Contents

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 $\sqrt{d_{\text{head}}}$. This scaling choice can be justified because each element of $\mathbf{Q}_h\mathbf{K}_h^\top$ represents a dot product between vectors of dimension d_{model}. The variance of this dot product scales as $\text{Var}\left(\langle q_h, k_h \rangle\right) \sim d_{\text{model}}\sigma_q^2\sigma_k^2$. Since variance scales quadratically, we divide by $\sqrt{d_{\text{head}}}$. The softmax operation turns the scores after masking into probabilities, along the last dimension.[2] Imagine $z = [z_1, \ldots, z_L] \in \mathbb{R}^L$ is a row vector, then essentially, softmax defines the operation:

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

   Sometimes we use a numerically stable version to replace it

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

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

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

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

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

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

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

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

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

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

4.2 Multi Query Attention (MQA)

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

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

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

Compared to standrad MHA, MQA has the following features:

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

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

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

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

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

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

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

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

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

Below is a complete list of RMSNorm's features.

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

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

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

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

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

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

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

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

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

6 Positional Embedding

6.1 Sinusoidal Positional Embedding

We just add the postion matrix $\mathbf{P} \in \mathbb{R}^{L \times d_{\text{model}}}$ to the input embedding $\mathbf{X} \in \mathbb{R}^{B \times L \times d_{\text{model}}}$, where L is the maximum sequence length. To be specific, we construct $\mathbf{P}$ as follows:

$$\mathbf{P}[l, 2i] = \sin\left(\frac{l}{10000^{\frac{2i}{d_{\text{model}}}}}\right), \quad \mathbf{P}[l, 2i+1] = \cos\left(\frac{l}{10000^{\frac{2i}{d_{\text{model}}}}}\right),$$

where $l \in [0, L-1]$ and $i \in [0, \frac{d_{\text{model}}}{2} - 1]$. This way, each dimension of the positional encoding corresponds to a sinusoid of different frequency. The wavelengths form a geometric progression. We just perform

$$\mathbf{X}' = \mathbf{X} + \mathbf{P},$$

after constructing $\mathbf{P}$.

6.2 RoPE (Rotary Positional Embedding)

RoPE, instead, directly encodes relative position information into the attention mechanism by **rotating the query and key vectors** based on their positions. Let us denote $\mathbf{Q} \in \mathbb{R}^{L \times d_{\text{head}}}$ and $\mathbf{K}^{L \times d_{\text{head}}}$ as the query matrix and the key matrix (We temporarily hide the B, H dimensions.). For each query vector

$$\mathbf{q}_l = \mathbf{Q}[l, :] = (\mathbf{q}_{l,1}, \ldots, \mathbf{q}_{l,d_{\text{head}}}) \in \mathbb{R}^{d_{\text{head}}}, \tag{12}$$

we group the dimensions into pairs

$$(\mathbf{q}_{l,1}, \mathbf{q}_{l,2}), (\mathbf{q}_{l,3}, \mathbf{q}_{l,4}), \ldots, (\mathbf{q}_{l,d_{\text{head}}-1}, \mathbf{q}_{l,d_{\text{head}}}), \tag{13}$$

assunimg d_{head} is even. This gives $\frac{d_{\text{head}}}{2}$ pairs and the same for $\mathbf{K}$. We now define angle frequencies

$$\theta_i = 10000^{-\frac{2(i-1)}{d_{\text{head}}}}, \qquad \text{for } i = 1,2,\dots \frac{d_{\text{head}}}{2}.$$

We let each pair $\mathbf{q}_i^{(l)} = (\mathbf{q}_{l,2i-1}, \mathbf{q}_{l,2i})$ be rotated so that

$$\begin{pmatrix} \mathbf{q}'_{l,2i-1} \\ \mathbf{q}'_{l,2i} \end{pmatrix} = \begin{pmatrix} \cos(l\theta_i), & -\sin(l\theta_i) \\ \sin(l\theta_i), & \cos(l\theta_i) \end{pmatrix} \begin{pmatrix} \mathbf{q}_{l,2i-1} \\ \mathbf{q}_{l,2i} \end{pmatrix} = \mathbf{R}_i(l)\mathbf{q}_i^{(l)}.$$

Notice that

$$\begin{pmatrix} \cos(s\theta_i), & -\sin(s\theta_i) \\ \sin(s\theta_i), & \cos(s\theta_i) \end{pmatrix}^\top \cdot \begin{pmatrix} \cos(t\theta_i), & -\sin(t\theta_i) \\ \sin(t\theta_i), & \cos(t\theta_i) \end{pmatrix} = \begin{pmatrix} \cos((t-s)\theta_i), & -\sin((t-s)\theta_i) \\ \sin((t-s)\theta_i), & \cos((t-s)\theta_i) \end{pmatrix}$$

Now define the rotated vector:

$$\tilde{\mathbf{q}}_l = \tilde{\mathbf{Q}}[l,:] = \mathbf{R}(l)\mathbf{Q}[l,:], \qquad \tilde{\mathbf{k}}_l = \tilde{\mathbf{K}}[l,:] = \mathbf{R}(l)\mathbf{K}[l,:], \tag{14}$$

where $\mathbf{R}(l)$ is a block diagonal rotation matrix built from $d_{\text{head}}/2$ rotation blocks, this can be intepreted as if we rotate clockwisely for angle t and counter-clockwisely for angle s, the combination of the result is to rotate clockwisely for angle $t-s$.

Key result: When computing attention, the score between query at position s and key at position t is

$$\begin{aligned} \tilde{\mathbf{Q}}[s,:]^\top \tilde{\mathbf{K}}[t,:] = \sum_{i=1}^{d_{\text{head}}/2} \left(\tilde{\mathbf{q}}_i^{(s)}\right)^\top \cdot \tilde{\mathbf{k}}_i^{(t)} &= \sum_{i=1}^{d_{\text{head}}/2} \left(\mathbf{q}_i^{(s)}\right)^\top \mathbf{R}_i(s)^\top \mathbf{R}_i(t)\mathbf{k}_i^{(t)} \\ &= \sum_{i=1}^{d_{\text{head}}/2} \left(\mathbf{q}_i^{(s)}\right)^\top \mathbf{R}_i(t-s)\mathbf{k}_i^{(t)}. \end{aligned} \tag{15}$$

As one can see, the attention score only depends on the relative position $t-s$, this suggests that the attention mechanism in this case is dependent on relative position only, which is a desirable property.[4]

Notice also that we are essentially spreading the frequencies by varying θ_i, lower index pairs have high-frequency oscillations which allows them to capture small relative displacements, while higher index pairs have low-frequency oscillations, which allows them to capture large relative displacements. The Fourier encoding of positions with exponentially decreasing frequencies means the set of basis functions spans a huge range, covering both short-term periodicity and long-range uniqueness, which is why the maximum context length of sinusoidal/RoPE embeddings is effectively determined by the lowest frequency (largest wavelength). This is similar to the sinusoidal positional embedding, but here we are directly applying it to the query and key vector.

Dynamic position range: One drawback of learned absolute positional embeddings is that they are fixed to the maximum sequence length used during training, and extrapolation to longer sequences is poor. RoPE, like sinusoidal embeddings, can generalize to longer sequences because the rotation matrices can be computed for any position index. However, the choice of base frequency (10000 in the original paper) determines the effective range of relative positions that can be represented before the rotations start to repeat.

[4] If d_{head} is odd, then there is no one pairing with the last dimension, we can simply leave it unchanged.

To be clear, at position p, for the i-th pair, the angle is $p\theta_i = p \cdot 10000^{-\frac{2(i-1)}{d_{\text{head}}}} = p \cdot \omega_i$, we say ω_i is the frequency and $\lambda_i = \frac{2\pi}{\omega_i}$ is the wavelength, ω_i decreases exponentially with i, while λ_i increases exponentially with i. For those pairs with small i, they have high frequency (small wavelength), which means they can only represent small relative positions before the rotations start to repeat, this is typically a few thousands tokens with base 10000, which is the reason why naive extrapolation beyond the trained length often fails.

In practice, there are several strategies to adjust to the desired context length

- **Position interpolation (NTK(Neural Tangent Kernel)-aware scaling).** Scale positions before feeding them to RoPE, let p' be the new position, and p be the original index, we have $p' = \frac{p}{L_{\text{new}}} \cdot L_{\text{train}}$.
- **Change base frequency.** Instead of 10000, use a larger base to spread out the frequencies more. Larger bases = slower frequencies = longer usable context, but worse local resolution.
- **Exponential extrapolation (YaRN, etc.)** Instead of letting the angle grow linearly with p, modulate it smoothly so the growth is slower and more controlled at large positions. An example of YaRN is to rescale the base frequencies non-uniformly across dimensions.
- **Hybrid interpolation + fine-tuning. NOT FINISHED!!!**

7 Precisions

Here is a summary provided by Chat-GPT o3.

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

Table 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).

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

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

1. **Sign bit**: (1 bit), 0 positive and 1 negative.
2. **Exponent bits**: control the scale (powers of 2). More exponent bits means wider dynamic range.

3. **Mantissa bits**: control the precision (how many distinct numbers you can represent between powers of 2). More mantissa bits means smaller ULP[5] (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[6] and optimizer state because it is numerically stable.
- **FP16** is used in forward/backward pass to save memory in Volta(Nvidia V100)/T4 mixed-precision training. Prone to overflow/underflow unless scaled (hence loss scaling).
- **Loss scaling** refers to the practice we multiply our loss by a large constant S before back propagation: $L_{\text{scaled}} = S \cdot L$, this scales up gradients which avoids gradient underflow. However, after computing the gradients, we divide them by S before updating the weights to restore correct magnitude. NVIDIA's AMP (Automatic Mixed Precision) does dynamic loss scaling, adjusting S automatically to prevent overflow.
- **bfloat16** is Brain floating point came from Google Brain, whose exp bits are same as **FP32**, but the mantissa bits are much fewer, this avoids the over/under-flow problem in **BF16**. This comes at a price of lower precision, but it is the default choice on A100 and H100.

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

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

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

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

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

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.

[5]Unit in the Last Place: The smallest possible difference between two representable floating-point numbers around a given value
[6]Master weight refers to the full-precision copy of the model's parameter that we keep during mixed-precision training.

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[7], we independently scaled them to FP8 (to fit the range) by

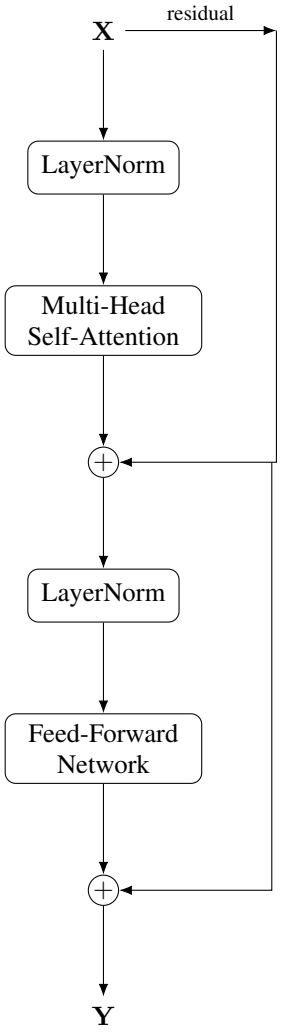
$$\mathbf{X}_{\text{scaled}} = \mathbf{X} \cdot S_{\mathbf{X}}; \qquad \mathbf{W}_{\text{scaled}} = \mathbf{W} \cdot S_{\mathbf{W}},$$

then we performed the matmul in FP8,

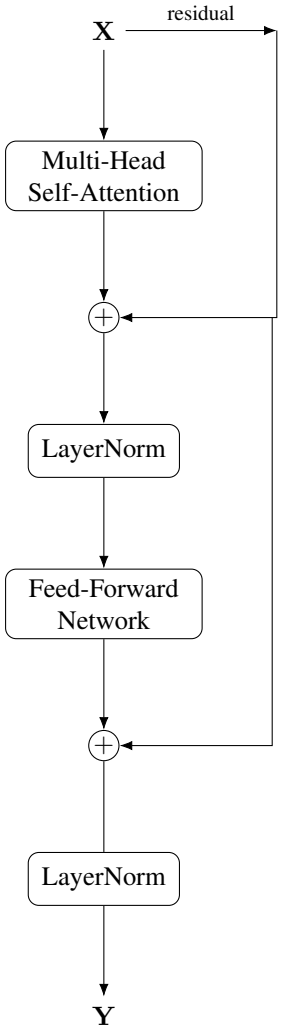
$$\mathbf{O}_{\text{scaled}} = \text{FP8GEMM}(\mathbf{W}_{\text{scaled}}, \mathbf{X}_{\text{scaled}}).$$

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

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



Pre-LayerNorm Transformer block



Post-LayerNorm Transformer block

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).

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

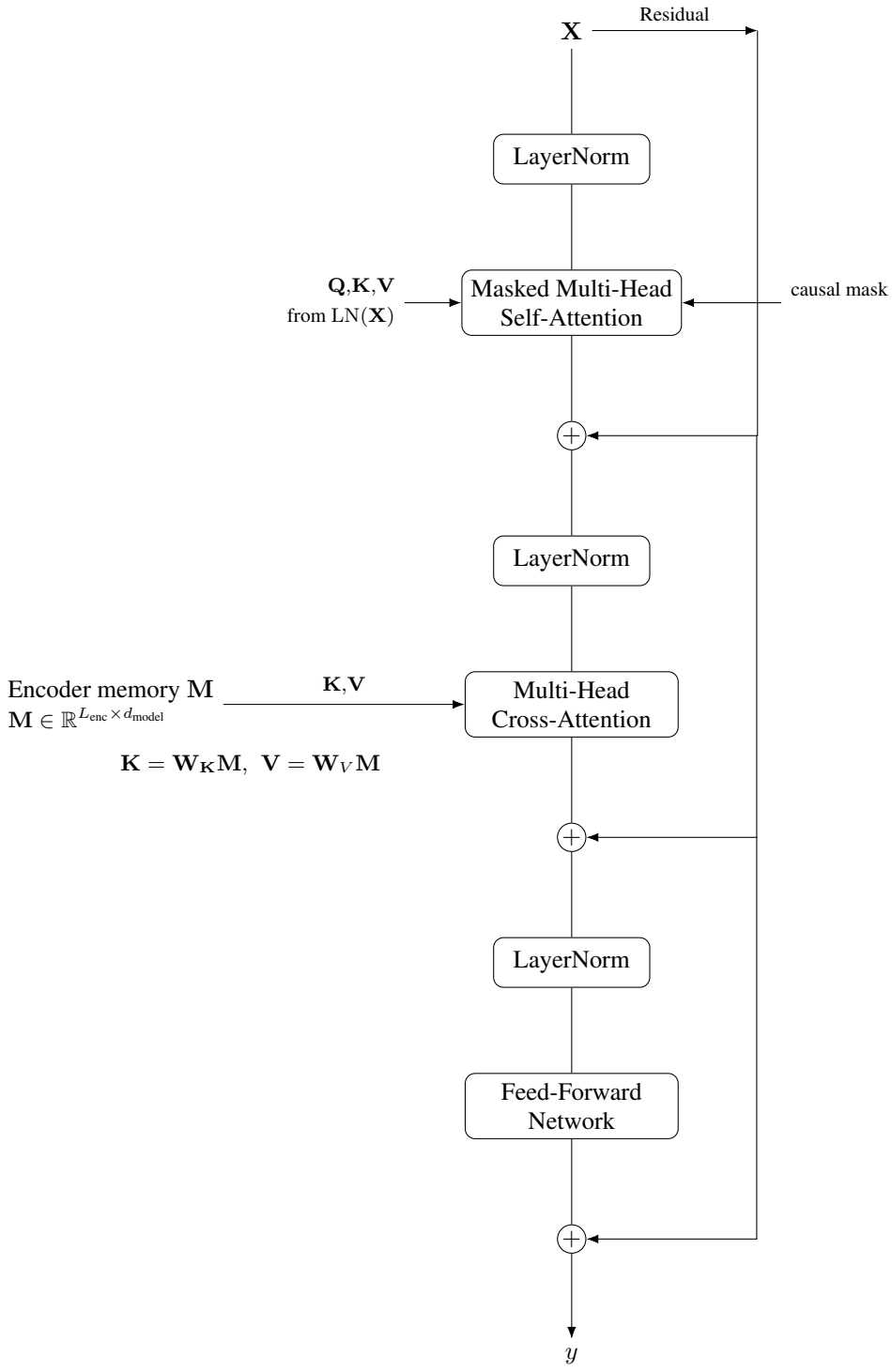


Figure 1: Pre-LN Transformer **Decoder** block. Each sublayer consumes a LayerNormed input. The masked self-attention applies a *causal mask* (no future tokens). Cross-attention queries come from the decoder stream, while keys/values are derived from the encoder memory $\mathbf{M}$ (encoder hidden states: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

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[8].

The goal of encoder-only models are to learn representations of the entire input sequence. Great for tasks like classification, NER, QA span extraction.

BERT pretraining in detail

1. **Input Preparation:**
   - **Tokenization:** Sentences are tokenized into WordPiece subwords. Special tokens are added such as [CLS] at the start (used for sequence level classification), [SEP] at the end to separate two sentences.
   - **Segment embeddings:** BERT was designed to handle pairs of sentences, this is to let the model know which tokens belong to which sentence, so we add a segment embedding on top of the token and positional embeddings. Sentence A tokens $\rightarrow$ segment embedding $\mathbf{0}$ ((vector for "sentence A").). Sentence B tokens $\rightarrow$ segment embedding $\mathbf{1}$ ((vector for "sentence B").).
   - **Position embeddings:** Added to encode order of tokens.
2. **Masked Language Modeling (MLM):**
   - **Random masking:** We select 15% of the input tokens for possible prediction. For each selected token, 80% of the time we replace it with [MASK], 10% of the time we replace it with a random word and 10% of the time we keep it unchanged. This trick prevents the model relies too heavily on [MASK] and improves robustness.
   - **Forward pass:** Feed the corrupted sentence into the encoder stack. For each masked position t, get hidden representation $\mathbf{h}_t$. Then we apply softmax over the vocabulary

     $$p_\theta(\mathbf{x}_t \mid \mathbf{x}_{\setminus t}) = \text{softmax}\left(\mathbf{W}\mathbf{h}_t\right),$$

     we then do cross-entropy only on masked positions

     $$\mathcal{L}_{\text{MLM}}(\theta) = -\sum_{t \in \text{Masked}} \log p_\theta(\mathbf{x}_t \mid \mathbf{x}_{\setminus t}),$$

     where $\mathbf{x}_{\setminus t}$ denotes the other positions than t.

[8] 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).

3. **Next Sentence Prediction (NSP):**

   - **Sentence pair construction:** 50% of the time, we set sentence B follows sentence A in corpus (label = isNext), and the other 50% of the time, sentence B is randomly sampled (label = notNext), we then construct them into the following format

     ```
     [CLS] Sentence A [SEP] Sentence B [SEP]
     ```

   - **Classification head:** we take the [CLS] embedding h_{CLS} from the final layers, and we pass through a classification layer

     $$y = \text{softmax}(\mathbf{W}_{\text{NSP}}\mathbf{h}_{\text{CLS}}),$$

     where y is the predicted probability output of the model, $y \in (0, 1)$, we will use $y^\star \in \{0, 1\}$ to denote the true label.

   - **Loss:** we use the binary classification loss,

     $$\mathcal{L}_{\text{NSP}}(\theta) = -y^\star \log y + (1 - y^\star) \log(1 - y),$$

     This is a classic binary cross entropy loss, which can be interpreted as follows, if the sentence is indeed in such a relation, then we maximize the log likelihood this is the case, otherwise if it is not in such a relation, then we maximize the log likelihood that this is not the case.

4. **Total Loss:** The two objectives are combined,

   $$\mathcal{L}(\theta) = \mathcal{L}_{\text{MLM}}(\theta) + \mathcal{L}_{\text{NSP}}(\theta),$$

   and we continue to optimize it.

5. **Extensions:** We may easily notice that BERT actually produces contextual embeddings for the entire sequence via [CLS] token or pooled output, so we can actually add a classification head to predict many kinds of relations. For examples: (i) Entailment / Contradiction (NLI) $\rightarrow$ [Premise] [SEP] [Hypothesis], (ii) Semantic similarity / Paraphrase $\rightarrow$ [Sentence1] [SEP] [Sentence2] (iii) Retrieval-style relevance $\rightarrow$ [Query] [SEP] [Document] (iv) Dialogue coherence $\rightarrow$ [Previous turn] [SEP] [Candidate response]

6. **Representations:** We note that using the [CLS] token, we can easily get a representation of the sequence (sentence) we are working with. A natural question arises, which is how we obtain such a representation for a document. In practice, if we are looking at short paragraphs (< 512 tokens: hard cap of vanilla BERT), the [CLS] token might be the right choice. For medium docs (2-3 pages), we may chunk it into 512 tokens and then using max/mean/attention pooling (learn weights to combine chunks). For larger docs, we may consider using a hierarchical BERT (imagine here sentences are tokens) or a long-document transformer.

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.

The goal of decoder-only models is to autoregressively generate text, good for open-ended generation.

8.3 Encoder-decoder models

Often on seq2seq task, requires paired data[9]. Take an input sequence, encode it, then decode into an output (good for seq2seq tasks like translation, summarization).

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 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

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

- **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**[10], 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 $\beta = 1$.)

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

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.

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 $\{1,\ldots,N_e\} \mapsto \{[b,l]\}$) 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

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

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[11], we have

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

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

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

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

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

1. **Dispatch:** tokens → owning expert GPU.
2. **Combine:** expert outputs → original token order.

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

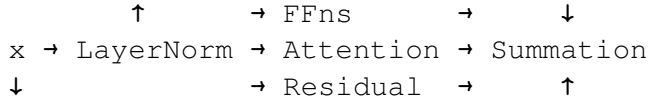
Processes and threads: The legacy nn.DataParallel (**DP**) is based on one process multi-threading, which is generally slower and less scalable (single optimizer state, host-side bottlenecks, GIL contention, extra device = host hops). This is why **DDP** (multiprocess, one rank per GPU) is recommended.

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:

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

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

Alignment: making the behavior of a large language model match human goals, values, or expectations.

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

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

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

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

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

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

where $\left(\mathbf{x}'_t\right)_v = \mathbf{1}\left\{v = \mathbf{x}_t\right\} \in \mathbb{R}^{|\mathcal{V}|}$. 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 Perplexity

Perplexity is defined as the defined as the exponential of the average loss. For each token position, we have a prediction of the true next token $\mathbf{y}_t$, our model outputs a probability distribution and the loss is the corresponding negative log likelihood given by

$$-\log p_\theta(\mathbf{y}_t \mid \mathbf{y}_{<t}).$$

The perplexity is defined as

$$\exp(\mathbb{E}\left[-\log p_\theta(\mathbf{y}_t \mid \mathbf{y}_{<t})\right]),$$

which is the exponential of the mean loss. The exponential transforms the original loss which is in the log space and brings it back to the original probability space.

Intuitively, imagine reading a sentence word by word. At each step, the model has to choose from many possible next tokens. If perplexity $= k$, it means the model is as "confused" as if it had to choose uniformly among k equally likely options at each step. That's why ppl is often described as the model's average uncertainty.

Some examples include,

1. **Perfect model** (always assigns probibility 1 to the correct token): in this case loss $= 0$, which leads to ppl $= 1$.
2. **Random guessing model** loss $= \log(\text{vocab size})$, ppl $=$ vocab size.

From this point it is easy to understand the lower the perplexity is, the better predictive power the model has.

10.3 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)[12], DPO / IPO, continual pretraining, domain adaptation.

The reason for perform such finetuning is that without such adaptations, LLMs only predict next tokens, if we ask a question, they might continue the text in a way that doesn't look like an answer.

10.3.1 Gradient Checkpointing

Normally, during backpropagation, PyTorch needs to store all intermediate activations so it can compute gradients later. That eats a lot of GPU memory. Gradient checkpointing is the technique that

- Instead of storing every intermediate activation, the model discards most of them during the forward pass.
- Later, during backward pass, it recomputes those activations on the fly.

[12] Reinforcement learning with a reward model.

- Saves GPU memory, but uses extra compute time.

This technique is useful when fine-tuning very large models on limited GPU memory, often lets you increase batch size or sequence length at the cost of some speed.

10.3.2 Gradient Accumulation

Sometimes, when the GPU does not fit the full effective batch in memory. In this case, we split the large batch into N micro-batches, and for each micro batch, we run (i) forward-backward and accumulate the gradient, but **do not** step (ii) scale the loss down by $\frac{1}{N}$ so that the accumulated gradient is correct. After N steps, we do optimizer.step() once, in this way, we can simulate a large batch size while staying within memory limits.

10.4 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.

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.

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

10.4.1 Parallelism

There are different types of parallelism, such as

- **Data Parallelism (DP):** Each GPU has a full copy of the model, different GPUs process different data batches, need to synchronize gradients so all model copies stay in sync. Examples incldue raw DDP, ZeRO-1/2/3, which is memory efficient. This is always used, and is the foundation.
- **Tensor Parallelism (TP):** Split the tensor (maths) inside a layer across GPUs, such as a giant weigt matrix $\mathbf{W} \in \mathbb{R}^{d \times d}$ can be row-sharded and column-sharded. Each forward/backward involves collective ops (all-reduce, all-gather). An example would be splitting a large weight matrix so that each GPU only holds a part of it, and they work together to do the full matrix multiplication (Each GPU only handles a fraction of the FLOPs.). Activations are then synced (e.g. via all-reduce or all-gather) to assemble the layer's output. However, this comes at the price of extra communication overhead. This is more about **mathematical parallelism** within layers. Example implementations include Megatron-LM tensor parallelism and Nvidia's Megatron-Deepspeed hybrid. This is needed when individual layers are too big for a single GPU.
- **Fully Sharded Data Parallelism (FSDP):** Shards all parameters, gradients, and optimizer states across devices. It works in a way that during forward pass: just before we need a layer's weights, they're gathered from all shards into full form, and after the forward/backward, they're sharded back. So, at any instant, only one layer's full weights need to live on a GPU, which brings memory usage down, but this comes at a price of lots of all-gather and reduce-scatter around each layer.

- **Hybrid:** One may think of combining TP + FSDP. Refer to the next subsection for an example of TP, FSDP and TP + FSDP. On top of this, we can also add DP, where an example of dimension 2 would be to divide 8 GPUs into 4 groups of 2 GPUs each, each group does TP + FSDP, and across groups we do DP.
- **Pipeline Parallelism (PP):** Split the model layers across GPUs (like an assembly line), each GPU owns a block of layers, passes activations to the next stage. There are techniques exist such as microbatching that keeps the pipeline busy. Examples include GPipe, PipeDream and DeepSpeed pipeline parallelism. This is used when the model is too deep to fit on a single GPU's memory.
- **Sequence Parallelism:** Split the sequence length dimension across GPUs.Each GPU processes a slice of the tokens. Reduces memory by distributing activations along the time axis. Examples: Megatron-LM introduced sequence parallelism for attention/MLP layers. This is used when very long context windows are present.
- **Expert / Mixture-of-Experts (MoE) Parallelism:** Explained before.

10.4.2 An example of TP, FSDP and TP + FSDP:

We will use a simple linear layer $\mathbf{Y} = \mathbf{X}\mathbf{W}$, $\mathbf{X} \in \mathbb{R}^{B\times d_{\text{in}}}$, $\mathbf{W} \in \mathbb{R}^{d_{\text{in}}\times d_{\text{out}}}$ and $\mathbf{Y} \in \mathbb{R}^{B\times d_{\text{out}}}$ to illustrate how TP, FSDP and TP + FSDP works, and consider the case of 2 GPUs.

FSDP only:

- We let GPU 0 holds $\mathbf{W}_{0:} = \mathbf{W}[0 : \frac{d_{\text{in}}}{2}, :] \in \mathbb{R}^{\frac{d_{\text{in}}}{2}\times d_{\text{out}}}$ and GPU 1 holds $\mathbf{W}_{1:} = \mathbf{W}[\frac{d_{\text{in}}}{2} : d_{\text{in}}, :] \in \mathbb{R}^{\frac{d_{\text{in}}}{2}\times d_{\text{out}}}$.
- When computing:
  1. We all-gather $\mathbf{W}$, which lets both GPUs have the full $\mathbf{W}$.
  2. Each GPU then compute $\mathbf{Y} = \mathbf{W}\mathbf{X}$ locally.
  3. Backward pass is similar, we compute local gradients, then reduce-scatter to shard the gradients back to $\mathbf{W}_{0:}$ and $\mathbf{W}_{1:}$.
- Clearly this is memory efficient since no GPU holds full $\mathbf{W}$ permanently, but each GPU still does the full FLOPs.

TP only:

- We let GPU 0 holds $\mathbf{W}_{:0} = \mathbf{W}[:, 0 : \frac{d_{\text{out}}}{2}] \in \mathbb{R}^{d_{\text{in}}\times \frac{d_{\text{out}}}{2}}$ and GPU 1 holds $\mathbf{W}_{:1} = \mathbf{W}[:, \frac{d_{\text{out}}}{2} : d_{\text{out}}] \in \mathbb{R}^{d_{\text{in}}\times \frac{d_{\text{out}}}{2}}$.
- During forward:
  1. Rank 0 computes $\mathbf{Y}_0 = \mathbf{X}\mathbf{W}_{:0} \in \mathbb{R}^{B\times \frac{d_{\text{out}}}{2}}$.
  2. Rank 1 computes $\mathbf{Y}_1 = \mathbf{X}\mathbf{W}_{:1} \in \mathbb{R}^{B\times \frac{d_{\text{out}}}{2}}$.
  3. We all-gather activations to concatenate $\mathbf{Y} = [\mathbf{Y}_0, \mathbf{Y}_1] \in \mathbb{R}^{B\times d_{\text{out}}}$.
- During backward:

1. The upstream gradient $\nabla \mathbf{Y} \in \mathbb{R}^{B \times d_{\text{out}}}$ is split consistently as $\nabla \mathbf{Y} = [\nabla \mathbf{Y}_{:0}, \nabla \mathbf{Y}_{:1}]$, where each rank receives only its local slice.
2. Each rank computes its local weight gradient:

$$\nabla \mathbf{W}_{:j} = \mathbf{X}^\top \nabla \mathbf{Y}_{:j} \in \mathbb{R}^{d_{\text{in}} \times \frac{d_{\text{out}}}{2}}, \quad j \in \{0,1\}.$$

   (No communication required.)
3. Each rank computes its partial input gradient:

$$\nabla \mathbf{X}_j = \nabla \mathbf{Y}_{:j} \mathbf{W}_{:j}^\top \in \mathbb{R}^{B \times d_{\text{in}}}.$$

4. We all-reduce (sum) these partials across ranks to obtain the full input gradient:

$$\nabla \mathbf{X} = \nabla \mathbf{X}_0 + \nabla \mathbf{X}_1.$$

   This is due to the formula based on chain rule:

$$\nabla \mathbf{X} = \nabla \mathbf{Y} \mathbf{W}^\top = \nabla \mathbf{Y}_{:0} \mathbf{W}_{:0}^\top + \nabla \mathbf{Y}_{:1} \mathbf{W}_{:1}^\top.$$

The formulas are easy to derive if we write out the sums explicitly[13].

- FLOPs are halved per GPU, Each GPU stores only its slice of $\mathbf{W}$, and we need synchronization for every layer's forward and backward.

TP + FSDP: Assume now we are in a little bit more complicated setting where we have 4 GPUs, and we want to do TP + FSDP with 2 GPUs for each.

- We will use index $i \in \{0, 1\}$ for FSDP shard and $j \in \{0, 1\}$ for TP shard.
- We first do TP splitting, where we have 2 GPUs in each TP group:

$$\mathbf{W} = [\mathbf{W}_{:0}, \mathbf{W}_{:1}], \quad \mathbf{W}_{:0} \in \mathbb{R}^{d_{\text{in}} \times \frac{d_{\text{out}}}{2}}, \mathbf{W}_{:1} \in \mathbb{R}^{d_{\text{in}} \times \frac{d_{\text{out}}}{2}}.$$

  Then we do FSDP splitting within each TP shard:

$$\mathbf{W}_{:j} = [\mathbf{W}_{0j}, \mathbf{W}_{1j}], \quad \mathbf{W}_{0j} \in \mathbb{R}^{\frac{d_{\text{in}}}{2} \times \frac{d_{\text{out}}}{2}}, \mathbf{W}_{1j} \in \mathbb{R}^{\frac{d_{\text{in}}}{2} \times \frac{d_{\text{out}}}{2}}.$$

- In the forward path:
  1. For the j-th TP block, we all-gather rows on the two GPU we have for FSDP, on each GPU, we reconstruct $\mathbf{W}_{:j} \in \mathbb{R}^{d_{\text{in}} \times \frac{d_{\text{out}}}{2}}$.
  2. We do local matmul and obtain

$$\mathbf{Y}_{:j} = \mathbf{X} \mathbf{W}_{:j} \in \mathbb{R}^{B \times \frac{d_{\text{out}}}{2}}$$

  3. Then we do TP all-gather activations, which allows us to obtain

$$\mathbf{Y} = [\mathbf{Y}_{:0}, \mathbf{Y}_{:1}] \in \mathbb{R}^{B \times d_{\text{out}}}$$

[13] A simple way of derivation would be to focus directly on the dimensions.

4. Then we do FSDP reshard, each GPU only keeps $\mathbf{W}_{ij}$, free the temporary all-gathered parameter.

- In the backward path:

1. We are given the upstream gradient

$$\nabla \mathbf{Y} = [\nabla \mathbf{Y}_{:0}, \nabla \mathbf{Y}_{:1}], \qquad \nabla \mathbf{Y}_{:j} \in \mathbb{R}^{B \times \frac{d_{\text{out}}}{2}}.$$

2. FSDP all gather rows of each $\mathbf{W}_{:j}$ again.
3. Local weight gradient, then reduce scatter back into shards,

$$\nabla \mathbf{W}_{:j} = \mathbf{X}^\top \nabla \mathbf{Y}_{:j} \in \mathbb{R}^{d_{\text{in}} \times \frac{d_{\text{out}}}{2}}.$$

and each rank (i, j) keeps $\nabla \mathbf{W}_{ij} \in \mathbb{R}^{\frac{d_{\text{in}}}{2} \times \frac{d_{\text{out}}}{2}}$.

4. Input gradient,

$$\nabla \mathbf{X}_j = \nabla \mathbf{Y}_{:j} \mathbf{W}_{:j}^\top \in \mathbb{R}^{B \times d_{\text{in}}}$$

and then all reduce to

$$\nabla \mathbf{X} = \nabla \mathbf{X}_0 + \nabla \mathbf{X}_1.$$

- The memory cost now becomes $\frac{d_{\text{in}} d_{\text{out}}}{4}$ parameters per rank, this is easy to understand, because parameters, optimizer states and gradients all correspond to the $\frac{d_{\text{in}} d_{\text{out}}}{4}$ matrix.
- The computational baseline ($\mathbf{XW}$) costs $2Bd_{\text{in}}d_{\text{out}}$ FLOPs (1 addition plus 1 multiplication). While if we do TP only, then each rank needs $2Bd_{\text{in}}\frac{d_{\text{out}}}{2} = Bd_{\text{in}}d_{\text{out}}$ FLOPs which is half of baseline, thats why TP reduced computation per rank. In the case of TP + FSDP, within each TP group, each rank still need $2Bd_{\text{in}}\frac{d_{\text{out}}}{2} = Bd_{\text{in}}d_{\text{out}}$ because of internal FSDP, so the computation per rank is still halved. Notice that in this case, the total FLOPs is doubled, compared to the baseline, which is $4Bd_{\text{in}}d_{\text{out}}$, per rank is halved is only possible because we have 4 GPUs.

10.4.3 Common Distributed Training Frameworks

PyTorch DDP: Currently, the baseline of finetuning models in parallel is to use raw PyTorch DDP (distributed data parallel), where we have 1 GPU for each process.

Gradients are averaged via "torch.distributed.all_reduce".

Pros: • Native PyTorch, minimal dependencies, and we have full control over everything.

Cons: • We must handle everything, checkpointing, gradient accumulation, optimizer sharding and memory scaling, which makes it hard to handle for very large models (10B +)

Hugging face accelerate: This is pretty common nowadays, basically, we just use the PyTorch training loop and add decorators, now it would run on one or serveral GPUs, with optional DeepSeed/FSDP integration.

Under the hood, it is still DDP, but we

- Handles GPU assignment, logging only on rank 0, checkpoint saving.
- Can automatically switch to DeepSpeed ZeRO or FSDP with a config file.
- Hugely reduces boilerplate.

Common use cases include: SFT of 7B - 13B models across 4-8 A100 GPUs.

DeepSpeed ZeRO This enables massive models by sharding optimizer, gradients and parameters across GPUs. Its key idea is ZeRO which is zero redundancy optimizer, it includes stages:

- Stage 1: shard optimizer states.
- Stage 2 (ZeRO2): shard optimizer states + gradients. (Splits gradients and optimizer states accross GPUs, each GPU still holds a full copy of the model parameter.)
- Stage 3 (ZeRO3 with offload): shard everything. (including parameters) (Splits parameters, gradients and optimizer states across GPUs, no single GPU holds the full model, which makes train very large models even with limited GPUs possible.)
- **Offload**: Store some of the states in CPU RAM (or NVMe SSD) and bring them in when needed.

and extras such as CPU/NVMe offloading, activation checkpointing, communication optimizations.

Use cases: Training 13B-70B models across 64-512 GPUs, needed when VRAM is limiting, even with 80GB A100s. Often combined with LoRA/QLoRA for finetuning.

PyTorch FSDP This is basically a native PyTorch way to do what DeepSpeed ZeRO does. It works in a way that it (i) shards parameters and gradients across ranks, (ii) loads only parts of the model per GPU at a time. This also supports activation checkpointing.

HuggingFace Trainer Where we do not write loop, just pass datasets + model + arguments.

As a summary

- Small models ($\leq$ 13B), we do SFTTrainer + LoRA + Accelerate (4-8) GPUs.
- Medium models (13B - 30B), we do Accelerate + DeepSpeed ZeRO-2/FSDP.
- Large models ($\geq$ 65B), DeepSpeed ZeRO-3 or FSDP across hundreds of GPUs. HuggingFace Trainer or custom PyTorch loop orchestrated by Accelerate.. Always with mixed precision (bf16/FP8)

10.4.4 Examples

Let us consider the following cases:

Case 1: How raw DDP works.

- **No ZeRO (normal DDP):** In this case, each GPU keeps a full copy of all N parameters, during training, each GPU does forward backward on its minibatch and gradients are averaged (all-reduce). The memory cost should be: each GPU stores N parameters + all optimizer states. The problem this that this does not scale to very large models, because each GPU must hold the entire model.

Case 2: How ZeRO2 works.

- **ZeRO2:** In this case, each GPU still has a full copy of the N parameters, but optimizer states, such as Adam's momentum m and second momentum v are sharded across GPUs. The memory cost would be the N parameters + only a slice of the optimizer states and grads, we save memory, but the full model still sits on every GPU.

Assuming there are 2 GPUs with N parameters and each GPU is proceeding with a microbatch of size 1 with gradient accumulation step 4.

1. Forward + backward on GPU 0, 1 with micro-batch=1, computing all gradients, then we do reduce scatter[14] (instead of all reduce in plain DDP), and each GPU holds a shard of gradients of size $\frac{N}{2}$, the shard is accumulated and later used to update the (second order) momentum.
2. We repeat the process so that the gradients shards gets accumulated.
3. At each optimizer step, every GPU only updates its shard of parameters using its shards of grads.
4. After the update, the parameters are all gathered, so all GPUs gets the fully freshed model for the next forward.

Basically, we are asking each GPU to be responsible for less parameters because some of them are just simply redundant, the final result is mathematically equivalent:

$$\theta_{\text{DDP}}^{t+1} = \theta_{\text{ZeRO-2}}^{t+1}. \tag{16}$$

Case 3: How ZeRO3 works.

- Now we also shard the parameters themselves, with Adam, we only need about $1/3$ of the memory on 1 GPU compared to that of DDP, instead of $2/3$ in ZeRO-2.

Assume we are in the same setting as before.

1. Forward passes: since now each GPU does not hold all params for a certain layer L (tensor parallelism) anymore, we need to call all gather for each GPU and they will temporarily assemble the full weight matrix for that layer, and the forward for layer L is computed locally on each GPU. After this layer is done, those gathered weights can be released (kept only in CPU/NVMe if offloading).
2. Backward passes: Before computing the grads for layer L, weights are all gathered and gradients are computed after that.
3. Reduce-Scatter: Instead of all reduce, gradients are reduce-scattered, so each GPU only keeps its shard.
4. Optimizer Step: Each GPU only updates its shard of optimizer state and parameters, no one ever touches the other shard.

Baically ZeRO-3 saves GPU memory but does more communication.

Case 4: How ZeRO-3 + CPU/NVMe offload works. The same sharding applies, but now we decide where the shards live when they're not in active use.

Assuming we are in the same setting as before.

1. **Parameter Offload:** Sharded params that are not immediately needed for the current layer can be moved to CPU RAM (or NVMe if RAM is still too small).[15] Before a layer executes, ZeRO all-gathers the shards back onto GPU(s). After use, they can be evicted back to CPU.
2. **Optimizer State Offload:** Adam's momentum states are big (2x param size), offloading lets each GPU keep only its shard in CPU RAM/NVMe, when optimizer step happens, that shard is pulled onto GPU, updated, and can be offloaded again. (This saves a huge amount of persistent GPU memory.)
3. **Gradient Offload:** Typically not offloaded explicitly but can be done, since grads are transient. They live in GPU memory just long enough to accumulate (over gradient accumulation steps).

[14] Sharded after all reduce (averaging).

[15] Originally, each GPU has to carry it otherwise it is gone. Now we store them in CPU and only load back the layer used. At each time, GPU memory only holds the weights for the current layer but not the whole shards.

10.5 Instruction Tuning

As we have said, pretrained LLMs (like GPT, LLaMA, Falcon, etc.) are trained on massive text corpora with the next-token prediction objective. They learn a lot of knowledge and general linguistic ability, but they are not naturally aligned with human intent, if we ask for a list or code snippet, they may not format it correctly or if we ask a question, they might continue the text in a way that doesn't look like an answer. Instruction tuning is a way to bridge the gap: we explicitly train the model to respond to natural-language commands (prompts) in the way humans expect. Formally, instruction tuning $\iff$ SFT of a pretrained LLM on a dataset of **(instruction, response)** pairs. The dataset is some times called an instruction set $\mathcal{D}_{\text{instr}}$. Let $\mathbf{x}$ be instructions and $\mathbf{y}$ be a desired response, then

$$\mathcal{L}^{\text{IT}} = - \sum_{(\mathbf{x},\mathbf{y})\in\mathcal{D}_{\text{instr}}} \log \pi_\theta \left(\mathbf{y} \mid \mathbf{x}\right),$$

which is exactly the same as the standard language modelling loss used in finetuning (SFT).

Common instruction datasets FLAN (Google): mixture of 60 NLP tasks rephrased as natural language instructions; Alpaca: 52k instruction-response pairs generated from GPT-3; OpenAssistant, Dolly, UltraChat, ShareGPT, etc.

10.6 Continued Pretraining

Continuted pretraining is the process of taking a large pretrained language model (LM) and further training it with the same pretraining objective (causal LM, masked LM, span corruption, etc.) but on a new corpus that represents the target domain, task data, or language. This is an unsupervised adaptation method: no labels or preference data are needed. The goal is to shift the model's distribution closer to the new data distribution without discarding general knowledge.

For a formal definition, let us image that we have θ_0, which is the pretrained parameters from the base model, denote P_{base} as the distribution of the original pretraining data, and P_{target} as the distribution of the new data corpus (domain/task/language). The continued pretraining then solves:

$$\theta^\star = \arg\min_\theta \mathbb{E}_{\mathbf{x}\sim P_{\text{target}}} \left[\mathcal{L}\left(\theta; x\right)\right].$$

which is identical on the original pretraining objective but with a shifted data distribution.

10.7 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 $\mathbf{Q}\mathbf{K}^\top$ 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.

Usually, we initialize the matrix $\mathbf{A}$ with normal initialization like kaiming_uniform_, but we initialize $\mathbf{B}$ as all zeros. The reason is that we need to prevent the activations we get from $\mathbf{A}$ from collapsing and we also need to make sure that initially the LoRA adaptor contributed nothing to the pretrained language model, by initializing $\mathbf{B}$ as all zero.

```
self.lora_A = nn.Parameter(torch.zeros(r, d_in))
self.lora_B = nn.Parameter(torch.zeros(d_out, r))
nn.init.kaiming_uniform_(self.lora_A, a=math.sqrt(5))
nn.init.zeros_(self.lora_B)
```

• **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)[16], 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 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.

10.8 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[17].

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

[17] 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).

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[18], 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 replace 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 $(\mathbf{x}, \mathbf{y})$

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

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)[19], 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 $\mathbf{S} = \left\{\hat{\mathbf{s}}_1^+, \mathbf{s}_1^-, \ldots \mathbf{s}_B^+, \mathbf{s}_B^-\right\}$ 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).

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

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

• (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:

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

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 $\mathbf{y}^1, \mathbf{y}^2, \ldots, \mathbf{y}^K$

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

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 using PPO (Proximal Policy Optimization) with KL regularization. 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:

$$\max_\theta \mathbb{E}_{\substack{\mathbf{x}\sim\mathcal{D}_{\text{RL}} \\ \mathbf{y}\sim\pi_\theta(\cdot|\mathbf{x})}} \left[R_\phi(\mathbf{x},\, \mathbf{y}) - \beta \cdot \mathrm{D}_{\text{KL}}\left(\pi_\theta(\cdot \mid \mathbf{x}) \parallel \pi_{\text{ref}}(\cdot \mid x)\right)\right], \tag{17}$$

where $\beta > 0$ is the KL-coeffficient[20]. 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 $\pi_{\text{ref}}\left(\cdot \mid \mathbf{x}, \mathbf{y}_{<t}\right)$. In practiccal training, rollouts (states, actions) are sampled from the old policy in the current outer iteration θ_{old}, which is the reason why the advantage-weighted likelihood ratio ($o_t(\theta; \theta_k)$ defined later) appears.

About KL divergence: Token level definition of KL divergence : at timestep t, the KL divergence between current policy π_θ and the reference π_{ref} is

$$\mathrm{D}_{\text{KL}}\left(\pi_\theta\left(\cdot \mid \mathbf{x}, \mathbf{y}_{<t}\right) \parallel \pi_{\theta_{\text{ref}}}\left(\cdot \mid \mathbf{x}, \mathbf{y}_{<t}\right)\right) = \sum_{\mathbf{a}} \pi_\theta\left(\mathbf{a} \mid \mathbf{x}, \mathbf{y}_{<t}\right) \cdot \log\left(\frac{\pi_\theta\left(\mathbf{a} \mid \mathbf{x}, \mathbf{y}_{<t}\right)}{\pi_{\theta_{\text{ref}}}\left(\mathbf{a} \mid \mathbf{x}, \mathbf{y}_{<t}\right)}\right),$$

which is an expectation over all possible next tokens $\mathbf{a}$. The previously defined log ratio $\mathrm{KL}_t\left(\mathbf{x}, \mathbf{y}_{<t}, \mathbf{y}\right)$ is an estimate of this true $\mathrm{D}_{\text{KL}}\left(\pi_\theta\left(\cdot \mid \mathbf{x}, \mathbf{y}_{<t}\right) \parallel \pi_{\theta_{\text{ref}}}\left(\cdot \mid \mathbf{x}, \mathbf{y}_{<t}\right)\right)$ based on the one token $\mathbf{y}_t$, which is a random variable. We have

$$\mathbb{E}_{\mathbf{y}_t\sim\pi_\theta}\left[\mathrm{KL}_t\left(\mathbf{x}, \mathbf{y}_{<t}, \mathbf{y}\right)\right] = \mathrm{D}_{\text{KL}}\left(\pi_\theta\left(\cdot \mid \mathbf{x}, \mathbf{y}_{<t}\right) \parallel \pi_{\theta_{\text{ref}}}\left(\cdot \mid \mathbf{x}, \mathbf{y}_{<t}\right)\right).$$

Even if we view it at a squence level, notice that

$$\sum_{t=1}^{T} \mathrm{KL}_t\left(\mathbf{x}, \mathbf{y}_{<t}, \mathbf{y}\right) = \sum_{t=1}^{T} \log\left(\frac{\pi_\theta\left(\mathbf{a} \mid \mathbf{x}, \mathbf{y}_{<t}\right)}{\pi_{\theta_{\text{ref}}}\left(\mathbf{a} \mid \mathbf{x}, \mathbf{y}_{<t}\right)}\right) \overset{(a)}{=} \log\left(\frac{\pi_\theta\left(\mathbf{y} \mid \mathbf{x}\right)}{\pi_{\theta_{\text{ref}}}\left(\mathbf{y} \mid \mathbf{x}\right)}\right),$$

where (a) is due to the chain rule. This still is an estimate in the sense that

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

In practice, during rollouts we only compute the log-prob ratio for the tokens we sampled, because that gives us a Monte Carlo estimate of the KL penalty. We don't compute the full expectation KL (that would require summing over the whole vocab).

[20]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.

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

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

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: Let us consider the k-th iteration for PPO and see what exactly we are going to do following the standard reinforcement learning road map. We denote the policy parameters at the start of iteration as θ_k and the critic parameter as ψ_k.

1. **Rollout**: We freeze policy $\theta_{\text{old}} = \theta_k$, and we sample a minibatch of prompts $\left\{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(B)}\right\} \sim \mathcal{D}_{\text{RL}}$, for each prompt $\mathbf{x}^{(j)}$ we generate a completion $\mathbf{y}^{(j)} = \left\{\mathbf{y}_1^{(j)}, \ldots, \mathbf{y}_{T_j}^{(j)}\right\}$ with θ_{old} (with potential temperature sampling and others). For each token t in each sequence, we form state $\mathbf{s}_t^{(b)} = \left(\mathbf{x}^{(b)}, \mathbf{y}_{<t}^{(b)}\right)$, and we let action $\mathbf{a}_t^{(b)} = \mathbf{y}_t^{(b)}$. We record the old log probability under $\pi_{\theta_{\text{old}}}$,

$$\log \pi_{\theta_{\text{old}}}\left(\mathbf{a}_t^{(b)} \mid \mathbf{s}_t^{(b)}\right),$$

and the log probability under the reference policy

$$\log \pi_{\theta_{\text{ref}}}\left(\mathbf{a}_t^{(b)} \mid \mathbf{s}_t^{(b)}\right),$$

we also evaluate the critic value $V_{\psi_k}(\mathbf{s}_t^{(b)})$ at roll-out time.

2. **Learning Signals (Inner):** We may take a minibatch $\mathcal{M}$ (token-step minibatch) of token steps $\{b, t\}$ (optionally with a padding mask $m_t^{(b)} \in \{0, 1\}$ if sequences have different lengths.) , we initialize $\theta = \theta_{k,0} = \theta_k$ and $\psi = \psi_{k,0} = \psi_k$ where we plan to run E steps of update (via SGD). Since we have the current inner iteration policy as θ, we can compute the log probability,

$$\pi_\theta \left(\mathbf{a}_t^{(b)} \mid \mathbf{s}_t^{(b)} \right),$$

which allows us to compute token level KL to the reference

$$\mathrm{KL}_t^{(b)} = \log \pi_\theta \left(\mathbf{a}_t^{(b)} \mid \mathbf{s}_t^{(b)} \right) - \log \pi_{\theta_{\text{ref}}} \left(\mathbf{a}_t^{(b)} \mid \mathbf{s}_t^{(b)} \right),$$

and further the KL-shaping reward:

$$r_t^{\mathrm{KL},(b)} = -\beta \cdot \mathrm{KL}_t^{(b)}.$$

Then we get the per token reward

$$r_t^{(b)}(\theta) \;=\; \begin{cases} r_t^{\mathrm{KL},(b)}(\theta), & t < T_b, \\ r_t^{\mathrm{KL},(b)}(\theta) + R_\phi\Big(\mathbf{x}^{(b)}, \mathbf{y}^{(b)}\Big), & t = T_b. \end{cases}$$

We further obtain the discounted return as

$$G_t^{(b)}(\theta) = \sum_{l=0}^{T_b - t} \gamma^l r_{t+l}^{(b)}(\theta),$$

and the TD residual

$$\delta_t^{(b)} = r_t^{(b)}(\theta) + \gamma(1 - \mathrm{done}_t^{(b)}) V_\psi(\mathbf{s}_{t+1}^{(b)}) - V_\psi(\mathbf{s}_t^{(b)}),$$

as well as the GAE through backward recursion

$$\hat{A}_t^{(b)}(\theta) = \delta_t^{(b)}(\theta) + \gamma\lambda(1 - \mathrm{done}_t^{(b)}) \hat{A}_{t+1}^{(b)}(\theta), \qquad \hat{A}_{T_b}^{(b)}(\theta) = 0.$$

Then we construct the value target

$$\hat{G}_t^{(b)}(\theta) = \hat{A}_t^{(b)}(\theta) + V_\psi(\mathbf{s}_t^{(b)}).$$

3. **PPO losses (Inner):** We first compute the importance ratio

$$o_t^{(b)} = \exp\left(\log \pi_\theta \left(\mathbf{a}_t^{(b)} \mid \mathbf{s}_t^{(b)} \right) - \log \pi_{\theta_{\text{old}}} \left(\mathbf{a}_t^{(b)} \mid \mathbf{s}_t^{(b)} \right) \right),$$

and we compute the actor loss

$$\mathcal{L}^{\text{actor}}(\theta) = \frac{1}{|\mathcal{M}|} \sum_{(b,t) \in \mathcal{M}} \min\Big(o_t^{(b)}(\theta)\, \hat{A}_t^{(b)}(\theta),\, \mathrm{Clip}\big(o_t^{(b)}(\theta), 1 - \epsilon, 1 + \epsilon\big)\, \hat{A}_t^{(b)}(\theta) \Big),$$

and critic loss

$$\mathcal{L}^{\text{critic}}(\psi) = \frac{1}{|\mathcal{M}|} \sum_{(b,t) \in \mathcal{M}} \big(V_\psi(\mathbf{s}_t^{(b)}) - \hat{G}_t^{(b)}(\theta) \big)^2,$$

as well as entropy

$$\mathcal{L}^{\text{entropy}}(\theta) = \frac{1}{|\mathcal{M}|} \sum_{(b,t)\in\mathcal{M}} \left[-\sum_{\mathbf{a}} \pi_\theta(\mathbf{a} \mid \mathbf{s}_t^{(b)}) \cdot \log \pi_\theta(\mathbf{a} \mid \mathbf{s}_t^{(b)}) \right].$$

To this end, we may form the PPO loss

$$L^{\text{PPO}}(\theta, \psi) = L^{\text{actor}}(\theta) - c_v \cdot L^{\text{critic}}(\psi) + c_e \cdot L^{\text{entropy}}(\theta),$$

where we can do gradient descent on $-\mathcal{L}^{\text{PPO}}$. Note that we can also do an optional mask for padded tokens, if sequence are padded, we just multiplly each summand by a binary mask $m_t^{(b)}$ and divide by $\sum_{(b,t)\in\mathcal{M}} m_b^{(t)}$ instead of $|\mathcal{M}|$ (Token-step minibatch convention does not need it.).

10.9 The Bigger Picture in Reinforcement Learning (PPO)

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[21] 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

$$J(\theta) := \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T} \gamma^t r(s_t, a_t)\right]. \tag{18}$$

That is maximize expected total reward. Note that the expectation actually means that we are sampling episodes (trajectories) 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:

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

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

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 is the return (sum of rewards) 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

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

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 (18), 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 (18), 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

$$\nabla_\theta J(\theta) = \mathbb{E}_\tau \left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta \left(\mathbf{a}_t \mid \mathbf{s}_t \right) \cdot G_t \right]. \tag{19}$$

The vanilla Policy Gradient method is based on this and Monte Carlo, which is of high variance, no constraints on updates. 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:

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

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 A^{π_θ} 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 basic 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[22]. 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 $(\mathbf{s}_t, \mathbf{a}_t, r_t)$[23], while we are collecting we also store the log probability 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.

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.[24] 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

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

where done_t is a termination flag, $\text{done}_t = 1$ means the episode has terminated at step t either because the agent reached a terminal state or a time limit was reached; $\text{done}_t = 0$ means the episode continues. When we reach the first termination iteration T, $1 - \text{done}_T = 0$, which means that $\delta_T = r_T - V_\psi(s_T)$[25]. One may wonder why we are setting the done flag, the reason is precisely that we are handling batched data, so episodes have variable length. As a result, we need to pad them so that the batch is aligned, and the done flag lets us distinguish real continuation from padding or termination. In this way, advantage/return computation stops at the right place, even inside a large batch.

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

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

MC returns are unbiased but high variance. Using a learned critic baseline reduces variance but can introduce bias if the critic is imperfect.

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

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

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

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

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

[25] In RL, we use bootstrapping to denote estimating a return by combining the observed reward with the critic's estimate of the future, (instead of waiting until the very end of the episode to know the true return). Here $\gamma(1 - \text{done}_t)V_\psi(\mathbf{s}_{t+1})$ is the bootstrap part.

where $\gamma = 0.99$ and $\lambda = 0.95$ are common. Notice that γ is the discounting factor and λ controls how far we can look ahead. To compute is efficiently, we use backward recursion,

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

In practice, due to the padding used in batch, we set

$$\hat{A}_t = \delta_t + \gamma\lambda(1 - \text{done}_t)\hat{A}_{t+1}, \qquad \hat{A}_T = 0 \tag{20}$$

where we use a termination flag to control bootstrapping, T is the index one past the last real step in the trajectory. From the estimate $\hat{A}_t$, we can reconstruct

$$\hat{G}_t = \hat{A}_t + V_\psi(\mathbf{s}_t), \tag{21}$$

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

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

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

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

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

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

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

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

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

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

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. In practice, at iteration k we first collect trajectories using the current policy π_{θ_k}. The most direct way to update would be to compute the gradient of the policy gradient objective at $\theta = \theta_k$, which is exactly on-policy with respect to the data. However, modern methods like PPO typically perform many minibatch gradient updates on the same rollout data. After the first update, the parameters have moved from θ_k to some $\theta_{k,t}$ (where t indexes local iterations on this minibatch), while the data is still drawn from π_{θ_k}. This creates a distribution mismatch: the true policy gradient is an expectation under the new policy $\pi_{\theta_{k,t}}$, but we only have samples from the old one. To address this, PPO introduces an importance-sampling correction using the ratio function

$$o_t(\theta; \theta_k) \;=\; \frac{\pi_\theta(\mathbf{a}_t \mid \mathbf{s}_t)}{\pi_{\theta_k}(\mathbf{a}_t \mid \mathbf{s}_t)}.$$

The surrogate objective is then written as

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

Differentiating gives

$$\begin{aligned}
\nabla_\theta \widehat{J}^{\text{PG}}(\theta; \theta_k) &= \mathbb{E}_{t, \mathbf{s}_t, \mathbf{a}_t} \left[\nabla_\theta o_t(\theta; \theta_k)\, \hat{A}^t \right] \\
&= \mathbb{E}_{t, \mathbf{s}_t, \mathbf{a}_t} \left[\frac{\pi_\theta(\mathbf{a}_t \mid \mathbf{s}_t)}{\pi_{\theta_k}(\mathbf{a}_t \mid \mathbf{s}_t)} \nabla_\theta \log \pi_\theta(\mathbf{a}_t \mid \mathbf{s}_t)\, \hat{A}^t \right] \\
&= \mathbb{E}_{t, \mathbf{s}_t, \mathbf{a}_t} \left[o_t(\theta; \theta_k)\, \nabla_\theta \log \pi_\theta(\mathbf{a}_t \mid \mathbf{s}_t)\, \hat{A}^t \right].
\end{aligned}$$

Thus the final expression is exactly the *importance-sampled version of the original policy gradient*, allowing us to reuse old-policy data π_{θ_k} while updating toward θ_{k+1}. This is easy to understand, we just first divide it by the probability it appears in the old disrtribution and then assigned the correct weight (the current probability) to it. For a formal proof, we only need to use the change of measure trick in (22).

Clipping. If we maximize this directly, the policy can change too aggressively, leading to instability. As a result, **PPO** in turn introduces a clipped surrogate,

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

where the clipping operator does the following:

1. When $o_t \in [1-\varepsilon, 1+\varepsilon]$, we do nothing.
2. When o_t goes beyond, we replace o_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.

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

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.

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

where

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

Practical hint. In practice, the training process goes very similarly like local training where there is a nested loop:

In the k-th iteration, we do

1. **Rollout Phase**: We fix $\theta_{\text{old}} = \theta_k$ where θ_{old} denotes the frozen snapshot of the current policy.
   - We use $\pi_{\theta_{\text{old}}}$ to interact with environment.
   - We collect a batch of trajectories: states $\mathbf{s}_t$, actions $\mathbf{a}_t$ and reward r_t, we then compute the log probabilities $\log \pi_{\theta_{\text{old}}}\left(\mathbf{a}_t \mid \mathbf{s}_t\right)$, and the old values $V_{\psi_k}(\mathbf{s}_t)$, and the done flags.

   Imagine that there are T steps across N parallel environments, eventually, we got

   $$\mathcal{B} = \left\{(\mathbf{s}_t, \mathbf{a}_t, r_t, \log \pi_{\text{old}(\mathbf{a}_t|\mathbf{s}_t)}, V_{\psi_k}(\mathbf{s}_t), \text{done}_t)_i\right\}_{t\in[T], i\in[N]}$$

2. **Advantage + Target Computation:** We compute learning signals:

   $$\delta_t = r_t + \gamma\left(1 - \text{done}_t\right) V_{\psi_k}(\mathbf{s}_{t+1}) - V_{\psi_k}(\mathbf{s}_t),$$

   $$\hat{A}_t = \delta_t + \gamma\lambda\left(1 - \text{done}_t\right)\hat{A}_{t+1}, \qquad \hat{A}_T = 0,$$

   where T is the index one past the last real step in the trajectory, and the value target

   $$\hat{G}_t = \hat{A}_t + V_{\psi_k}(\mathbf{s}_t).$$

3. **Update Actor & Critic:** We initialize trainable parameter $\theta = \theta_{k,0} = \theta_{\text{old}}$ and $\psi = \psi_{k,0} = \psi_k$, we loop over minibatches $\mathcal{M}$ of $\mathcal{B}$ for several epochs E. There are actually two types of minibatching, one is **episode level minibatching** and we are dividing based on episodes, but this can waste parallelism if episodes lengths vary a lot. Fortunately, we can also choose to use the **token-step minibatching**, constructing $\mathcal{M}$ using token steps from different episodes. We use the later minibatching, and in this case $|\mathcal{M}|$ is just the total pairs of token-steps.

   (I) Policy distribution under current θ,

   $$\pi_\theta\left(\cdot \mid \mathbf{s}_t\right), \quad \log \pi_\theta\left(\mathbf{a}_t \mid \mathbf{s}_t\right).$$

   (II) Importance ratio,

   $$o_t(\theta; \theta_{\text{old}}) = \exp\left(\log \pi_\theta\left(\mathbf{a}_t \mid \mathbf{s}_t\right) - \log \pi_{\theta_{\text{old}}}\left(\mathbf{a}_t \mid \mathbf{s}_t\right)\right)$$

(III) Actor loss (clipped surrogate)

$$\begin{aligned} L^{\text{actor}} &= \mathbb{E}_t \left[\min \left\{ o_t(\theta;\theta_k)\hat{A}^t, \text{clip}\left(o_t(\theta;\theta_k), 1-\varepsilon, 1+\varepsilon\right) \cdot \hat{A}^t \right\} \right] \\ &= \frac{1}{|\mathcal{M}|} \sum_{(\mathbf{s}_t,\mathbf{a}_t)\in\mathcal{M}} \min \left\{ o_t(\theta;\theta_k)\hat{A}^t, \text{clip}\left(o_t(\theta;\theta_k), 1-\varepsilon, 1+\varepsilon\right) \cdot \hat{A}^t \right\}. \end{aligned}$$

Notice that we already do summation over time since $|\mathcal{M}|$ contains it.

(IV) Critic loss

$$\begin{aligned} L^{\text{critic}} &= \mathbb{E}_t \left[\left(V_\psi(\mathbf{s}_t) - \hat{G}_t \right)^2 \right] \\ &= \frac{1}{|\mathcal{M}|} \sum_{\mathbf{s}_t\in\mathcal{M}} \left(V_\psi(\mathbf{s}_t) - \hat{G}_t \right)^2. \end{aligned}$$

(V) Entropy bonus (exploration):

$$\begin{aligned} L^{\text{entropy}} &= \mathbb{E}_t \left[-\sum_{\mathbf{a}} \pi_\theta(\mathbf{a} \mid \mathbf{s}_t) \cdot \log \pi_\theta(\mathbf{a} \mid \mathbf{s}_t) \right] \\ &= \frac{1}{|\mathcal{M}|} \sum_{\mathbf{s}_t\in\mathcal{M}} \left(-\sum_{\mathbf{a}} \pi_\theta(\mathbf{a} \mid \mathbf{s}_t) \cdot \log \pi_\theta(\mathbf{a} \mid \mathbf{s}_t) \right). \end{aligned}$$

(VI) Total PPO objective

$$L^{\text{PPO}} = -L^{\text{actor}} + c_v \cdot L^{\text{critic}} - c_e \cdot L^{\text{entropy}}.$$

(VII) Take a gradient step, so that we obtain $\theta_{k,e+1}, \psi_{k,e+1}$ from $\theta_{k,e}$ and $\psi_{k,e}$.

4. We let $\theta_{k+1} = \theta_{k,E}$ and $\psi_{k+1} = \psi_{k,E}$ using gradient ascent (or gradient descent to minus PPO loss), and prepare to enter the next iteration.

Regarding the meaning of concepts

- Reward r_t: decided by the environment, independent of the policy, indicates how much can one get at moment t. Normally, it depends on $\mathbf{a}_t$ and $\mathbf{s}_t$, i.e., $r_t = r_t(\mathbf{a}_t, \mathbf{s}_t)$. The policy do affect the realized policy since it decides what $\mathbf{a}_t$ and $\mathbf{s}_t$ could be.
- Return G_t: discounted cumulated future rewards, which is the total one to get from now on.
- Discounted factor γ, which is a number in $[0, 1]$ controlling how much future rewards matter. $\gamma \approx 1$ means that we think in a long term fashion, and γ close to 0 suggests to live in the moment.
- Value function $V^\pi(\mathbf{s}) = \mathbb{E}_\pi [G_t \mid \mathbf{s}_t = \mathbf{s}]$, which is the expected return in state $\mathbf{s}$ following policy π.
- Action-value function $Q^\pi(\mathbf{s}, \mathbf{a}) = \mathbb{E}_\pi [G_t \mid \mathbf{s}_t = \mathbf{s}, \mathbf{a}_t = \mathbf{a}]$, which is the expected return starting in state $\mathbf{s}$, taking action $\mathbf{a}$.
- Advantage $A^\pi(\mathbf{s}, \mathbf{a}) = Q^\pi(\mathbf{s}, \mathbf{a}) - V^\pi(\mathbf{s})$, which has the meaning that is this move better than my usual move? This term is the key for variance reduction.

- Temporal difference $\delta_t = r_t + \gamma V(\mathbf{s}_{t+1}) - V(\mathbf{s}_t)$, where V is the baseline prediction of cumulative reward. δ_t is the prediction error. The true value satisfies

$$V^\pi(\mathbf{s}_t) = \mathbb{E}_\pi \left[r_t + \gamma V^\pi(\mathbf{s}_{t+1}) \mid \mathbf{s}_t \right],$$

  which is the Bellman equation.

- Optimal policy $\pi^\star$, policy that maximizes expected return from every state.

10.10 PPO-KL

If we are using raw PPO as an alignment method, we are just optimizing against the reward model R_ϕ, which may render the policy dropping into unsafe, verbose, or unnatural completions. The solution to this is to add a KL-penalty that discourages moving too far from the reference policy (usually the SFT model). Essentially, PPO-KL just adds an extra KL penalty term

$$J^{\text{PPO-KL}}(\theta) = J^{\text{PPO}}(\theta) - \beta \cdot \mathbb{E}_t \left[\mathrm{D}_{\text{KL}} \left(\pi_\theta \left(\cdot \mid \mathbf{s}_t \right) \parallel \pi_{\theta_{\text{ref}}} \left(\cdot \mid \mathbf{s}_t \right) \right) \right].$$

Here we follow the convention and use J as the objective to maximize while $\mathcal{L}$ is reserved for objective to minimize.

10.11 DPO (Direct Preference Optimization)

Instead of using PPO with a reward model, DPO directly optimizes the model's policy to match human preferences using a pairwise comparison dataset. We no longer need to train or sample from a reward model at each step and we do not need complex RL machinery (advantage estimation, rollout, etc.). Training reduces to a standard supervised loss on pairs of responses.

We will train (finetune) on 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 (just like RLHF PPO stage 2). The core idea is that we assume that there is an underlying reward function $r(\mathbf{x}, \mathbf{y})$ such that

$$\mathbb{P}\left(\mathbf{y}^+ \succ \mathbf{y}^- \mid x \right) = \sigma\left(r(\mathbf{x}, \mathbf{y}^+) - r(\mathbf{x}, \mathbf{y}^-) \right), \qquad \text{(Bradley-Terry)}$$

where $\sigma(z) = \frac{1}{1+e^{-z}}$ is the sigmoid function. DPO avoids learning an explicit reward model by connecting rewards to log-probabilities under the learned policy vs. a reference model (typically the SFT model):

$$r(\mathbf{x}, \mathbf{y}) \propto \beta \left(\log \pi_\theta \left(\mathbf{y} \mid \mathbf{x} \right) - \log \pi_{\theta_{\text{ref}}} \left(\mathbf{y} \mid \mathbf{x} \right) \right),$$

where $\beta > 0$ is the temperature that controls the regularization strength.

If we plug in this into the Bradley-Terry model, we get

$$\mathbb{P}\left(\mathbf{y}^+ \succ \mathbf{y}^- \mid x \right) = \sigma\left(\beta \left(\log \frac{\pi_\theta \left(\mathbf{y}^+ \mid \mathbf{x} \right)}{\pi_{\theta_{\text{ref}}} \left(\mathbf{y}^+ \mid \mathbf{x} \right)} - \log \frac{\pi_\theta \left(\mathbf{y}^- \mid \mathbf{x} \right)}{\pi_{\theta_{\text{ref}}} \left(\mathbf{y}^- \mid \mathbf{x} \right)} \right) \right).$$

Thus we choose to maximize the likelyhood of observed preference,

$$J^{\text{DPO}}(\theta) = \mathbb{E}_{(x, y^+, y^-) \sim \mathcal{D}_{\text{pref}}} \left[\log \sigma \left(\frac{1}{\beta} \Big(\underbrace{\log \frac{\pi_\theta(y^+|x)}{\pi_{\text{ref}}(y^+|x)}}_{\text{advantage of } y^+} - \underbrace{\log \frac{\pi_\theta(y^-|x)}{\pi_{\text{ref}}(y^-|x)}}_{\text{advantage of } y^-} \Big) \right) \right],$$

which is just a binary logistic regression (here we maximize this likelihood) objective over pairs. The entire DPO algorithm can then be described as

1. We start with a supervised model both as π_θ and π_{ref}.
2. We take a minibatch of paris $\left\{\mathbf{x}_b, \mathbf{y}_b^+, \mathbf{y}_b^-\right\}_{b=1}^B$, and we compute the log probabilities from π_θ and π_{ref}.
3. We then compute the loss $\mathcal{L}^{\text{DPO}}$ and backpropagate and update θ.

Notice that the $\pi_{\theta_{\text{ref}}}(\cdot \mid \mathbf{x})$ term is there to ensure that the model doesn't drift too far from the SFT model (similar to the KL penalty in PPO-RLHF). Its advantages include

1. It directly aligns the policy with preference data.
2. Simple, stable, and efficient: training is just supervised cross-entropy with a custom loss.

Why do we assume r follows the specific form? Consider the per prompt optimization with reward and KL-regularization,

$$\max_{\pi_\theta} \left\{ \mathbb{E}_{\mathbf{y}\sim\pi_\theta(\cdot|\mathbf{x})} \left[r(\mathbf{x}, \mathbf{y}) \right] - \beta \cdot \mathrm{D}_{\text{KL}} \left(\pi_\theta(\cdot \mid \mathbf{x}) \parallel \pi_{\theta_{\text{ref}}}(\cdot \mid \mathbf{x}) \right) \right\}.$$

The objective itself is strictly concave in $\pi_\theta(\cdot \mid \mathbf{x})$ which is a probability distribution (a vector of probabilities) over all possible $\mathbf{y}$. Clearly for the first term, it is linear in $\pi_\theta(\cdot \mid y)$, while for the second term, we have

$$-\beta \cdot \sum_{\mathbf{y}} \pi_\theta(\mathbf{y} \mid \mathbf{x}) \log \frac{\pi_\theta(\mathbf{y} \mid \mathbf{x})}{\pi_{\theta_{\text{ref}}}(\mathbf{y} \mid \mathbf{x})},$$

which is also strictly concave in $\pi_\theta(\cdot \mid \mathbf{x})$ (KL-divergence itself is convex in $\pi_\theta(\cdot \mid \mathbf{x})$). If we write the write the Lagrangian with normalization muiltipler $\lambda(\mathbf{x})$:

$$\mathcal{L}(\pi_\theta, \lambda) = \sum_{\mathbf{y}} \pi_\theta(\mathbf{y} \mid \mathbf{x}) \cdot r(\mathbf{x}, \mathbf{y}) - \beta \sum_{\mathbf{y}} \pi_\theta(\mathbf{y} \mid \mathbf{x}) \cdot \log \frac{\pi_\theta(\mathbf{y} \mid \mathbf{x})}{\pi_{\theta_{\text{ref}}}(\mathbf{y} \mid \mathbf{x})} + \lambda(\mathbf{x}) \left(\sum_{\mathbf{y}} \pi_\theta(\mathbf{y} \mid \mathbf{x}) - 1 \right),$$

whose optimal condition implies

$$r(\mathbf{x}, \mathbf{y}) - \beta \left(\log \pi_\theta(\mathbf{y} \mid \mathbf{x}) - \pi_{\theta_{\text{ref}}}(\mathbf{y} \mid \mathbf{x}) + 1 \right) + \lambda(\mathbf{x}) = 0.$$

Rearrange we obtain:

$$r(\mathbf{x}, \mathbf{y}) = \beta \left(\log \pi^\star(\mathbf{y} \mid \mathbf{x}) - \log \pi_{\theta_{\text{ref}}}(\mathbf{y} \mid \mathbf{x}) \right) + C(\mathbf{x}).$$

where $C(\mathbf{x}) = \lambda(\mathbf{x}) + \beta$. Since in the pariwise preference model the bias $C(\mathbf{x})$ cancels out, this justifies our choice. In a word, the DPO choice of r comes from the PPO + KL setup, but with simplifications such as implicit reward model and sequence level loss instead of per token.

10.12 GPRO (Generalized Preference Optimization)

GPRO generalizes DPO further, in the sense that it instead of relying only on binary preferences (preferred vs dispreferred), it allows training with arbitrary reward functions (scalar signals, multiple preferences, or mixtures).

Consider the dataset $\mathcal{D}_{\text{pref}} = \{(\mathbf{x}_j, \mathbf{y}_j, r(\mathbf{x}_j, \mathbf{y}_j))\}_{j=1}^M$ (Different x_j could be the same.). We now define the log-ratio between policy and preference

$$z_\theta(\mathbf{x}, \mathbf{y}) = \log \frac{\pi_\theta(\mathbf{y} \mid \mathbf{x})}{\pi_{\theta_{\text{ref}}}(\mathbf{y} \mid \mathbf{x})}.$$

The GPRO loss is given by

$$J^{\text{GPRO}}(\theta) = \mathbb{E}_{(\mathbf{x},\mathbf{y})\sim\mathcal{D}_{\text{pref}}}\left[\Omega^\star\left(\beta z_\theta(\mathbf{x},\mathbf{y})\right) - \beta r(\mathbf{x},\mathbf{y}) z_\theta(\mathbf{x},\mathbf{y})\right],$$

where $\Omega^\star$ is the convex conjugate of a link function Ω,$\beta > 0$. The training follows maximizing this dual objective. In practice we use an equivalent form:

$$\widehat{L}(\theta) = \sum_{x\in\mathcal{D}_{\text{pref}}} \mathcal{L}_{\mathbf{x}}(\theta) = -\beta \sum_{\mathbf{x}} \sum_{i=1}^{K_{\mathbf{x}}} \omega_i(\mathbf{x}) \log \pi_\theta\left(\mathbf{y}_i \mid \mathbf{x}\right), \qquad \omega_i(\mathbf{x}) = \frac{e^{r_i(\mathbf{y}_i,\mathbf{x})/\beta}}{\sum_{j=1}^{K_{\mathbf{x}}} e^{r_i(\mathbf{y}_j,\mathbf{x})/\beta}}$$

The intuition: Fix a prompt $\mathbf{x}$, consider the standard KL-regularized objective

$$\max_{\pi} \left\{\mathbb{E}_{\mathbf{y}\sim\pi(\cdot|\mathbf{x})}\left[r(\mathbf{x},\mathbf{y})\right] - \beta \cdot \mathrm{D}_{\text{KL}}\left(\pi\left(\cdot \mid \mathbf{x}\right) \parallel \pi_{\theta_{\text{ref}}}\left(\cdot \mid \mathbf{x}\right)\right)\right\}.$$

We will reparameterize the policy via a density ratio. Let

$$z(\mathbf{y}) = \log \frac{\pi\left(\mathbf{y} \mid \mathbf{x}\right)}{\pi_{\theta_{\text{ref}}}\left(\mathbf{y} \mid \mathbf{x}\right)}, \qquad u(\mathbf{y}) = \exp\left(z(\mathbf{y})\right) = \frac{\pi\left(\mathbf{y} \mid \mathbf{x}\right)}{\pi_{\theta_{\text{ref}}}\left(\mathbf{y} \mid \mathbf{x}\right)}.$$

Then $\pi\left(\mathbf{y} \mid \mathbf{x}\right) = \pi_{\theta_{\text{ref}}}\left(\mathbf{y} \mid \mathbf{x}\right) u(\mathbf{y})$ must normalize, i.e.,

$$\sum_{\mathbf{y}} \pi_{\theta_{\text{ref}}}\left(\mathbf{y} \mid \mathbf{x}\right) u(\mathbf{y}) = 1 \Leftrightarrow \mathbb{E}_{\mathbf{y}\sim\pi_{\theta_{\text{ref}}}(\cdot|\mathbf{x})}\left[u(\mathbf{y})\right] = 1.$$

Moreover,

$$\mathrm{D}_{\text{KL}}\left(\pi\left(\cdot \mid \mathbf{x}\right) \parallel \pi_{\theta_{\text{ref}}}\left(\cdot \mid \mathbf{x}\right)\right) = \mathbb{E}_{\mathbf{y}\sim\pi(\cdot|\mathbf{x})}\left[z(\mathbf{y})\right]$$

and

$$\mathbb{E}_{\mathbf{y}\sim\pi(\cdot|\mathbf{x})}\left[f(\mathbf{y})\right] = \sum_{\mathbf{y}} \pi_{\theta_{\text{ref}}}\left(\mathbf{y} \mid \mathbf{x}\right) u(\mathbf{y}) f(\mathbf{y}) = \mathbb{E}_{\mathbf{y}\sim\pi_{\theta_{\text{ref}}}(\cdot|\mathbf{x})}\left[u(\mathbf{y}) f(\mathbf{y})\right]. \tag{22}$$

As a result, taking $f(\mathbf{y}) = r(\mathbf{x},\mathbf{y})$, we can reform the original objective as

$$\max_{u\geq 0, \mathbb{E}_{\pi_{\theta_{\text{ref}}}(\cdot|\mathbf{x})}[u(\mathbf{y})]=1} \left\{\mathbb{E}_{\mathbf{y}\sim\pi_{\theta_{\text{ref}}}(\cdot|\mathbf{x})}\left[u(\mathbf{y}) r(\mathbf{x},\mathbf{y})\right] - \beta \mathbb{E}_{\mathbf{y}\sim\pi_{\theta_{\text{ref}}}(\cdot|\mathbf{x})}\left[u(\mathbf{y}) z(\mathbf{y})\right]\right\}, \qquad z(\mathbf{y}) = \log u(\mathbf{y}).$$

The same conclusion also holds in the continuous case. We form the Lagrangian for the fixed $\mathbf{x}$,

$$\begin{aligned}\mathcal{L}\left(u,\lambda\right) &= \mathbb{E}_{\mathbf{y}\sim\pi_{\theta_{\text{ref}}}(\cdot|\mathbf{x})}\left[ur\right] - \beta \mathbb{E}_{\mathbf{y}\sim\pi_{\theta_{\text{ref}}}(\cdot|\mathbf{x})}\left[uz\right] - \lambda\left(\mathbb{E}_{\mathbf{y}\sim\pi_{\theta_{\text{ref}}}(\cdot|\mathbf{x})}\left[u\right] - 1\right) \\ &= \mathbb{E}_{\mathbf{y}\sim\pi_{\theta_{\text{ref}}}(\cdot|\mathbf{x})}\left[-\left(\beta uz - ur + \lambda u\right)\right] + \lambda.\end{aligned}$$

Differentiate pointwisely w.r.t. u, we get

$$r - \beta(\log u + 1) - \lambda = 0,$$

which suggests

$$\log u(\mathbf{y}) = \frac{r(\mathbf{x},\mathbf{y}) - \lambda - \beta}{\beta} \Rightarrow u^\star(\mathbf{y}) = C \cdot \exp\left(\frac{r(\mathbf{x},\mathbf{y})}{\beta}\right), \quad \text{where} \quad C := \exp\left(-\frac{\lambda+\beta}{\beta}\right).$$

Enforcing the constraint $\mathbb{E}_{\mathbf{y}\sim\pi_{\theta_{\text{ref}}}(\cdot|\mathbf{x})}[u] = 1$, we get

$$C \cdot \mathbb{E}_{\mathbf{y}\sim\pi_{\theta_{\text{ref}}}(\cdot|\mathbf{x})}\left[\exp\left(\frac{r}{\beta}\right)\right] = 1 \Rightarrow C = \frac{1}{\mathbb{E}_{\mathbf{y}\sim\pi_{\theta_{\text{ref}}}(\cdot|\mathbf{x})}\left[\exp\left(\frac{r}{\beta}\right)\right]}.$$

Therefore

$$u^\star(\mathbf{y}) = \frac{\exp\left(\frac{r(\mathbf{x},\mathbf{y})}{\beta}\right)}{\mathbb{E}_{\mathbf{y}\sim\pi_{\theta_{\text{ref}}}(\cdot|\mathbf{x})}\left[\exp\left(\frac{r}{\beta}\right)\right]} \quad \text{and} \quad \pi^\star(\mathbf{y} \mid \mathbf{x}) = \pi_{\theta_{\text{ref}}}(\mathbf{y} \mid \mathbf{x})\, u^\star(y) = \frac{\pi_{\theta_{\text{ref}}}(\mathbf{y} \mid \mathbf{x}) \exp\left(\frac{r(\mathbf{x},\mathbf{y})}{\beta}\right)}{\mathbb{E}_{\mathbf{y}\sim\pi_{\theta_{\text{ref}}}(\cdot|\mathbf{x})}\left[\exp\left(\frac{r(\mathbf{x},\mathbf{y})}{\beta}\right)\right]}.$$

The optimal policy is obtained by reweighting the reference distribution, so that outcomes with higher reward get exponentially more weight, this reweighting is what people called **exponential tilt**. The optimal log ratio is given by

$$z^\star(\mathbf{y}) = \frac{r(\mathbf{x},\mathbf{y})}{\beta} - \log \mathbb{E}_{\mathbf{y}\sim\pi_{\theta_{\text{ref}}}(\cdot|\mathbf{x})}\left[\exp\left(\frac{r(\mathbf{x},\mathbf{y})}{\beta}\right)\right].$$

Since the elementwise objective is strictly concave for $u > 0$ and the constraint is affine, KKT conditions apply; complementary slackness gives an interior solution $u^\star(\mathbf{y}) > 0$, which makes the stationary point above a unique maximizer. Plug it back into the primal gives the optimal value

$$\beta \log \mathbb{E}_{\mathbf{y}\sim\pi_{\theta_{\text{ref}}}(\cdot|\mathbf{x})}\left[\exp\left(\frac{r}{\beta}\right)\right],$$

which is of the classic Donsker-Varadhan form, consistent with the exponential-tilt solution. In principle, if we have explicit $r(\mathbf{x},\mathbf{y})$ for all candidate responses $\mathbf{y}$ we could just set our policy equal to $\pi^\star$, but in practice, the reward is only given on the dataset, and even if the value is available, we need to compute

$$\mathbb{E}_{\mathbf{y}\sim\pi_{\theta_{\text{ref}}}(\cdot|\mathbf{x})}\left[\exp\left(\frac{r}{\beta}\right)\right],$$

which is log-sum-exp (log-partition function) all possible completions $\mathbf{y}$, which is not computable.

The insight of GPRO: Instead of trying to match π_θ to the global closed form $\pi^\star$, we can design a tractable convex surrogate that drives the model's log-ratio[26]

$$z_\theta(\mathbf{y}) = \log\left(\frac{\pi_\theta(\mathbf{y} \mid \mathbf{x})}{\pi_{\theta_{\text{ref}}}(\mathbf{y} \mid \mathbf{x})}\right),$$

to behave as if it were the solution of that optimization. Now introducing the convex function (entropy link)

$$\Omega(u) = \beta\left(u \log u - u + 1\right), \qquad \text{for} \qquad u > 0,$$

whose convex conjugate is

$$\Omega^\star(t) = \sup_{u\geq 0}\{ut - \Omega(u)\} = \beta\left(e^{t/\beta} - 1\right).$$

Now we use the Fenchel-Young inequality,

$$ut - \Omega(u) \leq \Omega^\star(t)$$

[26]The question is simply: how do we train a parameterized π_θ so that its log-ratio z_θ behaves like $z_\star$ without that partition function.

with equality achieved if and only if $u = e^{\frac{t}{\beta}}$, which means that for any $\mathbf{y}$, we have

$$u^\star(\mathbf{y}) \cdot \beta z_\theta(\mathbf{y}) - \Omega(u^\star(\mathbf{y})) \leq \Omega^\star(\beta z_\theta(\mathbf{y}))$$

Taking expectation under $\pi_{\theta_{\text{ref}}}(\cdot \mid \mathbf{x})$, we have

$$\mathbb{E}_{\mathbf{y}\sim\pi_{\theta_{\text{ref}}}(\cdot|\mathbf{x})}\left[u^\star(\mathbf{y}) \cdot \beta z_\theta(\mathbf{y})\right] - \mathbb{E}_{\mathbf{y}\sim\pi_{\theta_{\text{ref}}}(\cdot|\mathbf{x})}\left[\Omega(u^\star(\mathbf{y}))\right] \leq \mathbb{E}_{\mathbf{y}\sim\pi_{\theta_{\text{ref}}}(\cdot|\mathbf{x})}\left[\Omega^\star(\beta z_\theta(\mathbf{y}))\right]. \tag{23}$$

For the first term, using the change of measure in (22), we have

$$\mathbb{E}_{\mathbf{y}\sim\pi_{\theta_{\text{ref}}}(\cdot|\mathbf{x})}\left[u^\star(\mathbf{y}) \cdot \beta z_\theta(\mathbf{y})\right] = \beta \cdot \mathbb{E}_{\mathbf{y}\sim\pi^\star(\cdot|\mathbf{x})}\left[z_\theta(\mathbf{y})\right],$$

while for the second term, we have

$$\begin{aligned}
\mathbb{E}_{\mathbf{y}\sim\pi_{\theta_{\text{ref}}}(\cdot|\mathbf{x})}\left[\Omega(u^\star(\mathbf{y}))\right] &= \beta\left(\mathbb{E}_{\mathbf{y}\sim\pi_{\theta_{\text{ref}}}(\cdot|\mathbf{x})}\left[u^\star(\mathbf{y})\log u^\star(\mathbf{y})\right] - \mathbb{E}_{\mathbf{y}\sim\pi_{\theta_{\text{ref}}}(\cdot|\mathbf{x})}\left[u^\star(\mathbf{y})\right] + 1\right) \\
&= \beta \cdot \mathbb{E}_{\pi^\star(\cdot|\mathbf{x})}\left[\log u^\star(\mathbf{y})\right] \\
&= \beta \cdot \mathbb{E}_{\pi^\star(\cdot|\mathbf{x})}\left[\frac{r(\mathbf{x},\mathbf{y})}{\beta} + C\right] \\
&= \beta \cdot \mathbb{E}_{\pi^\star(\cdot|\mathbf{x})}\left[\frac{r(\mathbf{x},\mathbf{y})}{\beta}\right] + \beta C.
\end{aligned}$$

Plus this into the original inequality (23), we obtain

$$\beta \cdot \mathbb{E}_{\mathbf{y}\sim\pi^\star(\cdot|\mathbf{x})}\left[z_\theta(\mathbf{y})\right] - \left(\beta \cdot \mathbb{E}_{\pi^\star(\cdot|\mathbf{x})}\left[\frac{r(\mathbf{x},\mathbf{y})}{\beta}\right] + \beta C\right) \leq \mathbb{E}_{\mathbf{y}\sim\pi_{\theta_{\text{ref}}}(\cdot|\mathbf{x})}\left[\Omega^\star(\beta z_\theta(\mathbf{y}))\right],$$

which rearranges to

$$\begin{aligned}
\beta \cdot \mathbb{E}_{\mathbf{y}\sim\pi^\star(\cdot|\mathbf{x})}\left[z_\theta(\mathbf{y})\right] - \mathbb{E}_{\pi^\star(\cdot|\mathbf{x})}\left[r(\mathbf{x},\mathbf{y})\right] &\leq \beta C + \mathbb{E}_{\mathbf{y}\sim\pi_{\theta_{\text{ref}}}(\cdot|\mathbf{x})}\left[\Omega^\star(\beta z_\theta(\mathbf{y}))\right], \\
&= \beta\mathbb{E}_{\mathbf{y}\sim\pi_{\theta_{\text{ref}}}(\cdot|\mathbf{x})}\left[e^{z_\theta(\mathbf{y})} - 1\right]
\end{aligned}$$

where the inequality is tight when $\log z_\theta(\mathbf{y}) = u^\star(\mathbf{y}) = \log z^\star(\mathbf{y})$. This is essentially an convex upper bound that is tight when $z_\theta(\mathbf{y}) = z^\star(\mathbf{y})$. Notice that we are trying to optimize towards z, so we can write the above inequality as

$$\mathbb{E}_{\mathbf{y}\sim\pi_{\theta_{\text{ref}}}(\cdot|\mathbf{x})}\left[\Omega^\star(\beta z_\theta(\mathbf{y}))\right] - \beta \cdot \mathbb{E}_{\mathbf{y}\sim\pi^\star(\cdot|\mathbf{x})}\left[z_\theta(\mathbf{y})\right] \geq \text{constant}.$$

We can define the following objective from Fenchel-Young,

$$\mathcal{S}_\mathbf{x}(z_\theta) := \mathbb{E}_{\mathbf{y}\sim\pi_{\theta_{\text{ref}}}(\cdot|\mathbf{x})}\left[\Omega^\star(\beta z_\theta(\mathbf{y}))\right] - \beta \cdot \mathbb{E}_{\mathbf{y}\sim\pi^\star(\cdot|\mathbf{x})}\left[z_\theta(\mathbf{y})\right] - \text{constant},$$

where $\mathcal{S}_\mathbf{x}(z_\theta) \geq 0$, $\mathcal{S}_\mathbf{x}(z_\theta) = 0$ if and only if $z = z^\star$ and clearly it is convex. Notice that this surrogates resembles the original GPRO loss. This surrogate serves as a way to drive z_θ to what we want without ever computing the log partition. Notice that for the first term, we have

$$\mathbb{E}_{\mathbf{y}\sim\pi_{\theta_{\text{ref}}}(\cdot|\mathbf{x})}\left[\Omega^\star(\beta z_\theta(\mathbf{y}))\right] = \beta \cdot \mathbb{E}_{\mathbf{y}\sim\pi_{\theta_{\text{ref}}}(\cdot|\mathbf{x})}\left[e^{z_\theta}\right] - \beta,$$

while

$$\mathbb{E}_{\mathbf{y}\sim\pi_{\theta_{\text{ref}}}(\cdot|\mathbf{x})}\left[e^{z_\theta}\right] = \sum_{y\sim\pi_\theta(\cdot|\mathbf{x})} \pi_{\theta_{\text{ref}}}(\mathbf{y} \mid \mathbf{x}) \cdot \frac{\pi_\theta(\mathbf{y} \mid \mathbf{x})}{\pi_{\theta_{\text{ref}}}(\mathbf{y} \mid \mathbf{x})} = 1.$$

Thus we can further write

$$\mathcal{S}_{\mathbf{x}}(z_\theta) = -\beta \cdot \mathbb{E}_{\mathbf{y}\sim\pi^\star(\cdot|\mathbf{x})}\left[z_\theta(\mathbf{y})\right],$$

leaving out the constant. Differentiate it w.r.t θ, and notice that

$$\nabla_\theta z_\theta(\mathbf{y}) = \nabla_\theta \log \pi_\theta\left(\mathbf{y} \mid \mathbf{x}\right).$$

Hence

$$\nabla_\theta \mathcal{S}_{\mathbf{x}}(z_\theta) = -\beta \cdot \mathbb{E}_{\mathbf{y}\sim\pi^\star(\cdot|\mathbf{x})}\left[\nabla_\theta \log \pi_\theta\left(\mathbf{y} \mid \mathbf{x}\right)\right],$$

which is the population gradient we want to estimate from data. Notice that we can not compute the term $\mathbb{E}_{\mathbf{y}\sim\pi^\star(\cdot|\mathbf{x})}\left[f(\mathbf{y})\right]$ directly because we do not know the distribution, however, we can estimate it, with rewards. We have

$$\begin{aligned}\mathbb{E}_{\mathbf{y}\sim\pi^\star(\cdot|\mathbf{x})}\left[f(\mathbf{y})\right] &= \sum_{\mathbf{y}\sim\pi^\star(\cdot|\mathbf{x})} \pi^\star(\mathbf{y} \mid \mathbf{x}) \cdot f(\mathbf{y}) \\ &= \frac{\pi_{\theta_{\text{ref}}}\left(\mathbf{y} \mid \mathbf{x}\right) \exp\left(\frac{r(\mathbf{x},\mathbf{y})}{\beta}\right) \cdot f(\mathbf{y})}{\mathbb{E}_{\mathbf{y}\sim\pi_{\theta_{\text{ref}}}(\cdot|\mathbf{x})}\left[\exp\left(\frac{r(\mathbf{x},\mathbf{y})}{\beta}\right)\right]} = \frac{\mathbb{E}_{\mathbf{y}\sim\pi_{\theta_{\text{ref}}}(\cdot|\mathbf{x})}\left[\exp\left(\frac{r(\mathbf{x},\mathbf{y})}{\beta}\right) \cdot f(\mathbf{y})\right]}{\mathbb{E}_{\mathbf{y}\sim\pi_{\theta_{\text{ref}}}(\cdot|\mathbf{x})}\left[\exp\left(\frac{r(\mathbf{x},\mathbf{y})}{\beta}\right)\right]}.\end{aligned}$$

Assume for prompt $\mathbf{x}$, we have K responses with rewards $\{(\mathbf{y}_i, r_i)\}_{i=1}^{K_{\mathbf{x}}}$, then we can estimate the expectation using[27]

$$\mathbb{E}_{\mathbf{y}\sim\pi^\star(\cdot|\mathbf{x})}\left[f(\mathbf{y})\right] \approx \frac{\frac{1}{K}\sum_{i=1}^{K_{\mathbf{x}}} e^{r_i/\beta} f(\mathbf{y}_i)}{\frac{1}{K_{\mathbf{x}}}\sum_{i=1}^{K_{\mathbf{x}}} e^{r_i/\beta}} = \sum_{i=1}^{K_{\mathbf{x}}} \omega_i(\mathbf{x}) f(\mathbf{y}_i), \qquad \omega_i(\mathbf{x}) := \frac{e^{r_i/\beta}}{\sum_{i=1}^{K_{\mathbf{x}}} e^{r_i/\beta}},$$

where the problematic As a result, the estimated gradient is given by

$$\nabla_\theta \mathcal{S}_{\mathbf{x}}(z_\theta) \approx -\beta \cdot \sum_{i=1}^{K_{\mathbf{x}}} \omega_i(\mathbf{x}) \nabla_\theta \log \pi_{\pi_\theta}\left(\mathbf{y}_i \mid \mathbf{x}\right),$$

which is a **weighted negative log-likelihood**. corresponding to

$$\widehat{\mathcal{L}}_{\mathbf{x}}(\theta) := -\beta \cdot \sum_{i=1}^{K_{\mathbf{x}}} \omega_i(\mathbf{x}) \log \pi_\theta\left(\mathbf{y}_i \mid \mathbf{x}\right),$$

if we further aggregate over prompts

$$\widehat{L}(\theta) = \sum_{x\in\mathcal{D}_{\text{pref}}} \mathcal{L}_{\mathbf{x}}(\theta) = -\beta \sum_{\mathbf{x}} \sum_{i=1}^{K_{\mathbf{x}}} \omega_i(\mathbf{x}) \log \pi_\theta\left(\mathbf{y}_i \mid \mathbf{x}\right),$$

which is clearly equivalent to the original GPRO dual loss, since it is our starting point (denoted as $\mathcal{S}_{\mathbf{x}}(z_\theta)$ here). Note that we are trying to minimize this loss as we are trying to minimizer $\mathcal{S}_{\mathbf{x}}$.

[27] This is the key, the expectation itself is easier to estimate than the distribution.

Relation to DPO DPO is related to GPRO, let a training item be a pair $(\mathbf{x}, \mathbf{y}^+, \mathbf{y}^-)$ with the label $\mathbf{y}^+ \succ \mathbf{y}^-$. From the GPRO analysis, we have

$$z^\star(\mathbf{x}, \mathbf{y}) = \frac{r(\mathbf{x}, \mathbf{y})}{\beta} - \log \mathbb{E}_{\mathbf{y} \sim \pi_{\theta_{\text{ref}}}(\cdot|\mathbf{x})} \left[\exp\left(\frac{r(\mathbf{x}, \mathbf{y})}{\beta} \right) \right],$$

the second partition term is eliminated if we take

$$\Delta z^\star(\mathbf{x}, \mathbf{y}^+, \mathbf{y}^-) = z^\star(\mathbf{x}, \mathbf{y}^+) - z^\star(\mathbf{x}, \mathbf{y}^-) = \frac{r(\mathbf{x}, \mathbf{y}^+) - r(\mathbf{x}, \mathbf{y}^-)}{\beta}.$$

If preferences follow a Bradley-Terry (logistic) model with temperature, then, we have

$$\mathbb{P}(\mathbf{y}^+ \succ \mathbf{y}^- \mid \mathbf{x}) = \sigma\left(\frac{r(\mathbf{x}, \mathbf{y}^+) - r(\mathbf{x}, \mathbf{y}^-)}{\beta} \right) = \sigma\left(\Delta z^\star(\mathbf{x}, \mathbf{y}^+, \mathbf{y}^-) \right).$$

To make z_θ mimics $z_\star$, we apply a pairwise logistic surrogate to the margin,

$$\begin{aligned} m_\theta(\mathbf{x}; \mathbf{y}^+, \mathbf{y}^-) &:= \frac{1}{\beta} \left(z_\theta(\mathbf{x}, \mathbf{y}^+) - z_\theta(\mathbf{x}, \mathbf{y}^-) \right) \\ &= \frac{1}{\beta} \left(\log \pi_\theta\left(\mathbf{y}^+ \mid \mathbf{x} \right) - \log \pi_\theta\left(\mathbf{y}^- \mid \mathbf{x} \right) - \log \pi_{\theta_{\text{ref}}}\left(\mathbf{y}^+ \mid \mathbf{x} \right) + \log \pi_{\theta_{\text{ref}}}\left(\mathbf{y}^- \mid \mathbf{x} \right) \right). \end{aligned}$$

The DPO loss is exactly the logistic loss that pushes m^θ to $\Delta z^\star$, i.e.,

$$L^{\text{DPO}}(\theta) = -\mathbb{E}_{(\mathbf{x}, \mathbf{y}^+, \mathbf{y}^-)} \left[\log \sigma\left(m_\theta(\mathbf{x}; \mathbf{y}^+, \mathbf{y}^-) \right) \right]$$

Note that this is something to minimize, and in the original introduction to DPO, we are writing the goal in terms of maximization. If we differentiate with respect to θ, we get

$$\begin{aligned} \nabla_\theta \left(-\log \sigma(m_\theta) \right) &= -\sigma(-m_\theta) \cdot \nabla_\theta m_\theta \\ &= -\frac{\sigma(-m_\theta)}{\beta} \left(\nabla_\theta \log \pi_\theta\left(\mathbf{y}^+ \mid \mathbf{x} \right) - \nabla_\theta \log \pi_\theta\left(\mathbf{y}^- \mid \mathbf{x} \right) \right) \\ &= -\frac{1}{\beta} \left(\sigma(-m_\theta) \nabla_\theta \log \pi_\theta\left(\mathbf{y}^+ \mid \mathbf{x} \right) - \sigma(-m_\theta) \nabla_\theta \log \pi_\theta\left(\mathbf{y}^- \mid \mathbf{x} \right) \right). \end{aligned}$$

As we can see, this is a weighted sum of the log-likelihood gradients, but the coefficient of the less preferred sample is negative, which means that one "weight" is negative. It is not representable as a weighted Negative Log Likelihood (NLL) form, GPRO generalizes DPO in spirits. To be specific, DPO uses a pairwise logistic surrogate focuses only on to relative differences, which leads to negative coefficients for the dispreferred sample.

10.13 On Policy and Off Policy

This is one of the most fundamental distinctions in reinforcement learning

- **On policy** refers to the case that the agent is learning about the same policy that it is currently following. It gathers experience and then update that exact strategy.
- **Off policy** refers to the case that the agent wants to learn about a different policy than the one it is using to act. It might behave randomly or follow an old policy, but still update another (better) policy.

The key difference is, in on policy, we learn from what we actually do and in off policy, we learn from what we (or the others) did, but update toward a different target strategy.

Some examples include:

1. **PPO:** This is on policy, because PPO collects rollouts (trajectories) using the current policy π_θ, and immediately uses them to update the same policy.
2. **DPO:** This is not really RL in the classical sense, DPO is usually described as offline preference-based fine-tuning. So in strict RL terms: it's neither on-policy nor off-policy, because it doesn't collect trajectories with a policy. (It behaves like an offline / off-policy method, since it learns from pre-collected data.)
3. **GPRO:** Same as DPO — not classical on-policy RL. It's trained on a static dataset of human preferencesm, so it's closer to off-policy / offline RLHF-style training.

This is actually related to the broader range of reinforcement learning problem. In reinfocrement learning, we usually have three types of methods:

1. Value based methods: learn the value of states or actions (e.g. Q function), and then derive a policy by picking the best action. Representive algorithms include Q-learning on small problems, and DQN (Deep-Q Networks) on large state spaces. They are usually off-policy.
2. Policy based methods: directly learn the policy $\pi_\theta\left(\mathbf{a} \mid \mathbf{s}\right)$. Representive algorithms include: REINFORCE (vanilla policy gradient), Actor-Critic (has both a policy and a value function helper), PPO (Proximal Policy Optimization), TRPO (Trust Region Policy Optimization). They are usually on policy. While they need fresh data each update, they work well with continuous or large action spaces.
3. Model based methods: learn a model of the dynamics of the environment (state transitions and rewards), then plan ahead inside that model. Representative algorithms include: Dyna-Q (classic), MuZero (modern, combines deep learning + planning). Although they can be very sample-efficient if the model is accurate, it is hard to learn good models for complex environments.

Value based methods: Let us recall the utimate goal of reinforcement learning:

$$\pi^\star = \arg\max_\pi \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t r_t\right],$$

and the definition of a Q function

$$Q^\pi(\mathbf{s}, \mathbf{a}) = \mathbb{E}\left[\text{future reward} \mid \text{start in } s, \text{take } a, \text{then follow } \pi\right] = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t r_t \mid \mathbf{s}_t = \mathbf{s}, \mathbf{a}_t = \mathbf{a}\right].$$

The idea is that if we know

$$Q^\star(\mathbf{s}, \mathbf{a}) = \max_\pi Q^\pi(\mathbf{s}, \mathbf{a}),$$

then we can find the corresponding best policy

$$\pi^\star(\mathbf{s}) = \arg\max_{\mathbf{a}} Q^\star(\mathbf{s}, \mathbf{a}).$$

SARSA: State-Action-Reward-State-Action (On policy) Update rule:

$$Q(\mathbf{s}_t, \mathbf{a}_t) \leftarrow Q(\mathbf{s}_t, \mathbf{a}_t) + \alpha\left[r_t + \gamma \cdot Q(\mathbf{s}_{t+1}, \mathbf{a}_{t+1}) - Q(\mathbf{s}_t, \mathbf{a}_t)\right],$$

which means that when we are in state $\mathbf{s}_t$ taking action $\mathbf{a}_t$ from our policy, we observe the reward r_t and land in the next state $\mathbf{s}_{t+1}$, our policy tells us to take some next action $\mathbf{a}_{t+1}$ in that state. In this case, the update

rule can be intepreted as update our $Q(\mathbf{s}_t, \mathbf{a}_t)$ so that it is closer to reward now + discounted reward of the (next state, next action), α is like a step size controlling how fast we proceed.

SARSA, as a result, is on policy and if our policy is ϵ-greedy (sometimes exploring randomly instead of picking the action with the highest Q-value so far.), SARSA will learn the value of that ϵ-greedy policy (SARSA converges to the value function of the behavior policy we actually follow), not necessarily the optimal greedy one.

Q-learning (Off-policy) Update rule:

$$Q(\mathbf{s}_t, \mathbf{a}_t) \leftarrow Q(\mathbf{s}_t, \mathbf{a}_t) + \alpha \left[r_t + \gamma \max_{\mathbf{a}'} Q(\mathbf{s}_{t+1}, \mathbf{a}') - Q(\mathbf{s}_t, \mathbf{a}_t) \right].$$

Instead of looking at the action we actually took next, we imagine what if we take the best possible action in $\mathbf{s}_{t+1}$, which results in the term $\max_{\mathbf{a}'} Q(\mathbf{s}_{t+1})$, and we are updating toward reward now + discounted value of the best possible next step.

As a result, Q-learning is off-policy: our behavior policy might explore randomly, but our updates always push you toward the optimal greedy policy. That's why Q-learning converges (under conditions) to $Q^\star$.

10.14 Domain Adaption

In machine learning, we usually assume that training and test data come from the same distribution. But in practice, this often fails (for example: training a sentiment classifier on Amazon reviews and testing it on Twitter posts). This motivates domain adaptation (DA), a subfield of transfer learning. Formally, a domain is formally defined as a pair

$$\mathcal{D} = (\mathcal{X}, P(X)),$$

where $\mathcal{X}$ is an input space (images, sentences), and $P(X)$ is a mariginal distribution over $\mathcal{X}$ (we denote the original distribution as $P(\mathcal{X}, \mathcal{Y})$), and a task is defined as

$$\mathcal{T} = (\mathcal{Y}, f),$$

where $\mathcal{Y}$ is the output (label) space, $f : \mathcal{X} \mapsto \mathcal{Y}$ is the true labeling function. We consider source domain $\mathcal{D}_S = (\mathcal{X}_S, P_S(X))$ and target domain $\mathcal{D}_T = (\mathcal{X}_T, P_T(X))$, we assume

1. **Different distributions:** $P_S(X) \neq P_T(X)$, but typically, $\mathcal{X}_S = \mathcal{X}_T$ and $\mathcal{Y}_S = \mathcal{Y}_T$.
2. **Same task (most classical setting):** $f_S = f_T = f$, variants exist where tasks differ, but then we are in broader transfer learning.
3. **Label scarcity:** Often, the source domain has many labeled samples: $\left\{ \left(x_i^S, y_i^S \right) \right\}_{i=1}^{n_S} \sim P_S(\mathcal{X}, \mathcal{Y})$ while target domain may have few or zero labeled samples: $\left\{ x_j^T \right\}_{j=1}^{n_T} \sim P_T(X)$.

We want to learn a hypothesis (model) $h : \mathcal{X} \mapsto \mathcal{Y}$ that minimizes target risk:

$$\epsilon_T(h) = \mathbb{E}_{(x,y)\sim P_T(x,y)} \left[\mathcal{L}(h(x), y) \right],$$

given most supervision comes from $\mathcal{D}_S$.

Typical approaches to handle this problem includes:

1. **Instance based methods**: reweight the the source samples so that they look like they were drawn from the target distribution.

$$\epsilon_T(h) = \mathbb{E}_{(x,y)\sim P_T(x,y)}\left[\frac{P_T(x)}{P_S(x)}\mathcal{L}(h(x), y)\right],$$

   there is no need to change the struture of the model, but the ratios are hard to estimate especially in high dimensions.

2. **Feature based methods**: Learn a transformation Φ, such that the source and target distributions are aligned in feature space $P_S(\Phi(X)) \approx P_T(\Phi(X))$, Φ can be viewed as a feature extractor (a mapping from the raw input space $\mathcal{X}$ to some latent space $\mathcal{Z}$), in practice, Φ is usually the encoder part of the model (e.g. the neural net before the final classifier head). For a neural model h_θ, we can think of

$$h_\theta(x) = g_\theta(\Phi(x)).$$

   The introduction of the feature map $\Phi(\cdot)$ allows us to transform soucre and target into a feature space $\mathcal{Z}$, where they will look very similar. For an exemplary pipeline: (i) an encoder Φ is trained so that features of source and target overlap. (ii) Classifier g_θ then works equally well across domains.

3. **Parameter based methods**: Assume the parameters of source and target models should share some structure, examples include shared encoder across domains, domain-specific classification heads. Useful when domains are similar but not identical. For a example, we assume some shared parameters across source and target, and some domain-specific parameters, we finetune a subset of layers on target domain data.

11 Miscellaneous

11.1 Chinchilla Optimality

Chinchilla optimalilty refers to the balance between model size (number of parameters) and dataset size (number of training tokens) for a fixed budget of compute.

- Before Chinchilla, the common practice (e.g. GPT-3) was to train very large models on relatively small datasets. This made the model undertrained, which means that they had more parameters than the dataset could properly fit.
- The Chinchilla paper by Deepmind in 2022 systematically showed that for a fixed budget of compute, the optimal stragety is not to make the model as large as possible, but instead to use a smaller model and train it on much more data.
- Roughly speaking, they found that the optimal scaling law would be letting the number of training tokens scale linearly with the number of parameters
- The empirical results show that the optimal data-to-model ratio to be around 20 tokens per paramete ($1\times$ case)[28]. If we train with fewer tokens per parameter such as $4 \sim 5$ tokens per parameter, the model would be under-trained, and if we train with more tokens per parameter such as $50 \sim 100$ tokens per parameter, the model would be over-trained.

In practice, GPT-3 had 175B parameters but was trained on 300B tokens. Chinchilla showed that 70B parameters trained on 1.4T tokens (with the same compute budget) achieves much better performance.

[28]We often see $2\times$ or $4\times$ cases where we increase the number of tokens 2 or 4 times.

11.2 Metrics (nats, bits, bpb)

The standard language modeling loss is the average of the negative log likelihood of the next token.

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

where $\mathbf{x} = (\mathbf{x}_1, \ldots, \mathbf{x}_L)$ is a sequence of tokens. The loss it self can be reported in nats (natural log base e) L_{nats} or bits (log base 2) L_{bits}. If we exponentiate the average loss, we get perplexity.

There is another related measure, namely bits per bytes (bpb), coming from viewing the model as a compression system, which connects to Shannon's source coding theorem. That is, the average number of bits needed to encode a symbol is equal to its entropy (expected negative log likelihood). Under this view, the bits L_{bits} indicates how many bits are needed to encode each token. This already gives us bits per token. However, tokens are not uniform units, because they vary in length, so bits per token is not directly comparable across tokenization schemes. This is where bpb comes in. Instead of normalizing by token count, normalize by raw UTF-8 byte count of the text. If we denote the dataset text length as T-tokens and B-bytes, then

$$\text{bpb} = \frac{T}{B}\cdot L_{\text{bits}}.$$

This makes the result more stable and comparable across models with different tokenizer.