\right) \cdot \phi(\mathbf{q}_i),$$

$$\text{Denominator:} \qquad \sum_{j=1}^{L} \phi(\mathbf{q}_i)^\top \phi(\mathbf{k}_j) = \left(\sum_{j=1}^{L} \phi(\mathbf{k}_j)^\top\right) \cdot \phi(\mathbf{q}_i).$$

Now denote $\mathbf{S_V} := \sum_{j=1}^{L} \mathbf{v}_j \phi(\mathbf{k}_j)^\top \in \mathbb{R}^{d_{\text{head}} \times d_\phi}$, and $\mathbf{S_K} := \sum_{j=1}^{L} \phi(\mathbf{k}_j)^\top \in \mathbb{R}^{1 \times d_\phi}$, we would have

$$\text{attn}_i = \frac{\mathbf{S_V} \phi(\mathbf{q}_i)}{\mathbf{S_K} \phi(q_i) + \epsilon}.$$

Notice that we may compute $\mathbf{S_V}$ and $\mathbf{S_K}$ in an linear fasion dependent on $\mathcal{O}(L)$ (since there are in total L terms). In this way we get a better dependence.

Causal (autoregressive) case: All we have to change is that we maintain running sum: at time (token) t:

$$\mathbf{S}_{\mathbf{V}}^{(t)} := \sum_{j \le t} \mathbf{v}_j \phi(\mathbf{k}_j)^\top \qquad \text{and} \qquad \mathbf{S}_{\mathbf{K}}^{(t)} := \sum_{j \le t} \phi(\mathbf{k}_j)^\top.$$

and

$$\text{attn}_t = \frac{\mathbf{S}_{\mathbf{V}}^{(t)} \phi(\mathbf{q}_i)}{\mathbf{S}_{\mathbf{K}}^{(t)} \phi(q_i) + \epsilon}.$$

There are a bunch of common choices for the feature maps,

1. **Positive feature maps (kernel trick for softmax)**: (1) ELU + 1 ($\phi(x) = \text{ELU}(x) + 1$), (2) ReLU / Square ReLU, (3) Exp with scaling $\phi(x) = \exp\left(\frac{x}{d_{\text{head}}}\right)$, (4) Random Fourier features ...
2. **Other maps**: Orthogonal / normalized maps; Exponential-normalized maps ...

4.11 Sparse Attention

5 Normalizations

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

5.1 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.

5.2 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 $X \in \mathbb{R}^{B\times L\times d_{\text{model}}}$

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

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

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.
- **(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 $x \in \mathbb{R}^d$)

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

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.

6 Positional Embedding

7 Precisions

Here is a summary provided by Chat-GPT o3.

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.

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 3: 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 4: Very low-precision formats used for efficient training research (FP8) or deployment quantisation (INT).

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.

[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.

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.

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,

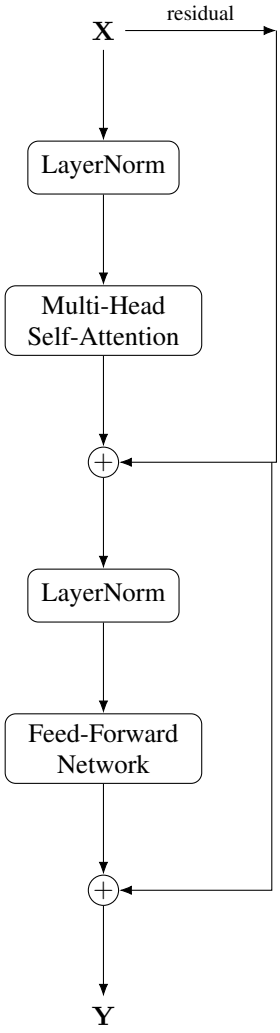
$$\mathbf{O} = \mathbf{O}_{\text{scaled}} \cdot \frac{1}{S_{\mathbf{X}} S_{\mathbf{W}}}.$$

8 Encoder and Decoder Structure

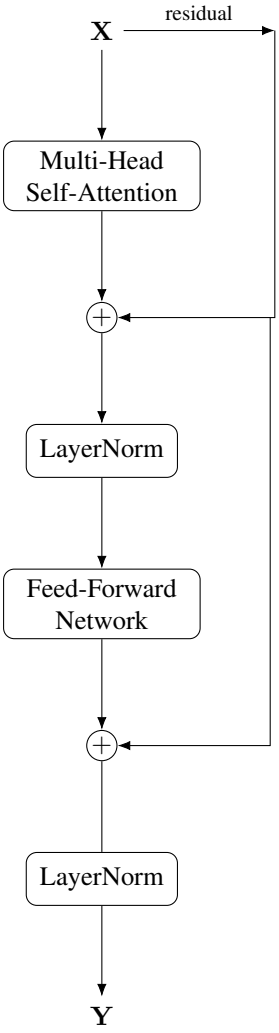
Encoder Typically, encodes looks like:

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).

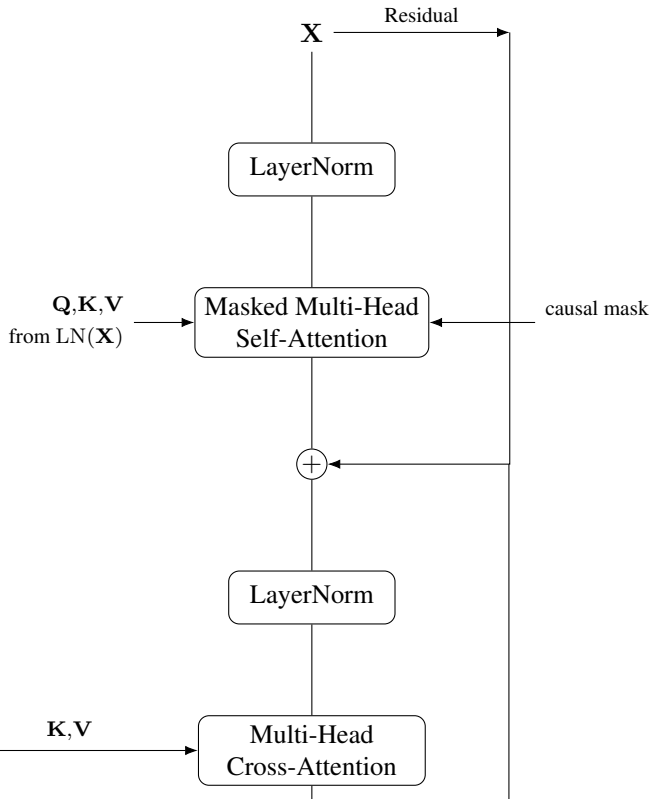
[6]GEneral Matrix-Matrix Multiplication, in the form of $\mathbf{D} = \alpha \cdot \mathbf{A}\mathbf{B} + \beta\mathbf{C}$, where α, β are scalars. This is the workhorse of modern deep learning.



Pre-LayerNorm Transformer block



Post-LayerNorm Transformer block



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 $\hat{y} = [\text{<BOS>}, y_1, ..., y_{L_{\text{dec}}-1}]$, we then embed to get $\hat{\mathbf{Y}}$ ans use it as an input to the decoder blocks.

8.1 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].

8.2 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.

8.3 Encoder-decoder models

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

8.4 Prefix-decoder models

Slightly less main stream variant in the tranformer family, similar to an interpolation between decoder-only and encoder-decoder models.

Key difference: In prefix decoder blocks, we only have one prefix self attention layer (causal self attention layer with prefix mask), and we do not have two attentions (cross attention + self attention) pattern

[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".

as in the normal decoder block. We are essentially changing the externel information from encoder to prefix region in the mask.

Imagine now we take a sequence of length $L = m + n$

$$\mathbf{x} = [\underbrace{p_1, \ldots, p_m}_{\mathbf{p}}, \underbrace{g_1, \ldots, g_n}_{\mathbf{g}}],$$

where $\mathbf{p}$ is the prefix tokens which are given in full before text generation starts. This could be tokens from a source sentence (like the "encoder output" flattened into tokens), learnable virtual tokens (prefix tuning), encoded representations from another modality (e.g., image embeddings mapped to token space),or just a long text prompt. $\mathbf{g}$ here is the generation tokens that we will generate causually one by one.

A standard decoder (decoder-only transformers) uses a strict causal mask, which is a lower triangular matrix,

$$\text{Mask}[i, j] = \begin{cases} 1 & \text{if} \quad j \leq i \\ 0 & \text{otherwise} \end{cases}$$

for all $i, j \in [L]$, this says that token i can only attends to positions $j \leq i$. While for a prefix decoder, we relaxes the prefix region

$$\text{Mask}[i, j] = \begin{cases} 1 & \text{if} \quad i \leq m \text{ and } j \leq m \qquad \text{prefix attends to all prefix} \\ 1 & \text{if} \quad i \geq m \text{ and } j \leq m \qquad \text{generation attends to all prefix} \\ 1 & \text{if} \quad i > m, m < j \leq i \qquad \text{generation attends to its own past} \\ 0 & \text{otherwise} \end{cases}$$

An example is that when we want to generate a translation for a sentence $\mathbf{t}$ (of tokens), we will use it as a prefix, and we know that $\mathbf{t}_0$ is translated into $\mathbf{s}_0$ already, in this case, our prefix would be $[\mathbf{t}_0, \text{<sep>}, \mathbf{s}_0, \text{<BOS>}]$ and our generation tokens would be $\mathbf{t}$. We include <BOS> in the prefix because it is a known token which is used to anchor.

We prefer prefix decoder sometimes, because

- **Fewer Parameters:** Parameter in the attention part drop roughly half.
- **Lower compute cost:** When prefix length is modest compared to generation length, prefix decoder can be fast overall, especially on hardware where fusing the attention into one pass matters.
- **Simpler architecture:** Do not need a separate encoder-decoder split
- **Easier caching for autoregressive generation**: In standard decoder with cross attention, we need to cache (i) decoder past key/values for masked self attention (ii) the encoder outputs for cross attention. In a prefix decoder, the prefix tokens are just part of the same KV-cache.

Down sides: (1) less flexibility in terms of shared layer stack and parameters. (2) for long prefixes, attention is more expansive.

8.5 Summary

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

8.6 The Linear Layers

FFNs (Feed Forward Networks) are a part of the encoder, decoder block, there are several design choices here. They are also known as MLPs (Multi-Layer Perceptrons), and they are used to transform the input features into a higher-dimensional space, apply non-linear activation, and then project back to the original dimension. For the 1-d case, imagine we have a vector $x \in \mathbb{R}^{d_{\text{model}}}$, then the FFN does the following:

$$\text{FFN}(x) = \mathbf{W}_2 \cdot \sigma(\mathbf{W}_1 x + b_1) + b_2,$$

where $mW_1 \in \mathbb{R}^{d_{\text{ffn}} \times d_{\text{model}}}$ is the matrix that projects the input to a higher dimension, σ is a non-linear activation function (e.g., ReLU, GELU), and $\mathbf{W}_2 \in \mathbb{R}^{d_{\text{model}} \times d_{\text{ffn}}}$ projects it back to the original dimension. d_{ffn} is often 2 $4\times$ d_{model}. Now if we consider the true input $\mathbf{X} \in \mathbb{R}^{B \times L \times d_{\text{model}}}$, then basically, for each sequence in the batch, we apply the same FFN to each token, and there will be no interactions between different tokens.

Why do we put FFNs in the encoder/decoder blocks? The FFN is used to introduce non-linearity into the model, allowing it to learn more complex representations. Self-attention is linear in the feature dimension for a fixed set of attention weights. It mixes tokens but does not increase the per-token expressivity much. To see this, consider the following attention formula:

$$\text{Attn}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \underbrace{\text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_{\text{model}}}}\right)}_{:=\mathbf{A}} \mathbf{V},$$

if the attention weight matrix $\mathbf{A}$ is fixed, then the output is a linear combination of the input $\mathbf{V}$ which is linear in $\mathbf{X}$, i.e, linear in the feature dimension.

If we **remove the FFN**, the model would reduce to mostly linear mixing layers accross tokens, which leads to collapse of expressivity (model underfits complex transformations), and much worse performance.

Design choices for FFNs: There are several design choices for FFNs, which can affect the model's performance and efficiency.

- Expansion ratio: $d_{\text{ffn}}/d_{\text{model}}$.

This is typically around 2 $4\times$. In the original Transformer paper, it is $4\times$, while in LLaMA, it is $2\times$. In many large models, FFNs account for $50\% - 60\%$ of total parameters.

- Activation function: ReLU, GELU, SiLU, etc.

The original Transformer paper uses ReLU, but GELU is more popular in modern models (BERT, GPT-2/3).

ReLU: piecewise linear, fast, but can lead to dead neurons (zero gradients for negative inputs).

$$\text{ReLU}(x) = \max(0, x).$$

GELU: a smoother, probabilistic activation function that approximates the Gaussian distribution.

$$\text{GELU}(x) = x \cdot \Phi(x) = x \cdot \frac{1}{2}\left(1 + \text{erf}\left(\frac{x}{\sqrt{2}}\right)\right),$$

where $\Phi(x)$ is the cumulative distribution function of the standard normal distribution, and erf is the error function.

- Pros: **Smooth, i.e., differentiable everywhere**[9], which enables better gradient flows. It often yields better performance in transformers, understanding and generation tasks.
- Cons: **Computationally more expensive**, sligltly more than ReLU.

SiLU: another smooth activation function, models such as EfficientNet used it.

$$\text{SiLU}(x) = x \cdot \sigma(x),$$

where $\sigma(x) = \frac{1}{1+e^{-x}}$ is the sigmoid function. It is sometimes called the Swish-1 (Swish with $\beta = 1$.)

8.7 Gated Linear Units

GLU (Gated Linear Unit) takes an input, splits it into two parts (value & gate), and uses one to gate the other via an elementwise product. Mathematically, for $\mathbf{X} \in \mathbb{R}^{B\times L\times d_{\text{model}}}$, we first apply two independent linear projections using $\mathbf{W}_g, \mathbf{W}_v \in \mathbb{R}^{d_{\text{ffn}}\times d_{\text{model}}}$,

$$\mathbf{G} = \mathbf{W}_g\mathbf{X} + \mathbf{B}_g, \qquad \mathbf{V} = \mathbf{W}_v\mathbf{X} + \mathbf{B}_v;$$

We then apply a nonlinear activation to the gate part, after which we do elementwise multiplication.

$$\mathbf{G}' = \sigma(\mathbf{G}), \qquad \text{GLU}(\mathbf{X}) = \mathbf{V} \odot \mathbf{G}'.$$

Note that we are applying activation to $\mathbf{G}$ (the gate) instead of $\mathbf{V}$, because we are applying the non-linearity here to shape its gating behaviour. This separation between gate and value allows us to learn specialized gating pattern, which acts like an learned feature selector.

SwiGLU: we are essentially replacing the original activation function $\sigma(\cdot)$ with $\text{SiLU}(\cdot)$, which is smoother. This is used in LLaMA, PaLM, etc.

GEGLU: we are essentially replacing the original activation with $\text{GELU}(\cdot)$ or its approximations with tanh.

[9] Smoothness in this context refers to infinitely differentiable functions C^∞, not the smoothness in the optimization theory sense.

8.8 Mixture-of-Experts FFNs

In a standard Transformer block, the FFN is a single feed-forward network (sometimes gated like SwiGLU) applied to every token. In MoE, instead of one FFN, we have E separate FFN "experts". For each token, a router chooses a subset (often 1 or 2) of experts to run. The outputs are combined according to the router's weights. Its advantages include

- **Scaling parameters without scaling compute:** We can have lots of experts trained but we only invoke one of them.
- **Specialization:** The model specializes experts for different types of inputs (topics, syntactic structures, etc.).

Mathematically, let $\mathbf{X} \in \mathbb{R}^{B\times L\times d_{\text{model}}}$, E be the number of experts in this case.
The forward pass:

(i) Router logits:

$$\mathbf{R} = \mathbf{W}_r\mathbf{X} + \mathbf{B}_r \quad \in \mathbb{R}^{B\times L\times E},$$

where $\mathbf{B}_r = \mathbf{1}b_r^\top$ is the extended bias matrix.

(ii) Gating probabilities:

$$\mathbf{P} = \text{softmax}(\mathbf{R}) \quad \in \mathbb{R}^{B\times L\times E}.$$

(iii) Top-k routing: Keep the best-k experts per token according to the probabilities, and renormalize. Specifically, for each token l in each batch b, denote $\mathcal{T}_k(b,l)$ as the indices of top-k experts according to $\mathbf{P}[b,l,:]$, we do

$$\tilde{\mathbf{P}}[b,l,d] = \frac{\mathbf{P}[b,l,d]}{\sum_{j=1}^{\mathcal{T}_k(b,l)}\mathbf{P}[b,l,j]}, \quad \forall d \in \mathcal{T}_k(b,l), \quad |\mathcal{T}_k(b,l)| = k,$$

and sets the other coordinates to be zero, which allows us to obtain $\tilde{\mathbf{P}}$. The renormalization here is to avoid suppressing magnitude purely because we dropped low-prob experts.

(iv) Dispatch: For each expert e, define the packed input by selecting tokens routed to e as $\mathbf{X}_e$. Let N_e be the number of tokens whose top-k includes expert e, $\pi_e(\cdot)$ be the corresponding map of expert e (from $\{1,\dots,N_e\} \mapsto \{[b,l]\}$) to the original indices pair (b,l) and let $\eta_e(\cdot)$ be the map from $\{[b,l]\}$ to the index in $[N_e]$. Then the input for an expert is

$$\mathbf{X}_e \in \mathbb{R}^{N_e\times d_{\text{model}}}, \quad \text{where} \quad \mathbf{X}_e[j,:] = \mathbf{X}[\pi_e(j),:], \qquad \forall j \in [N_e]$$

with the renormalized gate weight

$$g_e \in \mathbb{R}^{N_e}, \quad \text{where} \quad g_e[j] = \tilde{\mathbf{P}}[\pi_e(j), e].$$

(v) Expert maps: Depending on the expert type (dense FFN / Gated FFN), we create

$$\mathbf{Y}_e = \text{FFN}_e(\mathbf{X}_e) \quad \in \mathbb{R}^{N_e\times d_{\text{model}}}.$$

(vi) Gate scaling and combine: Apply per token gate (weight) to the output

$$\tilde{\mathbf{Y}}_e = \mathrm{diag}(g_e) \cdot \mathbf{Y}_e, \qquad \text{(in ML literature} \qquad g_e \odot \mathbf{Y}_e \text{ using broadcast.)}$$

Then the output $\mathbf{Z} \in \mathbb{R}^{B \times L \times d_{\text{model}}}$ would be filled such that

$$\mathbf{Z}[b, l, :] = \sum_{e \in \mathcal{T}_k(b,l)} \tilde{\mathbf{Y}}_e[\eta_e(b, l), :]$$

before outputing.

Capacity constraint: In practical implementation, we often set **capacity constraint** in its formalization, which is the limit on how many tokens each expert is allowed to process in a single forward pass. The reason to have it is

- **Load imbalance is common:** one "hot" expert could get far more tokens than the average, causing GPU memory overflow for that expert, and slower steps since we need to wait for it.
- **Hardware needs fixed allocation:** In distributed training, each expert lives on one or more GPUs, memory buffers for expert input/output must be pre-allocated. If the number of tokens per expert varies wildly, we cannot pre-allocate efficiently without wasting huge amounts of memory.

We set a capacity per expert:

$$C_e = \left\lfloor \alpha \cdot \frac{BL}{E} \right\rfloor,$$

where E is the number of experts and $\alpha \geq 1$ is the capacity factor to allow slackness. If an expert get more than C_e tokens:

1. **Dropping:** (common in Switch Transformers) Excess tokens are simply dropped for that expert: they don't get processed there, and their contribution from that expert is zeroed.
2. **Rerouting** (less common) Excess tokens are sent to a backup expert.

If an expert get fewer than C_e tokens: one may use **padding** we pad its buffer to C_e entries for vectorization but those padded entris are ignored.

Load balancing auxiliary loss: We need an extra loss term during training so that the router doesn't collapse onto just a few experts. The reasion is that, without any extra incentive, the model may prefer some experts a lot more than others, leaving many experts underused or unused. We have

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{task}} + \lambda \cdot \mathcal{L}_{\text{aux}},$$

where $\mathcal{L}_{\text{task}}$ is the task loss, and in the common GShard / Switch Transfomer[10], we have

$$\mathcal{L}_{\text{aux}} = E \cdot \sum_{e=1}^{E} f_e \cdot m_e.$$

[10]Both of these are specific MoE implementations from Google that popularized large-scale sparse FFNs. Switch is basically a simplified, more efficient GShard.

f_e is defined as the fraction of tokens assigned to expert e,

$$f_e = \frac{N_e}{BL} = \frac{1}{BL} \sum_{b=1}^{B} \sum_{l=1}^{L} \mathbf{1} \left\{ e \in \mathcal{T}_k(b, l) \right\},$$

where $\{e \in \mathcal{T}_k(b, l)\}$ is the indicator vector (sum of corresponding one hot vector) in $\mathbb{R}^{B \times L}$ suggesting if expert e is selected by this token, while m_e is defined as

$$m_e = \frac{1}{BL} \sum_{b=1}^{B} \sum_{l=1}^{L} P[b, l, e],$$

which is the average probability of assigning to expert e. One can show that the loss is smaller when f_e and m_e are more evenly distributed across experts. $\mathcal{L}_{\text{aux}}$ can be though of as an load balancing regularizer.

The all to all trick: In distributed training, experts are sharded across devices (e.g., 1 expert per GPU or multiple experts per GPU.) Naively, we must

1. Send tokens to the device hosting their assigned expert.
2. Process them locally in the expert's feed-forward network.
3. Send back the processed outputs to the original device to continue the model pipeline.

This is a typical many-to-many communication patter, where every GPU may need to send tokens to every other GPU. The most straight forward way is to let all tokens sent to all devices before filtering using a series of all_gather operations, but this can waste bandwitdh.

We can do it with the **all to all** trick, which use a collective communication primitive where each GPU directly sends only the tokens that the other GPUs need, in one coordinated call. In NCCL and similar libraries

```
torch.distributed.all_to_all_single(output, input, ...)
```

In practice, MoE frameworks do two all-to-all per MoE layer:

1. **Dispatch:** tokens $\rightarrow$ owning expert GPU.
2. **Combine:** expert outputs $\rightarrow$ original token order.

In this way, redundant communication is reduced. Real world examples includes GShard / Switch Transformer / DeepSpeed-MoE / Megatron-MoE.

Processes and threads: The legacy nn.DataParallel (**DP**) is based on one process multi-threading, which is generally slower and less scalable (single optimizer state, host-side bottlenecks, GIL contention, extra device = host hops). This is why **DDP** (multiprocess, one rank per GPU) is recommended.

Reasons:

1. **CUDA context is per-process:** Each process gets a clean, single-GPU context, which leads to simpler, safer memory management and fewer heisenbugs.
2. **Communication stacks (NCCL)** are designed around ranks = processes.

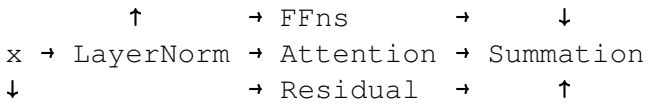
3. **Python GIL:** Python threads can't execute Python bytecode truly in parallel. Many CUDA ops release the GIL(Global Interpreter Lock), but coordination/launch logic still contends. One process per GPU avoids that bottleneck.

4. **Failure isolation:** If one GPU OOMs or a kernel errors, you don't take down every replica in the same process.

5. **Deterministic performance:** Independent schedulers per process avoid thread scheduling contention and reduce cross-device interference within a single interpreter.

8.9 Parallel Attention + FFN

The parallel design changes the flow so that attention and FFN happen at the same time on the same input (after normalization), instead of one after the other. Their outputs are summed together with the residual connection in a single step.



Why do we do this:

- **Parallelism:** Attention and FFN can be computed concurrently.
- **Fewer LayerNorms:** Saves computation.
- **Information Flow:** In sequential design, the FFN only sees the post-attention representation. In parallel, FFN sees the original representation directly, possibly preserving more raw token information.
- **Better gradient flow:** Shorter path between input and output in the computation graph.

This architecture is used in some newer LLMs, e.g. PaLM and GPT-JT, to improve speed without hurting performance.

Some practical notes:

- **Scaling:** Some designs scale the attention and FFN outputs to balance their contributions.
- **DropPath / Stochastic depth:** In residual networks, instead of always computing the residual branch, we may randomly drop it during training according to some probabilities, so that the model learns to rely on multiple paths, not just one. In parallel attention, we may consider dropping attention branch and FFN branch independently. In this way we reduces overfitting by introducing randomness, and encourages robustness.
- **Variance stability:** When we sum two residual branches instead of one, we risk blowing up the variance of activations early in training. This is another reason why we do scaling:

$$x' = x + \frac{1}{\sqrt{2}}\,(\text{attn\_out} + \text{ffn\_out})$$

We can also control by careful weight initializations, or considering set a learnable scaling parameters like α_{attn} and α_{ffn} which is small intially and train up.

9 Computing the Number of Parameters

10 Fintuning

10.1 Pretraining Loss (self-supervised)

As we know, in pretraining, we are basically doing a self-supervised learning task on predicting the next token. Given an sequence $\mathbf{x} = (\mathbf{x}_1, \ldots, \mathbf{x}_L)$, we are trying to model

$$\mathbb{P}_\theta(\mathbf{x}_1, \ldots, \mathbf{x}_L) = \prod_{t=1}^{L} \mathbb{P}_\theta(\mathbf{x}_t \mid \mathbf{x}_{<t}),$$

where θ represnets all the trainable parameters of the language model. The standard choice for it is the causal language modeling (CLM) loss:

$$\mathcal{L}_{\text{pretrain}}(\theta) = -\frac{1}{L} \sum_{t=1}^{L} \log\left(\mathbb{P}_\theta(\mathbf{x}_t \mid \mathbf{x}_{<t})\right), \tag{CLM}$$

which is the negative log-likelihood over the sequence. This can be written equivalently as the cross entropy between the model's predicted distribution $\mathbb{P}_\theta(\cdot \mid x_{<t}) \in \mathbb{R}^{|\mathcal{V}|}$ and the one-hot true token $\mathbf{y}_t \in \{0,1\}^{|\mathcal{V}|}$, where $\mathcal{V}$ is the vocabulary,

$$\mathcal{L}_{\text{pretrain}}(\theta) = -\frac{1}{L} \sum_{t=1}^{L} \sum_{v \in \mathcal{V}} (\mathbf{x}'_t)_v \log \mathbb{P}_\theta(v \mid \mathbf{x}_{<T}), \tag{Cross Entropy}$$

where $(\mathbf{x}'_t)_v = \mathbf{1}\{v = \mathbf{x}_t\} \in \mathbb{R}^{|\mathcal{V}|}$. Basically, we are summing over all possible vocabularies. The equivalence is clear once we see that, in (Cross Entropy) $\mathbf{x}'_t$ itself is a one hot vector, and it is 1 nowhere but when $v = \mathbf{x}_t$, with everywhere else 0. The training itself reduces to a **maximum likelihood** problem given a trainable set of parameter θ. A quick implementation is that

```
loss = F.cross_entropy(
    logits[:, :-1, :].reshape(-1,  vocab_size), # (B, L, V) -> (BL, V)
    tokens[:, 1:].reshape(-1) # (BL, V)
  )
```

10.2 Finetuning Loss

When adapting to downstream tasks or human preferences, loss functions differ. Typical approaches include Supervised Fine-Tuning (SFT), Parameter-Efficient Fine-Tuning (PEFT) (such as LoRA and QLoRA), Reinforcement Learning from Human Feedback (RLHF)[11], DPO / IPO, continual pretraining, domain adaptation.

10.3 Supervised Fine-Tuning (SFT)

Our goal is to make the LLM output exactly match human-provided target completions given prompts. Our loss is still cross-entropy loss, but computed **only** over the target part of the sequence.

[11] Reinforcement learning with a reward model.

Given a prompt $\mathbf{p}$ of length $L_{\mathbf{p}}$ and a target $\mathbf{y}$ of length $L_{\mathbf{y}}$, we compute

$$\mathcal{L}_{\text{SFT}} = -\frac{1}{L_{\mathbf{y}}} \sum_{t=1}^{L_{\mathbf{y}}} \log\left(\mathbb{P}_\theta\left(\mathbf{y}_t \mid \mathbf{y}_{<t}, \mathbf{p}\right)\right) = \frac{1}{L_{\mathbf{y}}} \sum_{t=1}^{L_{\mathbf{y}}} \sum_{v \in \mathcal{V}} \left(\mathbf{y}'_t\right)_v \log\left(\mathbb{P}_\theta\left(v \mid \mathbf{p}, \mathbf{y}_{<t}\right)\right), \tag{SFT}$$

where $\left(\mathbf{y}'_t\right)_v = \mathbf{1}\left\{v = \mathbf{y}_t\right\} \in \mathbb{R}^{|\mathcal{V}|}$. Notice that only completetion tokens are included in the sum (masking the prompt tokens in loss calculation). In practice we will just mask in the following way.

$$\underbrace{\text{<bos> prompt\_tokens}}_{\text{Mask}} \text{ target\_tokens <eos>}.$$

10.4 Parameter-Efficient Fine-Tuning (PEFT)

Updating all parameters of a model is costly. To resolve this, one way is to use the so called "adaptors". Other approaches include LoRA, QLoRA.

Adaptors In parameter-efficient fine-tuning (PEFT), adapters are small trainable modules inserted into a frozen pre-trained model's layers, typically after the feed-forward (FFN) or attention sublayer. The core idea is that,

- We freeze the original model weights.
- Insert a lightweight bottleneck layer (down-projection $\rightarrow$ non-linearity $\rightarrow$ up-projection) inside each transformer block.
- Train only these new parameters,

Mathematically, for an ouput $\mathbf{h} \in \mathbb{R}^{d_{\text{model}}}$ (we temporarily skip the batch and sequence length dimension as we are doing this to each token's represenetation.)

$$\text{Adapter}(\mathbf{h}) = \mathbf{h} + \mathbf{W}_{\text{up}} \cdot \sigma\left(\mathbf{W}_{\text{down}}\mathbf{h}\right),$$

We often have $\mathbf{W}_{\text{down}} \in \mathbb{R}^{d_{\text{down}} \times d_{\text{model}}}$ where $d_{\text{down}} << d_{\text{model}}$ so that the total number of trainable parameters are drastically reduced ($< 5\%$ of the full model).

Low-Rank Adaptation of LLMs Our goal here remains the same, we are trying to fine-tune large models with far fewer trainable parameters by only learning a low-rank update to certain weight matrices.

- **LoRA**

Formally, consider a weight matrix $\mathbf{W}_0 \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$ in the pre-trained model, in stead of update its pretrained parameter $\mathbf{W}_0$ directly, we keep it frozen, and learn:

$$\mathbf{W} = \mathbf{W}_0 + \Delta\mathbf{W}, \qquad \Delta\mathbf{W} = \frac{\alpha}{r}\mathbf{B}\mathbf{A},$$

where $\mathbf{A} \in \mathbb{R}^{r \times d_{\text{in}}}$, $\mathbf{B} \in \mathbb{R}^{d_{\text{out}} \times r}$, r is the low rank parameter, α is a scaling factor.

1. Usually, LoRA is applied to attention projection matrices ($\mathbf{W}_q$, $\mathbf{W}_v$ and sometimes $\mathbf{W}_k$, $\mathbf{W}_o$).
2. Rank r and scaling α are hyperparams.

Its advantage includes:

1. Drastic parameters savings.
2. No added inference latency if merged into $\mathbf{W}_0$ after training.

Notice that empirically, touching $\mathbf{W}_q$ and $\mathbf{W}_v$ is basically enough since it recovers almost all the performance of full fine-tuning, while touching fewer parameters. Intuitively, $\mathbf{W}_q$ and $\mathbf{W}_k$ affactes the matching space of attention. Since $\mathbf{QK}^\top$ is bilinear, it often suffices to change $\mathbf{W}_q$. $\mathbf{W}_v$ controls the attented values and changing it also affects the post projection controlled by $\mathbf{W}_o$.

There is a reason why do we apply a scaling factor here:

1. **Training stability:** If $\mathbf{A}$, $\mathbf{B}$ are initialized with small random values, the magnitude of their product varies with r, and without scaling, high r can produce much larger updates, destabilizing training.

2. **rank normalization** This keeps the expected variance of the update constant across different r values. α then allows manual controls (dependent on sensitivity).

Empirically $\alpha = r$ or $\alpha = 1$.

The parameter r controls the capacity of the low rank update. Small r means fewer trainable parameters and lower expressive power, works fine for minor domain shifts. Large r means more trainable parameters and expressive power, but if it is too large, then there is a risk of overfitting especially if training data is small. Empirically, $r = 4$ to 16 common for LLaMA, GPT-J, BLOOM, etc. Higher r used in complex multi-task fine-tuning. Futher improving r often gives negligible benefits.

•**QLoRA** This actually refers to Quantized LoRA, which allows fine-tuning very large language models (e.g., 65B params) on a single GPU without running out of memory. It combines 4-bit quantization for the frozen base model and LoRA.

1. Quantize $\mathbf{W}_0$ to 4-bit NormalFloat (NF4)[12], which cuts memory usage by 4.

2. Keep $\mathbf{W}_0$ frozen in quantized form.

3. Train LoRA modules $\mathbf{A}$, $\mathbf{B}$ in FP16 / bfloat16.

4. During forward pass:

   - Dequantize $\mathbf{W}_0$ on the fly.
   - Add $\Delta\mathbf{W}$ from LoRA, and proceed as usual.

Note that there are double quantization effective in QLoRA,

1. **First quantization:** The original pretrained weight tensor $\mathbf{W}_0$ is quantized is quantized from FP16/BF16 to 4-bit NF4 values plus per-block scaling factors (distribution-aware mapping, so it preserves more accuracy than uniform int4).

$$\mathbf{W}_0 = s \cdot \mathbf{W}_0^{\text{quantized}}.$$

2. **Second quantization:** The scaling factors s themselves are quantized into a lower-precision format (8-bit) plus a meta-scaling factor.

[12] It is a 4-bit quantization scheme that preserves more precision than plain int4 by using a learned normal distribution mapping.

10.5 Reinforcement Learning from Human Feedback (RLHF)

The goal is to train (finetune) a language model so its outputs align with human preferences by using reinforcement learning where the "reward" comes from human judgments.

Mathematically, let us first define the concept of the policy. Given $\mathbf{x}$ as an input sequence (prompt, context) and $\mathbf{y}$ of length T as an output sequence (continuation, response), let $\pi_0(\mathbf{y} \mid \mathbf{x})$ be defined as the probability that he model generates $\mathbf{y}$ given $\mathbf{x}$, defined token-by-token:

$$\pi_0(\mathbf{y} \mid \mathbf{x}) = \prod_{t=1}^{T} \pi_0(\mathbf{y}_t \mid \mathbf{x}, \mathbf{y}_{<t}).$$

We are using π as our notation for it is standard in reinforcement learning[13].

Stage 1: Supervised Finetuning We prepare a supervised dataset $\mathcal{D}_{\text{SFT}} = \{(\mathbf{x}_i, \mathbf{y}_i^\star)\}_{i=1}^N$ where $\mathbf{x}_i$ is prompt tokens, $\mathbf{y}_i$ is gold-standard human-written completion (target tokens). Typically, the dataset size is $10^3 - 10^6$, in this stage we do standard SFT trying to minimize the cross-entropy loss on the target tokens, which can be equivalently formulated as

$$\theta_{\text{SFT}} = \arg\min_\theta \mathbb{E}_{(\mathbf{x},\mathbf{y}^\star \sim \mathcal{D}_{\text{SFT}})} \left[-\log \pi_\theta(\mathbf{y}^\star \mid \mathbf{x}) \right],$$

where

$$\log \pi_\theta(\mathbf{y}^\star \mid \mathbf{x}) = \sum_{t=1}^{L_{\mathbf{y}^\star}} \log \pi_\theta \left((\mathbf{y}^\star)_t \mid \mathbf{x}, (\mathbf{y}^\star)_{<t} \right).$$

Basically[14], we are only computing the cross entropy loss on the human labeled completions (target sequences) without touching the prompt.

Stage 2: Reward Modelling We prepare a preference dataset $\mathcal{D}_{\text{pref}} = \left\{ (\mathbf{x}_j, \mathbf{y}_j^+, \mathbf{y}_j^-) \right\}_{j=1}^M$ where $\mathbf{x}$ is the prompt, $\mathbf{y}_j^+$ is the preferred output and $\mathbf{y}_j^-$ is the less preferred output, M is the total number of pairs of preference pairs (10^4 to 10^6) coming from human labelers ranking multiple model completions for the same prompt.

Padding: We define the reward model $R_\phi(\mathbf{x}, \mathbf{y})$ whose input is the (input prompt tokens, completion), i.e. $(\mathbf{x}, \mathbf{y})$ and outputs a scalar reward score. Typically, we start with a base language model architecture (often the same type as the policy, e.g., Transformer decoder), and we rplace the usual LM head (which produces a distribution over next tokens) with a scalar regression head. Without the language modeling head, the output is a hidden state for each $(\mathbf{x}, \mathbf{y})$

$$\mathbf{h} = (\mathbf{x}, \mathbf{y})\, 0 \in \mathbb{R}^{B \times (L_{\mathbf{x}} + L_{\mathbf{y}}) \times d_{\text{model}}}$$

One may notice that for each pair of $\mathbf{x}_i, \mathbf{y}_i$, the length $L_{\mathbf{x}_i} + L_{\mathbf{y}_i}$ is not the same, so we must do padding in order to make it a batch **before** inputting them to the model. We may pad all sequences on the left/right to ensure that they are the same length $L_{\max}$ to form a batch (not the global maximum, otherwise too wasteful)[15],

[13] In RL, statues s_t is current context (prompt + generated tokens so far), action a_t is next token to generate, policy $\pi(a_t \mid s_t)$ gives the probability distribution. The pretrained LM is thus an autoregressive policy that maps states (text histories) to action distributions (token probabilities).

[14] Here we use $(\cdot)_t$ to denote the t-the component of the vector.

[15] This is called dynamic padding and is standard in Hugging Face DataCollatorWithPadding

and record a corresponding attention mask $\mathbf{m}_i \in \{0,1\}^{B\times L_{\max}}$ (1 for real token). Often the paddings are treated as normal tokens and its embedding is either a learned vector or all zeros.

Batch Forming: Now for notational convenience, let us define

$$\mathbf{s}^+ = (\mathbf{x}, \mathbf{y}^+) \in \mathbb{N}^{L_{\mathbf{x}}+L_{\mathbf{y}^+}} \quad \text{and} \quad \mathbf{s}^- = (\mathbf{x}, \mathbf{y}^-) \in \mathbb{N}^{L_{\mathbf{x}}+L_{\mathbf{y}^-}},$$

we padded them locally to form a batch $\mathbf{S} = \left\{\hat{\mathbf{s}}_1^+, \mathbf{s}_1^-, \dots \mathbf{s}_B^+, \mathbf{s}_B^-\right\}$ where each element is now $L_{\max}$ long, with its attention mask $\mathbf{M} \in \{0,1\}^{2B\times L_{\max}}$. We feed them to the headless model and obtain

$$\mathbf{H} \in \mathbb{R}^{2B\times L_{\max}\times d_{\text{model}}}.$$

Pooling: We now need to do pooling to choose a single vector for the entire sequence to be used for pairwise comparison. We can choose

• (A) Last-token pooling (most common), which chooses the representation of the last non-PAD tokens (since it contains context from all preceding tokens because of casual attention).

• (B) Mean pooling over completions (ignore prompt tokens if we only want to summarize the generated output).

• (C) Special token pooling (Insert a <EOS> token and take its hidden state), works as [CLS] pooling in BERT.

After those steps, $\mathbf{H}$ becomes $\mathbf{P} \in \mathbb{R}^{2B\times d_{\text{model}}}$.

Scalar Reward: After we obtain $\mathbf{P}$, we apply a reward head to it, which forms the reward

$$R_\phi(\mathbf{x}, \mathbf{y}) = \mathbf{W}_r\mathbf{P} + \mathbf{b}_r \in \mathbb{R}^{2B},$$

where we get a scalar value for each sequence.

Computing Loss: We appear the pairwise Bradley-Terry loss:

$$\mathcal{L}_{\text{BT}}(\phi) = -\frac{1}{B}\sum_{i=1}^{B} \log\sigma\left(R_\phi\left(\mathbf{x}_i, \mathbf{y}_i^+\right) - \mathbb{R}_\phi\left(\mathbf{x}_i, \mathbf{y}_i^-\right)\right),$$

The above loss encourages $R_\phi\left(\mathbf{x}_i, \mathbf{y}_i^+\right)$ to be much larger than $R_\phi\left(\mathbf{x}_i, \mathbf{y}_i^-\right)$.

This is not the only way we can choose this loss, we can also pick hinge loss:

$$\mathcal{L}_{\text{Hinge}}(\phi) = \frac{1}{B}\sum_{i=1}^{B} \max\left\{0, m - \left(R_\phi\left(\mathbf{x}_i, \mathbf{y}_i^+\right) - \mathbb{R}_\phi\left(\mathbf{x}_i, \mathbf{y}_i^-\right)\right)\right\},$$

where we tries to enforce a margin $m > 0$ and if that is satified, no gradient will be applied. Compared to BT-loss, it is non-smooth.

Mean-squared error loss can also be used:

$$\mathcal{L}_{\text{MSE}}(\phi) = \frac{1}{B}\sum_{i=1}^{B} \left(R_\phi\left(\mathbf{x}_i, \mathbf{y}_i^+\right) - \mathbb{R}_\phi\left(\mathbf{x}_i, \mathbf{y}_i^-\right) - t\right)^2,$$

where t is a target gap which assumed to be known (which is also the reason why it is used).

If we have more than 2 candidates ranked, we can normalize the exponentiated rewards into a softmax, say we have $\mathbf{y}^1, \mathbf{y}^2, \dots, \mathbf{y}^K$

$$P_\phi(\mathbf{y}^j \mid \mathbf{x}) = \frac{\exp\left(R_\phi\left(\mathbf{x}, \mathbf{y}^j\right)\right)}{\sum_{j=1}^{K}\exp\left(R_\phi\left(\mathbf{x}, \mathbf{y}^j\right)\right)},$$

then minimize the cross entropy loss based on this,

$$\mathcal{L}(\phi) = -\sum_{i=1}^{M}\sum_{j=1}^{K} \text{prob\_of\_rank}_j \cdot \log\left(P_\phi(\mathbf{y}^j \mid \mathbf{x})\right),$$

where this prob_of_rank can be 1 for the top choice and 0 for others, or fractional.

Note that the starting point of ϕ is the pretrained headless model. We can choose to use the version with/without SFT, most pipelines use the one with SFT.

1. Pros 1: RM is specialized to the distribution of completions the RL stage will actually produce
2. Pros 2: KL penalty in PPO is between two models (policy and reference) that are both in-domain with respect to RM's training data
3. Cons 1: RM might overfit to the stylistic quirks of the SFT model and undergeneralize to very different policies

Stage 3: RL fine-tuning with PPO (Proximal Policy Optimization) We have the reference policy after SFT, which we denote as $\pi_{\text{SFT}} = \pi_{\text{ref}}$ and the reward model $\mathbb{R}_\phi$. Assume that now we have a prompt set $\mathcal{D}_{\text{RL}}$ which are unlabeled instructions / prompt drawn from a prompt distribution (can be the same pool used in SFT, plus extra domains).

RL objective: We intialize our policy as $\pi_\theta = \pi_{\text{ref}}$, we want to now update π_θ so that it scores high under R_ϕ and stay close to π_{ref} (to avoid drifting into low-quality or unsafe behavior), which can be described by the following KL-regularized objective:

$$\max_\theta \mathbb{E}_{\substack{\mathbf{x}\sim\mathcal{D}_{\text{RL}} \\ \mathbf{y}\sim\pi_\theta(\cdot|\mathbf{x})}} \left[R_\phi(\mathbf{x},\, \mathbf{y}) - \beta \cdot \mathrm{D}_{\text{KL}}\left(\pi_\theta(\cdot \mid \mathbf{x}) \parallel \pi_{\text{ref}}(\cdot \mid x)\right)\right], \tag{12}$$

where $\beta > 0$ is the KL-coeffficient[16]. Notice that to compute the sequence level KL divergence as given above, we may decompose it into token level contribution. For one sequence $\mathbf{y} = (\mathbf{y}_1, \ldots, \mathbf{y}_T)$ generated from prompt $\mathbf{x}$,

$$\mathrm{D}_{\text{KL}}\left(\pi_\theta\left(\cdot \mid \mathbf{x}\right) \parallel \pi_{\text{ref}}\left(\cdot \mid \mathbf{x}\right)\right) = \mathbb{E}_{\mathbf{y}\sim\pi_\theta(\cdot|\mathbf{x})}\left[\sum_{t=1}^{T} \log \frac{\pi_\theta\left(\mathbf{y}_t \mid \mathbf{x}, \mathbf{y}_{<t}\right)}{\pi_{\text{ref}}\left(\mathbf{y}_t \mid \mathbf{x}, \mathbf{y}_{<t}\right)}\right],$$

we can see that the token level contribution is given by

$$\mathrm{KL}_t\left(\mathbf{x}, \mathbf{y}_{<t}, \mathbf{y}\right) = \log \frac{\pi_\theta\left(\mathbf{y}_t \mid \mathbf{x}, \mathbf{y}_{<t}\right)}{\pi_{\text{ref}}\left(\mathbf{y}_t \mid \mathbf{x}, \mathbf{y}_{<t}\right)}.$$

If we think about it, this discourages the case that the probability of policy $\pi_\theta\left(\cdot \mid \mathbf{x}, \mathbf{y}_{<t}\right)$ generates $\mathbf{y}_t$ is smaller than that of policy $\pi_{\text{ref}}\left(\cdot \mid \mathbf{x}, \mathbf{y}_{<t}\right)$.

Sampling: We can apply certain **sampling strategies** for turning the model's per-token probability distribution into actual token choices during rollout.

• **Temperature sampling:** assume the size of the vocabulary is V, for logits $\mathbf{z}_t \in \mathbb{R}^V$ generated by the model at the current step t, we have the softmax probability

$$\mathbf{p}_i = \frac{\exp(\mathbf{z}_i)}{\sum_{j=1}^{V} \exp(\mathbf{z}_j)},$$

[16]Note that this is two nested expectation, first we can sample $\mathbf{x}$ from the dataset, then because that the policy it self is stochastic, we have multiple possible y.

however, we can add another parameter which is the so called temperature $T > 0$ and rescales the logits before softmax

$$\mathbf{p}_i^{(T)} = \frac{\exp(\mathbf{z}_i/T)}{\sum_{j=1}^{V} \exp(\mathbf{z}_j/T)}.$$

Observation:

1. $T = 1$: normal sampling from model's predicted distribution.
2. $T > 1$: flatter, more uniform, more randomness.
3. $T < 1$: sharper, more peaked, less randomness, more deterministic.

In a word, is T is high, we move to regions of smaller values which is more random, model may produce incoherent text (off-policy noise), and if T is low, model keeps generating the same high-probability responses (no diversity).

• **Top-p sampling (nucleus sampling):** We sort tokens by probability $\mathbf{p}_i$ in descending order, and take the smallest set $\mathcal{S}$ so that

$$\sum_{i \in \mathcal{S}}^{\mathbf{p}_i} \mathbf{p}_i \geq \mathbf{p}_{\text{threshold}},$$

where $\mathbf{p}_{\text{threshold}}$ (e.g. 0.9) is the top-p parameter. Its effect is to keep most probable tokens, while cut off the long tailed improbable tokens.

• **Top-k sampling:** similar to other top-K.

• We may use them in combination, such as temperature in $[0.7, 1]$, top $p \in [0.9, 0.95]$. Note that they influence not only the KL divergence but also the reward model score and the learning signal diversity.

Computation: For each t, we compute $\pi_\theta\left(\mathbf{y}_t \mid \mathbf{x}, \mathbf{y}_{<t}\right)$ and $\pi_{\text{ref}}\left(\mathbf{y}_t \mid \mathbf{x}, \mathbf{y}_{<t}\right)$ and we store the corresponding

$$\text{KL}_t\left(\mathbf{x}, \mathbf{y}_{<t}, \mathbf{y}\right) = \log \pi_\theta\left(\mathbf{y}_t \mid \mathbf{x}, \mathbf{y}_{<t}\right) - \log \pi_{\text{ref}}\left(\mathbf{y}_t \mid \mathbf{x}, \mathbf{y}_{<t}\right),$$

along the trajectory. Notice that, we have β controlling the strength of this shaping reward (penalty reward) $-\beta \cdot \text{KL}_t\left(\mathbf{x}, \mathbf{y}_{<t}, \mathbf{y}\right) := r_t^{\text{KL}}$. Often, large β means strong pull toward the reference. Finally, when we approach $t = T$, we may also include the value of $R_\phi(\mathbf{x}, \mathbf{y})$. The total reward per token is

$$r_t = \begin{cases} r_t^{\text{KL}}, & t < T \\ r_t^{\text{KL}} + R_\phi(\mathbf{x}, \mathbf{y}) & t = T. \end{cases}$$

10.6 The Bigger Picture in Reinforcement Learning

In RL theory, the formal setting is a Markov Decision Process (MDP):

$$(\mathcal{S}, \mathcal{A}, P, r, \gamma),$$

where

1. State $\mathcal{S}$: all possible situations the agent could be in. In RLHF, a state $\mathbf{s}_t = (\mathbf{x}, \mathbf{y}_{<t})$.
2. Action $\mathcal{A}$: choices the agent can make. In RLHF, it stands for picking the next token from vocabulary.

3. Transition function $P(\mathbf{s}_{t+1} \mid \mathbf{s}_t, \mathbf{a}_t)$: how actions change the state. In RLHF: appending the chosen token to the sequence.

4. Reward function $r(\mathbf{a}_t, \mathbf{s}_t)$: numeric feedback,in RLHF it is to construct KL penalty in most steps and plus the terminal reward at last.

5. Discount factor γ: how much future rewards are worth compared to immediate ones. (In RLHF we set this to 1 since episodes[17] are short.)

Objective: Assuming we have a policy $\pi_\theta(\mathbf{a} \mid \mathbf{s})$, which is a probability distribution over actions, parameterized by θ (the weights of LLM). The objective in RL is

$$\theta^\star = \arg\max_\theta J(\theta),$$

where $J(\theta)$ is defined as

$$J(\theta) := \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T} \gamma^t r(s_t, a_t)\right]. \tag{13}$$

That is maximize expected total reward. Note that the expectation actually means that we are sampling episodes (tracjetories) using π_θ and for each trajectory, we compute the total discounted reward. We average those totals over all possible trajectories, weighted by their probability of occurring under π_θ. For example, assume τ is a trajectory:

$$\tau = (\mathbf{s}_0, \mathbf{a}_0, \mathbf{s}_1, \mathbf{a}_1, \ldots, \mathbf{s}_T, \mathbf{a}_T),$$

with the probability $p_\theta(\tau)$

$$p_\theta(\tau) = p(\mathbf{s}_0) \prod_{t=0}^{T} \pi_\theta(\mathbf{a}_t \mid \mathbf{s}_t) \cdot p(\mathbf{s}_{t+1} \mid \mathbf{s}_t, \mathbf{a}_t).$$

we can rewrite $J(\theta)$ as

$$J(\theta) = \sum_\tau p_\theta(\tau) \cdot \left[\sum_{t=0}^{T} \gamma^t r(s_t, a_t)\right].$$

Notice that we can of course think of using gradient ascent to solve this. However, if we define $R(\tau) := \sum_{t=0}^{T} \gamma^t r(s_t, a_t)$, notice that R the reward function given by the environment, and it is only implicitly dependent on θ in terms of the randomness of the trajectory and if we fix one trajectory, the dependence on θ is gone. We cannot backpropagate through this function properly.

Policy Gradient: Differentiate $J(\theta)$, we get

$$\begin{aligned}
\nabla_\theta J(\theta) &= \int \nabla p_\theta(\tau) R(\tau) \quad d\tau \\
&= \int p_\theta(\tau) \nabla_\theta \log p_\theta(\tau) \cdot R(\tau) \quad d\tau \\
&= \mathbb{E}_{\tau \sim p_\theta}\left[\nabla_\theta \log p_\theta(\tau) \cdot R(\tau)\right].
\end{aligned}$$

[17] An episode is one rollout: starting from a prompt, generating tokens until we hit an end condition (EOS token or max length).

The environment dynamics $p(\mathbf{s}_{t+1} \mid \mathbf{s}_t, \mathbf{a}_t)$ and $p(\mathbf{s}_0)$ does not depend on θ which is just a const scaling on the original objective (13), we drop them and result in

$$\nabla_\theta J(\theta) = \mathbb{E}_\tau \left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta \left(\mathbf{a}_t \mid \mathbf{s}_t \right) \cdot R(\tau) \right]. \tag{A}$$

This is called the **Reinforce**. Now define the return starting at time t: G_t, which is

$$G_t := \sum_{k=t}^{T} \gamma^{k-t} r(\mathbf{s}_k, \mathbf{a}_k).$$

It is a random variable depends on the future state after t, note that

$$R(\tau) = \underbrace{\sum_{k=0}^{t-1} \gamma^k r(\mathbf{s}_k, \mathbf{a}_k)}_{\text{past reward, const given } \mathbf{s}_t.} + \gamma^t \underbrace{\sum_{k=t}^{T} \gamma^{k-t} r(\mathbf{s}_k, \mathbf{a}_k)}_{G_t}.$$

For the t-th term in the summation in (A) and (13), notice that the expectation is already conditioned on $\mathbf{a}_t$ and $\mathbf{s}_t$, if we remove the past reward, the gradient will not change since it is equivalent to remove a constant on $J(\theta)$, this allows us to obtain

$$\nabla_\theta J(\theta) = \mathbb{E}_\tau \left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta \left(\mathbf{a}_t \mid \mathbf{s}_t \right) \cdot G_t \right]. \tag{14}$$

Now defining the **action-value** function $Q^\pi(\mathbf{s}, \mathbf{a})$,

$$Q^\pi(\mathbf{s}, \mathbf{a}) := \mathbb{E}_\pi \left[G_t \mid \mathbf{s}_t = \mathbf{s}, \mathbf{a}_t = \mathbf{a} \right],$$

which is the expected return if we are at $\mathbf{s}$ state and take $\mathbf{a}$ as action now, following policy π thereafter. The expectation itself is over the random future trajectory. Using the tower property of expectation, we can further write

$$\begin{aligned}
\nabla_\theta J(\theta) &= \mathbb{E}_\tau \left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta \left(\mathbf{a}_t \mid \mathbf{s}_t \right) \cdot G_t \right] \\
&= \sum_{t=0}^{T} \mathbb{E}_{\mathbf{s}_t, \mathbf{a}_t} \left[\nabla_\theta \log \pi_\theta \left(\mathbf{a}_t \mid \mathbf{s}_t \right) \cdot \mathbb{E} \left[G_t \mid \mathbf{s}_t, \mathbf{a}_t \right] \right] = \sum_{t=0}^{T} \mathbb{E}_{\mathbf{s}_t, \mathbf{a}_t} \left[\nabla_\theta \log \pi_\theta \left(\mathbf{a}_t \mid \mathbf{s}_t \right) \cdot Q^{\pi_\theta}(\mathbf{s}_t, \mathbf{a}_t) \right]
\end{aligned}$$

Now, for variance reduction, we introduce the state-value function:

$$V^\pi(s) := \mathbb{E}_{\mathbf{a} \sim \pi_\theta(\cdot \mid \mathbf{s})} \left[Q^\pi(\mathbf{s}, \mathbf{a}) \right],$$

which is the expected return starting in state $\mathbf{s}$ following π. The advantage function is then

$$A^\pi(\mathbf{s}, \mathbf{a}) := Q^\pi(\mathbf{s}, \mathbf{a}) - V^\pi(s),$$

which measure how much better the action $\mathbf{a}$ is compare to the average action in state $\mathbf{s}$. Notice that for any baseline function $b(\mathbf{s}_t)$ that depends only on the state $\mathbf{s}_t$ but not on $\mathbf{a}_t$, we observe that

$$\begin{aligned}
\mathbb{E}_{\mathbf{s}_t,\mathbf{a}_t}\left[\nabla_\theta \log \pi_\theta\left(\mathbf{a}_t \mid \mathbf{s}_t\right) \cdot b(\mathbf{s}_t)\right] &= \mathbb{E}_{\mathbf{s}_t}\left[b(\mathbf{s}_t) \cdot \mathbb{E}_{\mathbf{a}_t}\left[\nabla_\theta \log \pi_\theta\left(\mathbf{a}_t \mid \mathbf{s}_t\right)\right]\right] \\
&= \mathbb{E}_{\mathbf{s}_t}\left[b(\mathbf{s}_t) \cdot \sum_{\mathbf{a}_t} \pi_\theta\left(\mathbf{a}_t \mid \mathbf{s}_t\right) \nabla_\theta \log \pi_\theta\left(\mathbf{a}_t \mid \mathbf{s}_t\right)\right] \\
&\overset{\spadesuit}{=} \mathbb{E}_{\mathbf{s}_t}\left[b(\mathbf{s}_t) \cdot \sum_{\mathbf{a}_t} \nabla_\theta \pi_\theta\left(\mathbf{a}_t \mid \mathbf{s}_t\right)\right] \\
&= \mathbb{E}_{\mathbf{s}_t}\left[b(\mathbf{s}_t) \cdot \nabla_\theta \sum_{\mathbf{a}_t} \pi_\theta\left(\mathbf{a}_t \mid \mathbf{s}_t\right)\right] = \mathbb{E}_{\mathbf{s}_t}\left[b(\mathbf{s}_t) \cdot \nabla_\theta 1\right] = 0
\end{aligned}$$

where $\spadesuit$ is due to chain rule. As, a result, we have

$$\nabla_\theta J(\theta) = \sum_{t=0}^{T} \mathbb{E}_{\mathbf{s}_t,\mathbf{a}_t}\left[\nabla_\theta \log \pi_\theta\left(\mathbf{a}_t \mid \mathbf{s}_t\right) \cdot \left(Q^{\pi_\theta}(\mathbf{s}_t, \mathbf{a}_t) - b(\mathbf{s}_t)\right)(\mathbf{s}_t, \mathbf{a}_t)\right].$$

Now take $b(\mathbf{s}_t) = V^{\pi_\theta}(\mathbf{s}_t)$, we result in

$$\nabla_\theta J(\theta) = \sum_{t=0}^{T} \mathbb{E}_{\mathbf{s}_t,\mathbf{a}_t}\left[\nabla_\theta \log \pi_\theta\left(\mathbf{a}_t \mid \mathbf{s}_t\right) \cdot A^{\pi_\theta}(\mathbf{s}_t, \mathbf{a}_t)\right].$$

This is called the **policy gradient theorem**. In practice, we still **cannot** compute Q^{π_θ} or V^{π_θ} exactly. So we train an auxiliary **value network** (Critic) to approximate V^{π_θ}, and we estimate $\mathbf{A}^{\pi_\theta}$ from the rollout data using methods like **Generalized Advantage Estimation** (GAE), which trades bias for variance reduction, which gives us an estimate $\hat{A}_t$. To do this, we relies on three basica building blocks:

1. **Monte Carlo (MC) return**: $G_t = \sum_{u=t}^{T} \gamma^{u-t} r_u$. This is the unbiased estimate of the true return, which is very noisy and of high variance, especially if the horizon T is long.

2. **Critic (value function)**: $V_\psi \approx V^{\pi_\theta}(\mathbf{s}_t) = \mathbb{E}\left[G_t \mid \mathbf{s}_t\right]$, as θ is consistently changing, we need to train critic per iteration.

3. **TD residual**: $\delta_t := r_t + \gamma V_\psi(\mathbf{s}_{t+1}) - V_\psi(\mathbf{s}_t)$, which is the one step error signal telling us how much better/worse things turned out compared to the critic's expectation[18]. Notice that δ_t can be viewed as 1-step estimate of the advantage function, since by definition (we get immediate reward + discounted value of next state)

$$Q^{\pi_\theta}(\mathbf{s}_t, \mathbf{a}_t) \simeq r_t + \gamma V_\psi(\mathbf{s}_{t+1})$$

From reward to policy update: We are solving a standard episodic RL problem.

Step 1: Rollout with current policy. Assume we are in iteration k, and we collect our trajectories by following π_{θ_k}, this gives us data $(\mathbf{s}_t, \mathbf{a}_t, r_t)$[19], while we are collecting we also store the log probibility of each action $\log \pi_{\theta_k}(\mathbf{a}_t \mid \mathbf{s}_t)$.

Step 2: Advantage estimation. We would like to know whether each action was better or worse than average.

[18] If the critic is perfect, we have $V_\phi(\mathbf{s}_t) = \mathbb{E}\left[r_t + \gamma V^{\pi_\theta}(\mathbf{s}_{t+1})\right]$, this is known as Bellman expectation equation which tells us what a critic would satisfy.

[19] During training, we sample many such $\mathbf{x}$ from $\mathcal{D}_{\text{RL}}$ in parallel, and collect a batch of rollouts at once.

Step 2.1: Train a critic. So we train a critic V_{ψ_k} to approximate the function value V^{π_θ}. Usually, this V_ψ can be a MLP parametrized by ψ.[20] This result in an extra loss term based on MSE loss in the final PPO loss, which we name critic.

Step 2.2: Temporal Difference (TD) residuals. We use the trained critic to compute 1-step errors

$$\delta_t = r_t + \gamma V_\psi(\mathbf{s}_{t+1}) - V_\psi(\mathbf{s}_t) = \big(\text{observed reward} + \text{predicted future value}\big) - \big(\text{predicted value at } s_t\big),$$

If $\delta_t > 0$, the return is better than expected and the action was better than average, if $\delta_t < 0$, vice versa. To understand this intuitively, we only have to remember that V_{π_k} is the expected total discounted reward you will get if you start from state $\mathbf{s}$ then following policy π_{θ_k}.

Step 2.3: Generalized Advantage Estimation (GAE). We have two choices to estimate advantage, one is the 1-step estimation generated by TD-residual, which is very noisy because it only looks one step ahead, especially when we are in a long episode and there is a large reward at the end. From an opposite point of view, if we use $G_t = r_t + \gamma r_{t+1} + \dots$ and define

$$\hat{A}_t = G_t - V_{\psi_k}(\mathbf{s}_t).$$

Although this uses the full trajectory, it introduces bias if the critic is imperfect, and is also slow to compute since we need to wait until the episodes end.

So we just combine them which results in GAE (instead of picking one horizon, we take a weighted average of all horizons). Formally,

$$\hat{A}_t = \delta_t + \gamma\lambda\delta_{t+1} + (\gamma\lambda)^2\,\delta_{t+2} + \dots,$$

where $\gamma = 0.99$ and $\lambda = 0.95$ are commom. Notice that γ is the discounting factor and λ controls how far we can look ahead. To compute is efficiently, we use backward recursion, and start with $\hat{A}_T = 0$ (by definition at the end of rollout), and

$$\hat{A}_t = \delta_t + \gamma\lambda\hat{A}_{t+1}.$$

From the estimate $\hat{A}_t$, we can reconstruct

$$\hat{G}_t = \hat{A}_t + V_\psi(\mathbf{s}_t), \tag{15}$$

which we will later use in loss.

Step 2.4: Gradient. After running GAE, we now have per-time-step advantage estimates $\hat{A}_t$, which allows us to scale the log-probability gradient

$$\nabla_\theta \log \pi_\theta(\mathbf{a}_t \mid \mathbf{s}_t) \cdot \hat{A}_t, \qquad \text{(Policy Gradient)}$$

where $\hat{A}_t$ increases the probability and vice versa.

Step 3: Policy update with PPO. Now we want to adjust π_{θ_k}, so that actions with positive advantage become more likely, and those with negative advantage less likely. If we use the naive policy gradient then the policy might change too much in one step. It could also collapse (e.g., probability mass shifts to a single action, causing instability), so we need to regularize the update. Notice that the naive objective after policy gradient transform and approximations can be written as

$$\mathcal{L}^{\text{PG}}(\theta) = \mathbb{E}_{t,\mathbf{s}_t,\mathbf{a}_t}\left[\log \pi_\theta\left(\mathbf{a}_t \mid \mathbf{s}_t\right) \cdot \hat{A}_t\right].$$

[20] In complex environments (e.g. Atari, robotics, LLMs), it can be a convolutional net, a transformer, or any architecture suited for the state representation.

Historically, we use **TRPO** (Trust Region Policy Optimization), where we do

$$\max_{\theta} \mathbb{E}_{t,\mathbf{s}_t,\mathbf{a}_t}\left[r_t(\theta;\theta_k)\hat{A}_t\right] \qquad \text{s.t.} \qquad \mathbb{E}_{\substack{\mathbf{s}\sim\rho_{\pi_{\theta_k}} \\ \mathbf{a}\sim\pi_{\theta_k}(\cdot|\mathbf{s})}}\left[\log \pi_{\theta_k}(\mathbf{a}\mid\mathbf{s}) - \log \pi_{\theta}(\mathbf{a}\mid\mathbf{s})\right] \le \delta,$$

where $\rho_{\pi_{\theta_k}}$ denotes the (on-policy) state distribution under π_{θ_k} and

$$r_t(\theta;\theta_k) := \frac{\pi_\theta(\mathbf{a}_t\mid\mathbf{s}_t)}{\pi_{\theta_k}(\mathbf{a}_t\mid\mathbf{s}_t)},$$

is the ratio. The constraint can also be written equivalently as

$$\mathbb{E}_{\mathbf{s}\sim\rho_{\pi_{\theta_k}}}\left[\mathrm{D}_{\mathrm{KL}}\left(\pi_{\theta_k}(\cdot\mid\mathbf{s}) \parallel \pi_\theta(\cdot\mid\mathbf{s})\right)\right] \le \delta.$$

There are two layers of expectations: (1) over states $\mathbf{s}$ that the old policy visits, (2) over actions $\mathbf{a}$ drawn from the old policy at each state (this is built into the KL). However, **TRPO** is too time consuming in the sense that it often requires second order optimization methods (conjugate gradient with Fisher-vector products to enforce the KL trust region).

Importance sampling based on old policy data. **PPO** (Proximal Policy Optimization) uses a trick involved the ratio function, and first replaces the objective as

$$\widehat{\mathcal{L}}^{\mathrm{PG}}(\theta;\theta_k) = \mathbb{E}_{\substack{t \\ \mathbf{s}_t\sim\rho_{\pi_{\theta_k}} \\ \mathbf{a}_t\sim\pi_{\theta_k}}}\left[r_t(\theta;\theta_k)\hat{A}^t\right].$$

If we maximize this directly, the policy can change too aggressively, leading to instability.

Clipping. As a result, **PPO** in turn introduces a clipped surrogate,

$$\mathcal{L}^{\mathrm{Clip}}(\theta;\theta_k) = \mathbb{E}_{\substack{t \\ \mathbf{s}_t\sim\rho_{\pi_{\theta_k}} \\ \mathbf{a}_t\sim\pi_{\theta_k}}}\left[\min\left\{r_t(\theta;\theta_k)\hat{A}^t, \mathrm{clip}\left(r_t(\theta;\theta_k), 1-\varepsilon, 1+\varepsilon\right)\cdot\hat{A}^t\right\}\right],$$

where the clipping operator does the following:

1. When $r_t \in [1-\varepsilon, 1+\varepsilon]$, we do nothing.
2. When r_t goes beyond, we replace r_t by the nearest boundary value.

Entropy bonus. If we only maximize rewards, the policy may collapse to a near-deterministic one (always picking the "best" action it has found so far). To counter this, we add an entropy bonus to the objective.

$$\mathcal{H}_{\pi_\theta(\cdot|\mathbf{s})} = -\sum_{\mathbf{a}} \pi_\theta(\mathbf{a}\mid\mathbf{s})\log\pi_\theta(\cdot\mid\mathbf{s}),$$

if the policy is uniform then it is high.

Full PPO loss. The clipped surrogate is the policy gradient, but PPO also trains the critic and uses an entropy bonus.

$$\mathcal{L}^{\mathrm{PPO}}(\theta,\psi;\theta_k) := \underbrace{\mathbb{E}\left[\mathcal{L}^{\mathrm{Clip}}(\theta;\theta_k)\right]}_{\text{actor}} - c_v\cdot\underbrace{\mathbb{E}\left[\left(V_\psi(s_t)-\hat{G}_t\right)^2\right]}_{\text{critic}} + c_e\cdot\underbrace{\mathbb{E}\left[\mathcal{H}\left(\pi_\theta(\cdot\mid\mathbf{s}_t)\right)\right]}_{\text{exploration}},$$

where

$$\mathbb{E}\left[\mathcal{H}\left(\pi_\theta(\cdot\mid\mathbf{s}_t)\right)\right] = \frac{1}{T}\sum_{t=0}^{T}\mathcal{H}(\pi_\theta(\cdot\mid\mathbf{s}_t)).$$

Practical hint.