A Visual, Dimension-Oriented Guide to Transformer Parallelism

November 6, 2025

Contents

List of Figures

1 Introduction

Transformers and their variants are now the dominant architecture for large-scale language and vision models. Modern systems combine them with multiple forms of parallelism (tensor, data, pipeline, and others) to train and serve models with hundreds of billions of parameters across many accelerators.

The high-level ideas are widely known, but the execution details are scattered across code bases, papers, and blog posts. As a result, even experienced practitioners often find it surprisingly hard to answer very concrete questions, such as:

- What are the exact tensors and matrix multiplications in a single Transformer layer, in both forward and backward passes?
- How many large matrix multiplications does backpropagation through attention and the MLP actually require?
- Where do All-Reduce and All-Gather operations appear when we introduce tensor or data parallelism?

This document is a visual, dimension-oriented guide to those questions. It does not propose a new architecture or a new training method. Instead, its goal is to reorganize standard, widely-used techniques—vanilla Transformers, backpropagation, and common parallel training schemes—into a form where their structure, tensor shapes, and costs are easy to see at a glance.

1.1 Scope and Goals

The focus is deliberately narrow:

- We consider a standard Transformer block with multi-head self-attention, a position-wise feed-forward (MLP/FFN) block, residual connections, and layer normalization.
- We work at the level of one layer at a time, plus the input embedding and output projection, rather than full end-to-end training pipelines.
- We emphasize how tensors and gradients flow through a layer and how different parallelism strategies modify that flow, rather than architectural variants or empirical benchmarks.

Within this scope, our aims are:

- To make the forward and backward computation graphs of a single Transformer layer explicit, including tensor shapes.
- To show how these graphs change (or do not change) under tensor parallelism, data parallelism, and their combination.
- To provide mental models and rules of thumb that help practitioners reason about compute, memory, and communication costs.

Everything in this document is a restatement of standard backpropagation and standard parallel training techniques. The contribution is purely expository: we aim to unify and clarify, not to introduce new algorithms.

1.2 Intended Audience

This guide is written for readers who:

- Are comfortable with basic linear algebra and the idea of gradients and backpropagation.
- Have at least a high-level understanding of Transformers (e.g., Q/K/V, attention weights, MLP block, residual connections).
- Want a concrete, implementation-oriented view of what actually happens inside a layer during forward and backward passes, especially in distributed settings.

The diagrams and notation are designed to be close to how large-model frameworks actually implement these models, without depending on any particular codebase.

1.3 Structure of the Document

The rest of the document proceeds from simple scalar gradients to full parallel Transformer layers:

- Section 2 provides a very light neural-network refresher: tensors, linear layers, MLPs, and the basic idea of backpropagation. This is only meant to fix notation and intuition; readers already familiar with these concepts can skim it.
- Section 2 also explains why sequence models need self-attention and how a standard Transformer block (embeddings, MHA, MLP, residuals, normalization) is organized at a high level.
- Section 3 introduces gradients and backpropagation from an operator point of view, setting up the abstract forward/backward nodes that our diagrams use.
- Section 4 defines the graphical notation: how we draw tensors, parameters, gradients, shapes, and communication collectives.
- Subsequent sections build up a single-node Transformer layer, with detailed forward and backward computation graphs for the embedding, MHA, MLP, and output projection.
- Later sections introduce tensor parallelism and data parallelism, and then combine these ideas into a DP+TP setting, where each data-parallel replica is itself a tensor-parallel group.

Readers interested mainly in high-level parallelism can skim the background section and focus on the diagrams in the parallelism sections. Those who want a detailed understanding of backpropagation inside a Transformer layer may find it useful to walk through the single-node graphs first and then treat the parallel variants as structured modifications of that base case.

2 Background: Neural Networks and Transformers

This section provides a minimal background on neural networks and Transformer models. The goal is not to re-teach deep learning, but to fix notation and high-level structure for the dimension-annotated diagrams in the rest of the document.

We first review how we represent tensors and linear layers, then move from simple MLPs to sequence models with self-attention and standard Transformer blocks. All of these are widely-used, standard techniques; we only reorganize them in a way that will make the later graphical representations easy to follow.

2.1 Basic Notation and Tensors

We view a neural network as a composition of simple operators applied to tensors. A model with L layers can be written as

$$X_0 \xrightarrow{f_1} X_1 \xrightarrow{f_2} \cdots \xrightarrow{f_L} X_L,$$

where

- X_0 is the input tensor,
- X_L is the final output,
- each f_ℓ is a local operation (e.g., a matrix multiplication, a non-linearity, a normalization layer) with its own parameters θ_ℓ.

We use the following conventions throughout:

- Bold uppercase letters like $\mathbf{X}$ for tensors.
- Plain uppercase letters like W for weight matrices.
- Plain lowercase letters like b for bias vectors.
- Gradients are prefixed by d, e.g., $d\mathbf{X}$, dW.

Unless otherwise stated, the main hidden representations in this document are 3-D tensors of shape

$$\mathbf{X} \in \mathbb{R}^{B \times S \times D},$$

where B is the batch size, S is the sequence length (number of tokens), and D is the hidden (model) dimension.

For weights and other parameters we mostly use:

- 2-D matrices, e.g., $W \in \mathbb{R}^{D_{\text{in}} \times D_{\text{out}}}$,
- 1-D vectors, e.g., $b \in \mathbb{R}^{D_{\text{out}}}$.

Later figures explicitly label these shapes on edges (e.g., $[B, S, D]$, $[D, D_{\text{ff}}]$), so it is useful to keep this convention in mind.

2.2 Linear Layers and MLP Blocks

The main computational workhorse in a Transformer block is the fully-connected (linear) layer and its composition into an MLP (or FFN) block.

Given an input tensor $\mathbf{X} \in \mathbb{R}^{B \times S \times D_{\text{in}}}$, a linear layer with weight $W \in \mathbb{R}^{D_{\text{in}} \times D_{\text{out}}}$ and bias $b \in \mathbb{R}^{D_{\text{out}}}$ applies the same affine transformation independently to all (b, s) positions:

$$\mathbf{Y}[b, s, :] = \mathbf{X}[b, s, :]W + b.$$

We can write this compactly as

$$\mathbf{Y} = \text{Linear}(\mathbf{X}; W, b).$$

A typical Transformer MLP block uses two such linear layers with an intermediate non-linearity:

$$\mathbf{X} \xrightarrow{\text{linear up}} \mathbf{Z}_{\text{up}} \xrightarrow{\phi} \mathbf{Z}_{\text{act}} \xrightarrow{\text{linear down}} \mathbf{Y},$$

where

- $\mathbf{X}, \mathbf{Y} \in \mathbb{R}^{B \times S \times D}$,
- $\mathbf{Z}_{\text{up}}, \mathbf{Z}_{\text{act}} \in \mathbb{R}^{B \times S \times D_{\text{ff}}}$,
- D_{ff} is an intermediate feed-forward dimension, usually larger than D,
- ϕ is an elementwise non-linearity (e.g., GELU, ReLU, or similar).

Later sections draw each of these linear and non-linear steps as separate nodes with explicit tensor shapes, so that both forward and backward flows are easy to trace.

2.3 Sequence Modeling and Self-Attention

The previous subsection treats inputs as independent vectors. However, many applications involve ordered sequences:

- Text: a sentence or paragraph as a sequence of tokens.
- Time series: a sequence of measurements over time.
- Audio or video: a sequence of frames.

We represent a batch of sequences as $\mathbf{X} \in \mathbb{R}^{B \times S \times D}$, where S is the sequence length. The model must capture dependencies across positions in the same sequence, not just transform each token independently.

Recurrent architectures (RNNs, LSTMs, GRUs) handle this by updating hidden states step by step. Transformer models instead use self-attention, which allows each position to directly attend to other positions in the same sequence.

Given hidden states $\mathbf{X} \in \mathbb{R}^{B \times S \times D}$, self-attention first projects them into queries (Q), keys (K), and values (V):

$$\mathbf{Q} = \mathbf{X}W_Q, \quad \mathbf{K} = \mathbf{X}W_K, \quad \mathbf{V} = \mathbf{X}W_V,$$

where $W_Q, W_K, W_V \in \mathbb{R}^{D \times D}$ or, more generally, from D to some head dimension. The attention scores between positions are then computed as scaled dot products between queries and keys, followed by a softmax, and used to form weighted sums of the values.

In multi-head attention (MHA), we use N_H heads in parallel, each with head dimension D_h, such that

$$D = N_H \cdot D_h.$$

Conceptually, the shapes involved are:

- Input / output hidden states: $\mathbf{X}, \mathbf{Y} \in \mathbb{R}^{B \times S \times D}$,
- Per-head projections: $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{B \times N_H \times S \times D_h}$,
- Attention scores: $\mathbf{A} \in \mathbb{R}^{B \times N_H \times S \times S}$.

Our diagrams in later sections make these shapes explicit and show exactly where the large matrix multiplications occur in both forward and backward passes.

2.4 Standard Transformer Block Structure

A standard Transformer block combines self-attention, an MLP block, normalization, and residual connections into a fixed pattern. There are several equivalent variants in the literature (e.g., pre-norm vs. post-norm). For the purposes of this document, we assume a typical pre-norm formulation:

1. **Input and layer normalization.** The block receives an input $\mathbf{X}_{\text{in}} \in \mathbb{R}^{B\times S\times D}$ and applies layer normalization:
$$\mathbf{X}_1 = \text{LayerNorm}(\mathbf{X}_{\text{in}}).$$
2. **Multi-head self-attention (MHA).** Self-attention produces an output $\mathbf{H}_{\text{att}} \in \mathbb{R}^{B\times S\times D}$, often including dropout:
$$\mathbf{H}_{\text{att}} = \text{MHA}(\mathbf{X}_1).$$
3. **First residual connection.**
$$\mathbf{X}_2 = \mathbf{X}_{\text{in}} + \mathbf{H}_{\text{att}}.$$
4. **Second layer normalization.**
$$\mathbf{X}_3 = \text{LayerNorm}(\mathbf{X}_2).$$
5. **MLP (feed-forward) block.**
$$\mathbf{H}_{\text{mlp}} = \text{MLP}(\mathbf{X}_3),$$
where the MLP has the structure described in the previous subsection.
6. **Second residual connection.**
$$\mathbf{X}_{\text{out}} = \mathbf{X}_2 + \mathbf{H}_{\text{mlp}}.$$

Additional components such as dropout, bias terms, and positional encodings can be inserted, but from the perspective of our dimension-annotated computation graphs, the key ingredients are:

- Repeated use of tensors of shape $[B, S, D]$,
- Multi-head attention tensors with shapes involving $[N_H, D_h]$ and $[S, S]$,
- Linear layers with weights of shapes like $[D, D_{\text{ff}}]$, $[D_{\text{ff}}, D]$, $[D, D]$,
- Residual additions and layer normalization applied along the hidden dimension.

We also assume:

- An input embedding (and possibly positional embedding) that maps token IDs or features into $[B, S, D]$,
- A final output projection that maps hidden states back to vocabulary logits or task-specific outputs.

These embeddings and projections are treated as additional linear layers in our diagrams.

2.5 Connection to the Rest of the Document

The purpose of this background is simply to establish what a standard Transformer layer looks like and which tensor shapes and operators appear inside it. All of these are standard and widely-used; we do not propose any new model.

Starting from the next section, we shift from this high-level architectural view to a more operator-centric view:

- We represent each forward operation (e.g., matrix multiplication, softmax, layer norm, residual addition) as a node in a computation graph.
- We introduce matching backward operators that consume upstream gradients and produce gradients with respect to inputs and parameters.
- We then apply this framework to single-node Transformer layers and later extend it to tensor parallelism, data parallelism, and their combination.

The rest of the document can thus be read as a detailed, visual unpacking of the standard Transformer block defined in this section.

3 Gradients and Backpropagation Basics

This section introduces the basic notions of loss functions, gradients, and backpropagation from an abstract operator point of view. In the figures later in the document we mainly show how tensors flow along edges; the detailed Jacobian matrices are not drawn explicitly. Instead, we use a compact notation for local backward operators such as $\mathrm{d}f$, which map upstream gradients on the outputs of a node to gradients on its inputs.

3.1 Scalar Loss and Gradient Notation

Training a Transformer model is formulated as minimizing a scalar loss function $\mathcal{L}(\theta)$ with respect to the model parameters θ. For a batch of input–target pairs $(\mathbf{X}, \mathbf{Y}_{\text{targets}})$, the model produces predictions

$$\mathbf{Y} = f_\theta(\mathbf{X}),$$

and a scalar loss

$$\mathcal{L} = \mathcal{L}(\mathbf{Y}, \mathbf{Y}_{\text{targets}}).$$

We use the differential-style notation $\mathrm{d}\theta = \partial\mathcal{L}/\partial\theta$ for gradients, and we also write $\nabla\theta$ as a shorthand for the same quantity. For example,

$$\mathrm{d}\mathbf{X} \;=\; \nabla\mathbf{X} \;=\; \frac{\partial\mathcal{L}}{\partial\mathbf{X}}.$$

In the diagrams, these gradients appear as edges labeled $\mathrm{d}\mathbf{X}$, $\mathrm{d}\mathbf{W}$, etc. together with their tensor shapes such as $[B, S, D]$ or $[D, D]$.

3.2 Single-Input Nodes and Backward Operators

Consider a single node in a computation graph with forward computation

$$\mathbf{y} = f(\mathbf{x}),$$

where $\mathbf{x}$ and $\mathbf{y}$ are vectors or tensors. If the loss $\mathcal{L}$ depends on $\mathbf{y}$, then by the chain rule

$$\mathrm{d}\mathbf{x} = \frac{\partial\mathcal{L}}{\partial\mathbf{x}} = \frac{\partial\mathcal{L}}{\partial\mathbf{y}} \cdot \frac{\partial\mathbf{y}}{\partial\mathbf{x}} = \mathrm{d}\mathbf{y} \cdot \frac{\partial f}{\partial\mathbf{x}}.$$

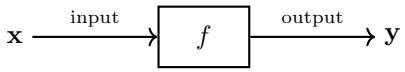
Brief note on Jacobians: When $\mathbf{x}$ and $\mathbf{y}$ are vectors or tensors, the term $\frac{\partial\mathbf{y}}{\partial\mathbf{x}}$ is formally a *Jacobian matrix* $\mathbf{J}_f(\mathbf{x})$, and the gradient $\mathrm{d}\mathbf{x}$ is computed as a Jacobian-vector product: $\mathrm{d}\mathbf{x} = \mathbf{J}_f(\mathbf{x})^T \mathrm{d}\mathbf{y}$. However, in practice we never materialize the full Jacobian matrix—it would be prohibitively expensive for high-dimensional tensors. Instead, automatic differentiation frameworks directly compute the gradient using efficient chain-rule implementations tailored to each operation.

We introduce an **abstract backward operator** $\mathrm{d}f$ and write

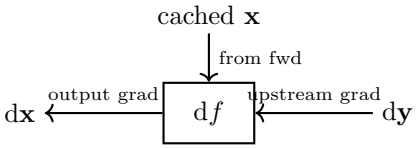
$$\boxed{\mathrm{d}\mathbf{x} = \mathrm{d}f(\mathbf{x}, \mathrm{d}\mathbf{y})},$$

which computes the gradient $\mathrm{d}\mathbf{x}$ given the forward input $\mathbf{x}$ (cached) and the upstream gradient $\mathrm{d}\mathbf{y}$.

Graphical representation:



Forward: $\mathbf{y} = f(\mathbf{x})$



Backward: $\mathrm{d}\mathbf{x} = \mathrm{d}f(\mathbf{x}, \mathrm{d}\mathbf{y})$

The diagram shows the key idea: the backward node $\mathrm{d}f$ takes two inputs—the cached forward input $\mathbf{x}$ and the upstream gradient $\mathrm{d}\mathbf{y}$—and produces the output gradient $\mathrm{d}\mathbf{x}$ that flows to the previous layer.

Softmax, dropout, layer normalization, and many other operators in a Transformer layer are special cases of this pattern. Their concrete backward rules are described in terms of such operators $\mathrm{d}f$.

3.3 Nodes with Multiple Inputs

Many nodes in a Transformer layer have several inputs. For example, a matrix multiplication node uses both an activation tensor and a weight matrix, and a layer-normalization node uses inputs as well as learned scale and shift parameters. Abstractly, we write

$$\mathbf{y} = f(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_k),$$

where each $\mathbf{x}_i$ may be a tensor of its own.

Given the upstream gradient $\mathrm{d}\mathbf{y} = \partial\mathcal{L}/\partial\mathbf{y}$, the chain rule yields gradients with respect to all inputs:

$$\mathrm{d}\mathbf{x}_i = \frac{\partial\mathcal{L}}{\partial\mathbf{y}} \cdot \frac{\partial\mathbf{y}}{\partial\mathbf{x}_i} = \mathrm{d}\mathbf{y} \cdot \frac{\partial f}{\partial\mathbf{x}_i}, \quad i = 1, \ldots, k.$$

We encode this in an abstract backward operator that produces all input gradients simultaneously:

$$\boxed{(\mathrm{d}\mathbf{x}_1, \ldots, \mathrm{d}\mathbf{x}_k) = \mathrm{d}f(\mathbf{x}_1, \ldots, \mathbf{x}_k, \mathrm{d}\mathbf{y})}.$$

3.3.1 Computational Cost: Forward vs. Backward

Can we estimate backward cost from forward cost? Yes, there are general rules:

- **Linear operations (e.g., matrix multiplication):**

  $$\text{Backward cost} \approx 2 \times \text{Forward cost}$$

  For $\mathbf{Y} = \mathbf{X}\mathbf{W}$, the forward pass requires one matmul, while backward requires two: $\mathrm{d}\mathbf{X} = \mathrm{d}\mathbf{Y}\mathbf{W}^T$ and $\mathrm{d}\mathbf{W} = \mathbf{X}^T\mathrm{d}\mathbf{Y}$.

- **Element-wise operations (e.g., ReLU, GELU, dropout):**

  $$\text{Backward cost} \approx \text{Forward cost}$$

  For $\mathbf{y} = g(\mathbf{x})$, the backward pass $\mathrm{d}\mathbf{x} = \mathrm{d}\mathbf{y} \odot g'(\mathbf{x})$ has similar complexity to the forward pass.

- **Softmax and normalization:**

$$\text{Backward cost} \approx (1\text{–}2) \times \text{Forward cost}$$

The backward pass involves additional dot products or reductions, but remains of the same computational order.

- **Attention mechanism:**

$$\text{Backward cost} \approx 2 \times \text{Forward cost}$$

Each matmul in forward (e.g., $\mathbf{QK}^T$, attention scores $\times$ $\mathbf{V}$) generates two matmuls in backward.

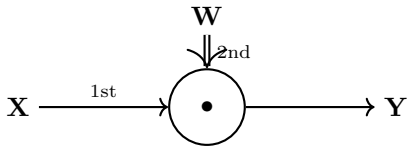
Rule of thumb for full networks:

$$\boxed{\text{Total backward cost} \approx 2\text{–}3 \times \text{Total forward cost}}$$

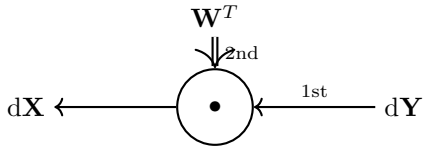
This factor of 2-3 is why training (forward + backward + optimizer step) takes roughly 3-4$\times$ the time of inference (forward only).

Example: Matrix Multiplication

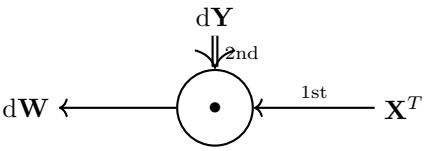
For matrix multiplication $\mathbf{Y} = \mathbf{XW}$, the backward pass computes two separate gradients using two matrix multiplications. This separation makes the computational cost and memory usage explicit.



Forward: $\mathbf{Y} = \mathbf{XW}$ (1 matmul)



$\mathrm{d}\mathbf{X} = \mathrm{d}\mathbf{Y}\mathbf{W}^T$



$\mathrm{d}\mathbf{W} = \mathbf{X}^T\mathrm{d}\mathbf{Y}$

Backward: 2 matmuls $\Rightarrow$ 2$\times$ forward cost

Concrete FLOPs analysis: For $\mathbf{X} \in \mathbb{R}^{B\times S\times D}$ and $\mathbf{W} \in \mathbb{R}^{D\times D}$:

$$\begin{aligned}
\text{Forward:} \quad & \mathbf{Y} = \mathbf{XW} \quad \to \quad 2BSD^2 \text{ FLOPs} \\
\text{Backward:} \quad & \mathrm{d}\mathbf{X} = \mathrm{d}\mathbf{Y}\mathbf{W}^T \quad \to \quad 2BSD^2 \text{ FLOPs} \\
& \mathrm{d}\mathbf{W} = \mathbf{X}^T\mathrm{d}\mathbf{Y} \quad \to \quad 2BSD^2 \text{ FLOPs} \\
\text{Total backward:} \quad & 4BSD^2 = 2 \times (\text{forward cost})
\end{aligned}$$

In the diagrams, we follow the convention that for matrix multiplication nodes:

- The **first operand** enters horizontally (from left in forward, from right in backward)
- The **second operand** enters vertically from above (shown with a double arrow)
- The **output** exits horizontally (to right in forward, to left in backward)

This explicit representation allows us to:

- Count FLOPs precisely for each gradient computation
- Track memory usage for intermediate tensors
- Identify opportunities for parallelization (these two matmuls can run in parallel)
- Match the actual implementation in frameworks like PyTorch and JAX

Throughout the document, we follow this convention: whenever the backward pass involves multiple distinct operations, we draw them as separate nodes rather than hiding them inside a single abstract $\mathrm{d}f$ operator.

3.4 Computation Graph and Backpropagation

A full Transformer layer can be viewed as a composition of simpler operations:

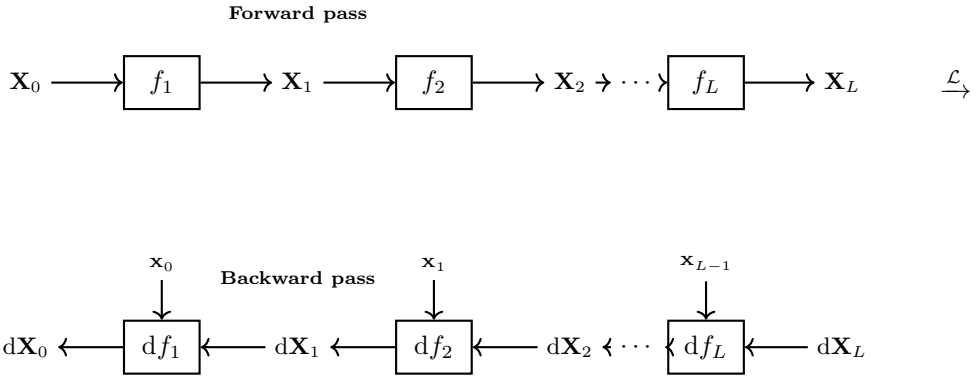
$$\mathbf{X}_0 \xrightarrow{f_1} \mathbf{X}_1 \xrightarrow{f_2} \mathbf{X}_2 \xrightarrow{\cdots} \mathbf{X}_L,$$

where $\mathbf{X}_0$ is the input to the layer and $\mathbf{X}_L$ is the final output before the loss. Each f_ℓ is a local operator such as a matrix multiplication, a nonlinearity, a dropout, or a normalization.

Backpropagation proceeds in reverse order. Starting from $\mathrm{d}\mathbf{X}_L = \partial\mathcal{L}/\partial\mathbf{X}_L$, we apply the corresponding backward operator for each node:

$$\mathrm{d}\mathbf{X}_\ell = \mathrm{d}f_\ell(\mathbf{X}_\ell, \mathrm{d}\mathbf{X}_{\ell+1}), \quad \ell = L-1, L-2, \ldots, 0.$$

Graphical representation of a computation chain:



Key observations:

- **Forward**: Data flows left to right through function nodes f_ℓ
- **Backward**: Gradients flow right to left through backward nodes $\mathrm{d}f_\ell$
- **Cache**: Each backward node $\mathrm{d}f_\ell$ receives the cached forward state from above
- **Chain rule**: $\mathrm{d}\mathbf{X}_\ell = \mathrm{d}f_\ell(\mathbf{X}_\ell, \mathrm{d}\mathbf{X}_{\ell+1})$ for $\ell = L-1, \ldots, 0$

If f_ℓ depends on additional inputs (e.g. parameters), then its backward operator also produces gradients with respect to those inputs, as discussed below.

In the diagrams, we emphasize this process by drawing:

- **forward edges** for $\mathbf{X}_\ell$ flowing into the forward nodes f_ℓ,
- **backward edges** for $\mathrm{d}\mathbf{X}_\ell$ flowing out of the corresponding backward nodes $\mathrm{d}f_\ell$.

The detailed formulas that define each local operator $\mathrm{d}f_\ell$ are hidden inside the node and explained in the operator dictionary of Section 4.

3.5 Parameter Gradients and Updates

Parameters such as weight matrices and bias vectors enter the graph as additional inputs to some node. For example, consider a matrix multiplication

$$\mathbf{Y} = \mathbf{X}\mathbf{W},$$

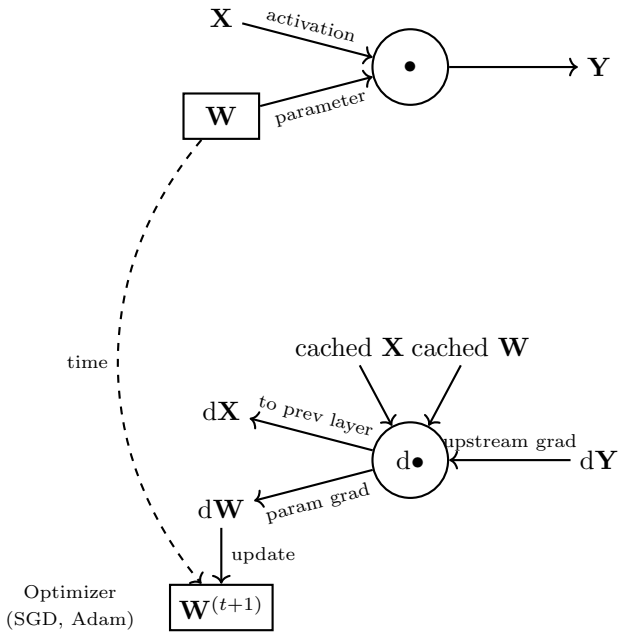
where $\mathbf{X}$ is an activation tensor and $\mathbf{W}$ is a weight matrix. We view this as a function of two inputs,

$$\mathbf{Y} = f(\mathbf{X}, \mathbf{W}).$$

The corresponding backward operator produces both activation and parameter gradients:

$$\boxed{(\mathrm{d}\mathbf{X}, \mathrm{d}\mathbf{W}) = \mathrm{d}f(\mathbf{X}, \mathbf{W}, \mathrm{d}\mathbf{Y})}.$$

Parameter gradient flow:



An optimizer then uses the parameter gradients to update the parameters. For example, stochastic gradient descent with learning rate η performs

$$\theta^{(t+1)} = \theta^{(t)} - \eta\, \mathrm{d}\theta^{(t)}.$$

Key distinction:

- **Activation gradients** ($\mathrm{d}\mathbf{X}$): Flow to the previous layer in the backward pass
- **Parameter gradients** ($\mathrm{d}\mathbf{W}$): Accumulated and used by the optimizer to update weights

In this document we do not draw optimizer steps in the diagrams; we only show how $\mathrm{d}\mathbf{W}$ and $\mathrm{d}\mathbf{b}$ are computed by the backward nodes.

3.6 Connection to the Diagrams

The detailed MHA, MLP, and output-projection figures in later sections are best read with this abstract picture in mind:

- Each **forward node** (e.g. `SM`, `S`, `DO`, `LN`, `matmul`) represents a mapping $\mathbf{y} = f(\mathbf{x}_1, \ldots, \mathbf{x}_k)$.
- Each corresponding **backward node** (e.g. `dSM`, `dS`, `dDO`, `dLN`, `dMatmul`) represents the operator

$$(\mathrm{d}\mathbf{x}_1, \ldots, \mathrm{d}\mathbf{x}_k) = \mathrm{d}f(\mathbf{x}_1, \ldots, \mathbf{x}_k, \mathrm{d}\mathbf{y}),$$

  implemented using the appropriate Jacobian-transpose formulas for that operator.
- Edges labeled with tensors such as $\mathbf{X}$, $\mathbf{H}$, $\mathbf{Q}$, $\mathbf{K}$, $\mathbf{V}$, $\mathbf{AS}$, and their gradients $\mathrm{d}\mathbf{X}$, $\mathrm{d}\mathbf{Q}$, $\mathrm{d}\mathbf{W}$, etc., capture only the flow of data, together with compact shape annotations like $[B, S, D]$ or $[B, N_H, S, D_h]$.

In the next section we define the graphical notation and operator dictionary used in the figures, and we specialize the abstract backward operator $\mathrm{d}f$ to concrete nodes such as softmax (`S`/`dS`), scale/mask (`SM`/`dSM`), dropout (`DO`/`dDO`), and layer normalization (`LN`/`dLN`).

4 Graphical Notation and Figure Conventions

The diagrams in this document are designed to show how tensors flow through a Transformer layer and its parallel variants. This section summarizes the graphical notation, including tensor-shape labels, node types, edge styles, and the small operator dictionary for the most common nodes (matmul, softmax, scale/mask, dropout, layer normalization, communication, and broadcast).

Throughout the figures, the goal is to emphasize the flow of tensors along edges. The exact Jacobian matrices for each operation are not drawn; instead, the backward nodes are understood as the abstract operators $\mathrm{d}f$ introduced in Section 3.

4.1 Tensor Shapes and Index Notation

We use a consistent braced notation for tensor shapes. Instead of writing $\mathbb{R}^{B\times S\times D}$, we annotate edges in the diagrams with labels such as

$$[B, S, D], \quad [B, N_H, S, D_h], \quad [D, D_{ff}],$$

directly next to the arrows. This makes it easier to match each edge to a particular dimension ordering.

The main symbols are:

- B: batch size.
- S: sequence length (number of tokens per sequence).
- D: model (hidden) dimension.
- D_{ff}: intermediate MLP (feed-forward) dimension.
- N_H: number of attention heads.
- D_h: per-head dimension, typically $D_h = D/N_H$.

Typical tensor shapes in the diagrams include:

- $\mathbf{X} \in [B, S, D]$: input or hidden states.
- $\mathbf{H} \in [B, S, D]$: normalized or intermediate states.
- $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in [B, N_H, S, D_h]$: projected queries, keys, and values.
- $\mathbf{AS} \in [B, N_H, S, S]$: attention scores after scaling/masking and softmax.
- $\mathbf{Y} \in [B, S, D]$: output of a Transformer block or layer.

Under tensor parallelism (TP) and data parallelism (DP) we mostly keep the same shape notation on edges. For example, a TP shard that actually stores a slice of width D/N_T may still be labeled $[B, S, D]$ in an end-to-end figure when we want to focus on the logical model dimension rather than the physical shard size. When necessary, shard dimensions such as $[B, S, D/N_T]$ or $[B, N_H/N_T, S, D_h]$ are written explicitly.

Gradients use the same shape conventions. For example:

$$\mathrm{d}\mathbf{X} \in [B, S, D], \quad \mathrm{d}\mathbf{W}_Q \in [D, D], \quad \mathrm{d}\mathbf{W}_{\mathrm{up}} \in [D, D_{ff}].$$

4.2 Nodes, Edges, and Arrow Styles

The diagrams are drawn as computation graphs. Nodes represent local operations; edges represent tensors flowing between them.

4.2.1 Forward vs. Backward Arrows

We distinguish between forward and backward flows:

- **Overall flow diagrams** (e.g. the top-level Transformer flow) use:
  - solid arrows for forward activations (e.g. $\mathbf{X}$ to $\mathbf{Y}$);
  - dashed arrows for backward gradients (e.g. $\mathrm{d}\mathbf{Y}$ to $\mathrm{d}\mathbf{X}$).
- **Detailed backward diagrams** (e.g. MHA backward, MLP backward) use thicker arrows with different styles (single vs. double) to distinguish:
  - gradient flow along the main backward path,
  - cached forward values reused as secondary inputs to backward nodes.

In all cases, the arrow direction follows the direction of computation for forward edges and the direction of gradient propagation for backward edges.

4.2.2 Node Types

We use a small set of recurring node types:

Matrix multiplication. A matrix multiplication node is drawn as a circle containing a dot:

In the forward pass this corresponds to an operation such as $\mathbf{Y} = \mathbf{X}\mathbf{W}$. In the backward diagrams the corresponding dNode (e.g. `dMatmul`) implements the abstract backward operator

$$(\mathrm{d}\mathbf{X}, \mathrm{d}\mathbf{W}) = \mathrm{d}f(\mathbf{X}, \mathbf{W}, \mathrm{d}\mathbf{Y}),$$

as described in Section 3.

Addition and residuals. Elementwise addition is drawn as a circle containing a plus:

This is used for bias addition, combining residual connections, and aggregating multiple gradient contributions. Small circles labeled $\sum$ denote explicit summations over batch and/or sequence dimensions (e.g. $\sum_{B,S}$ for bias-gradient accumulation).

Generic rectangular operators. Many local operations (layer normalization, nonlinearity, dropout, scale/mask, reshape, transpose) are drawn as rectangles with short labels such as `LN`, `GL`, `DO`, `SM`, `R`, or `T`. Their backward counterparts are labeled with a leading "d", for example `dLN`, `dDO`, `dSM`. Each such dNode implements the corresponding backward operator $\mathrm{d}f(\cdot, \ldots, \cdot, \mathrm{d}\mathbf{y})$.

Communication nodes. Distributed communication collectives are drawn as small rectangular nodes with labels such as:

- `AR`: All-Reduce.
- `AG`: All-Gather.

The arrows entering/leaving these nodes indicate which tensors are participating in the collective, and the shape annotations show the logical tensor size before and after the communication.

4.3 Forward and Backward Nodes: Abstract View

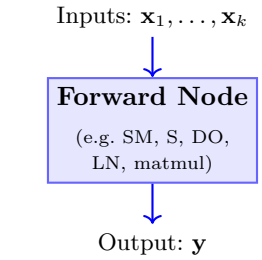
Most operators in a Transformer layer can be written abstractly as

$$\mathbf{y} = f(\mathbf{x}_1, \ldots, \mathbf{x}_k),$$

where $\mathbf{x}_1, \ldots, \mathbf{x}_k$ include both activations and parameters (such as weight matrices or bias vectors). In Section 3 we introduced the abstract backward operators

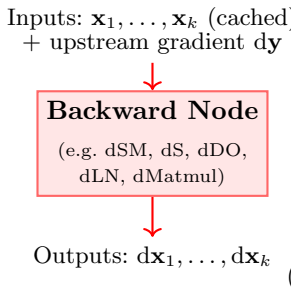
$$\mathrm{d}\mathbf{x}_i = \mathrm{d}_{\mathbf{x}_i} f(\mathbf{x}_1, \ldots, \mathbf{x}_k, \mathrm{d}\mathbf{y}), \quad i = 1, \ldots, k.$$

How to read nodes in our diagrams:



Represents

$$\mathbf{y} = f(\mathbf{x}_1, \ldots, \mathbf{x}_k)$$



Represents

$$(\mathrm{d}\mathbf{x}_1, \ldots, \mathrm{d}\mathbf{x}_k) = \mathrm{d}f(\mathbf{x}_1, \ldots, \mathbf{x}_k, \mathrm{d}\mathbf{y})$$

In the diagrams:

- A **forward node** (e.g. `S`, `SM`, `DO`, `LN`, `matmul`) represents the mapping $\mathbf{y} = f(\mathbf{x}_1, \ldots, \mathbf{x}_k)$.
- The corresponding **backward node** (labeled with a leading "d", e.g. `dS`, `dSM`, `dDO`, `dLN`) represents the mapping

  $$(\mathrm{d}\mathbf{x}_1, \ldots, \mathrm{d}\mathbf{x}_k) = \mathrm{d}f(\mathbf{x}_1, \ldots, \mathbf{x}_k, \mathrm{d}\mathbf{y}),$$

  i.e. it consumes the upstream gradient $\mathrm{d}\mathbf{y}$ and the necessary cached forward inputs, and produces gradients for all inputs.

The purpose of this section is not to re-derive all Jacobian formulas, but to provide a dictionary that tells the reader what each node means in the diagrams and how to read its inputs/outputs at the level of tensors and gradients.

4.4 Operator Dictionary: Forward and Backward

We now describe the most common node types used in the MHA, MLP, and output-projection diagrams. For each operator we briefly summarize the forward computation and the corresponding backward operator in the abstract notation $\mathrm{d}f(\cdot, \ldots, \cdot, \mathrm{d}\mathbf{y})$.

4.4.1 Matrix Multiplication (Matmul)

Forward. A matmul node computes

$$\mathbf{Y} = \mathbf{X}\mathbf{W},$$

with shapes such as $\mathbf{X} \in [B, S, D]$, $\mathbf{W} \in [D, D]$, and $\mathbf{Y} \in [B, S, D]$. The same pattern appears in the MLP block with $\mathbf{W}_{\text{up}} \in [D, D_{ff}]$ or $\mathbf{W}_{\text{down}} \in [D_{ff}, D]$.

Backward. The backward node `dMatmul` implements

$$(\mathrm{d}\mathbf{X}, \mathrm{d}\mathbf{W}) = \mathrm{d}f(\mathbf{X}, \mathbf{W}, \mathrm{d}\mathbf{Y}),$$

with the usual formulas

$$\mathrm{d}\mathbf{X} = \mathrm{d}\mathbf{Y}\mathbf{W}^T, \quad \mathrm{d}\mathbf{W} = \mathbf{X}^T\mathrm{d}\mathbf{Y},$$

applied with appropriate reshaping for batched tensors. In the diagrams, $\mathbf{X}$ and $\mathbf{W}$ (or their transposes) are supplied to the `dMatmul` node via double arrows, and outgoing arrows carry $\mathrm{d}\mathbf{X}$ and $\mathrm{d}\mathbf{W}$.

4.4.2 Broadcast (BC)

In many diagrams we annotate bias addition by a term such as $\mathrm{BC}_{B,S}(\mathbf{b}_0)$, which denotes a logical broadcast of a 1-D bias vector $\mathbf{b}_0 \in [D]$ or $[D_{ff}]$ across the batch and sequence dimensions to match a tensor of shape $[B, S, D]$ or $[B, S, D_{ff}]$.

Forward. Conceptually,

$$\mathbf{Y} = \mathbf{X} + \mathrm{BC}_{B,S}(\mathbf{b}_0),$$

where $\mathrm{BC}_{B,S}(\mathbf{b}_0)$ is a tensor in $[B, S, D]$ obtained by repeating $\mathbf{b}_0$ over the B and S dimensions. In the diagrams we typically draw only the addition node and label the edge near the bias with $\mathrm{BC}_{B,S}(\mathbf{b}_0)$ to indicate that the bias is broadcast in this way.

Backward. The backward contribution to $\mathrm{d}\mathbf{b}_0$ is obtained by summing $\mathrm{d}\mathbf{Y}$ over the broadcast dimensions:

$$\mathrm{d}\mathbf{b}_0 = \sum_{b=1}^{B} \sum_{s=1}^{S} \mathrm{d}\mathbf{Y}_{b,s,:},$$

which is represented in the diagrams by a small summation node labeled $\sum_{B,S}$. The gradient $\mathrm{d}\mathbf{X}$ simply inherits $\mathrm{d}\mathbf{Y}$, since the addition is symmetric.

4.4.3 Scale/Mask Node (SM, dSM)

In the attention mechanism, raw scores $\mathbf{A}$ from $\mathbf{Q}\mathbf{K}^T$ are scaled and masked before softmax. This is represented by a node labeled `SM` (scale/mask).

Forward (SM). Given attention scores $\mathbf{A} \in [B, N_H, S, S]$, the `SM` node computes

$$\mathbf{Z} = \mathrm{SM}(\mathbf{A}) = \alpha\mathbf{A} + \mathbf{M},$$

where $\alpha = 1/\sqrt{D_h}$ is a scalar and $\mathbf{M}$ encodes the mask (e.g. large negative values at disallowed positions). The shape of $\mathbf{Z}$ matches $\mathbf{A}$. The forward pass caches the scaling factor and the mask pattern.

Backward (dSM). The backward node `dSM` is the abstract operator

$$\mathrm{d}\mathbf{A} = \mathrm{dSM}(\mathbf{A}, \mathrm{d}\mathbf{Z}),$$

with

$$\mathrm{dSM}(\mathbf{A}, \mathrm{d}\mathbf{Z}) = \alpha\,\mathrm{d}\mathbf{Z},$$

and no gradient is propagated into the fixed mask $\mathbf{M}$. In the diagram, $\mathbf{A}$ and $\mathrm{d}\mathbf{Z}$ enter the `dSM` node, and $\mathrm{d}\mathbf{A}$ exits as the gradient with respect to the raw attention scores.

4.4.4 Softmax Node (S, dS)

The node labeled `S` performs softmax over the key dimension of the attention scores.

Forward (S). For a fixed batch b, head h, and query position s, let $\mathbf{z} \in \mathbb{R}^S$ be the vector of scores over keys. Softmax produces

$$\mathbf{p} = \mathrm{softmax}(\mathbf{z}), \quad p_i = \frac{e^{z_i}}{\sum_j e^{z_j}}.$$

This is applied to every (b, h, s), so the overall shape $[B, N_H, S, S]$ is preserved. The forward pass typically caches $\mathbf{p}$ (or $\mathbf{z}$).

Backward (dS). The backward node `dS` implements the local mapping

$$\mathrm{d}\mathbf{z} = \mathrm{dS}(\mathbf{z}, \mathrm{d}\mathbf{p}),$$

which, in Jacobian form, is

$$\mathrm{dS}(\mathbf{z}, \mathrm{d}\mathbf{p}) = \mathbf{J}_{\mathrm{softmax}}(\mathbf{z})^T \mathrm{d}\mathbf{p}.$$

In practice this is computed using the standard softmax backward formula. In the diagrams, the `dS` node has incoming edges carrying $\mathbf{p}$ (or $\mathbf{z}$) and $\mathrm{d}\mathbf{p}$, and an outgoing edge carrying $\mathrm{d}\mathbf{z}$.

4.4.5 Nonlinearities and Dropout (GL, dGL, DO, dDO)

Rectangular nodes labeled `GL`, `GELU`, or similar denote elementwise nonlinearities; nodes labeled `DO` denote dropout.

Forward (GL / DO). For a generic scalar nonlinearity g,

$$\mathbf{Y} = g(\mathbf{X})$$

is applied elementwise. For dropout we write

$$\mathbf{Y} = \mathrm{DO}(\mathbf{X}; \mathbf{m}) = \mathbf{m} \odot \mathbf{X},$$

where $\mathbf{m}$ is a binary mask of the same shape as $\mathbf{X}$ and $\odot$ denotes elementwise multiplication. The mask $\mathbf{m}$ is cached in the forward pass.

Backward (dGL / dDO). For a generic nonlinearity, the backward node `dGL` implements

$$\mathrm{d}\mathbf{X} = \mathrm{dGL}(\mathbf{X}, \mathrm{d}\mathbf{Y}) = \mathrm{d}\mathbf{Y} \odot g'(\mathbf{X}),$$

using the forward input $\mathbf{X}$ from the cache.

For dropout, the backward node `dDO` implements

$$\mathrm{d}\mathbf{X} = \mathrm{dDO}(\mathbf{X}, \mathrm{d}\mathbf{Y}) = \mathbf{m} \odot \mathrm{d}\mathbf{Y}.$$

In the diagrams, the node labeled `dDO` consumes the upstream gradient $\mathrm{d}\mathbf{Y}$ and the cached mask $\mathbf{m}$, and produces $\mathrm{d}\mathbf{X}$.

4.4.6 Layer Normalization (LN, dLN)

Layer normalization nodes are labeled `LN` in the forward pass and `dLN` in the backward pass.

Forward (LN). Given $\mathbf{X} \in [B, S, D]$, layer normalization computes

$$\mathbf{H} = \mathrm{LN}(\mathbf{X}; \gamma, \beta),$$

by normalizing each $[D]$-dimensional vector at fixed B, S, and applying learned scale and shift parameters $\gamma, \beta \in [D]$. The forward pass caches per-position mean/variance as well as γ and β.

Backward (dLN). The backward node `dLN` implements

$$(\mathrm{d}\mathbf{X}, \mathrm{d}\gamma, \mathrm{d}\beta) = \mathrm{dLN}(\mathbf{X}, \gamma, \beta, \mathrm{d}\mathbf{H}, \mathrm{stats}),$$

where stats denotes the cached means and variances. The explicit formulas follow from the standard layer-normalization backward derivation; in the diagrams we treat `dLN` as a single node that consumes $\mathbf{X}$, γ, the cached statistics, and $\mathrm{d}\mathbf{H}$, and produces three outgoing gradient edges $\mathrm{d}\mathbf{X}$, $\mathrm{d}\gamma$, and $\mathrm{d}\beta$.

4.4.7 Communication Nodes (AR, AG)

In tensor-parallel (TP), data-parallel (DP), and hybrid DP+TP settings, collective communication operations synchronize tensors across devices.

All-Reduce (AR). An All-Reduce node `AR` takes as input a tensor shard from each participant and outputs the elementwise reduced tensor (typically a sum), optionally divided by the number of participants for averaging. For example, in DP, weight gradients $\mathrm{d}\mathbf{W}$ of shape $[D, D]$ are All-Reduced across all data-parallel ranks before the optimizer step.

All-Gather (AG). An All-Gather node `AG` concatenates or aggregates tensor shards across a parallel group to reconstruct a full tensor. For example, in TP, partial outputs along a hidden dimension may be gathered to form a full $[B, S, D]$ tensor.

In the diagrams, these communication nodes are treated as pure operators on tensors; their backward behavior (e.g. gradient flow through `AR` and `AG`) is implicit in the symmetry of the operations.

4.5 Reading the Detailed MHA and MLP Figures

With the conventions above, the large MHA and MLP forward/backward figures can be read as follows:

- **Edges** indicate the flow of tensors (activations or gradients), annotated with shapes like $[B, S, D]$ or $[B, N_H, S, D_h]$.
- **Forward nodes** (`S`, `SM`, `DO`, `LN`, `matmul`, `reshape`, `transpose`) compute local functions $f(\mathbf{x}_1, \ldots, \mathbf{x}_k)$.
- **Backward nodes** (`dS`, `dSM`, `dDO`, `dLN`, `dMatmul`) implement the corresponding backward operators $\mathrm{d}f(\mathbf{x}_1, \ldots, \mathbf{x}_k, \mathrm{d}\mathbf{y})$, producing gradients for all inputs.
- **Broadcast labels** such as $\mathrm{BC}_{B,S}(\mathbf{b}_0)$ indicate that a bias vector is conceptually expanded to match a higher-rank tensor before addition.
- **Communication nodes** (`AR`, `AG`) indicate where collective operations occur in TP, DP, or DP+TP settings, and their shape annotations show the logical dimensions involved.

These conventions allow the reader to understand the data and gradient flows at a glance, without being distracted by low-level indexing, while still being precise enough to derive the underlying equations when needed.

5 Single-Node Transformer: Forward and Backward

This section presents the full Transformer layer running on a single node (no parallelism). We emphasize tensor shapes and data flow for both forward and backward passes. Each subsection covers one major component:

- **Input Embedding**: Converting token IDs to dense vectors.
- **Multi-Head Attention (MHA)**: Computing attention over the sequence.
- **Feed-Forward Network (MLP/FFN)**: Position-wise non-linear processing.
- **Output Projection and Loss**: Generating predictions and computing loss.

We assume a batch of tokenized sequences with shape $[B, S]$, where B is batch size and S is sequence length. The hidden representations inside the layer typically have shape $[B, S, D]$, where D is the model (hidden) dimension. In the backward pass we track how gradients with respect to the scalar loss $\mathcal{L}$ flow backward through these tensors and the corresponding weight matrices.

5.1 Overall Transformer Layer Flow

Before diving into individual components, we provide a high-level view of how data flows through the entire Transformer layer in both forward and backward passes. Figure 1 summarizes the forward path from input embeddings through MHA, MLP, and output projection, and the corresponding backward paths from the loss back to the embeddings.

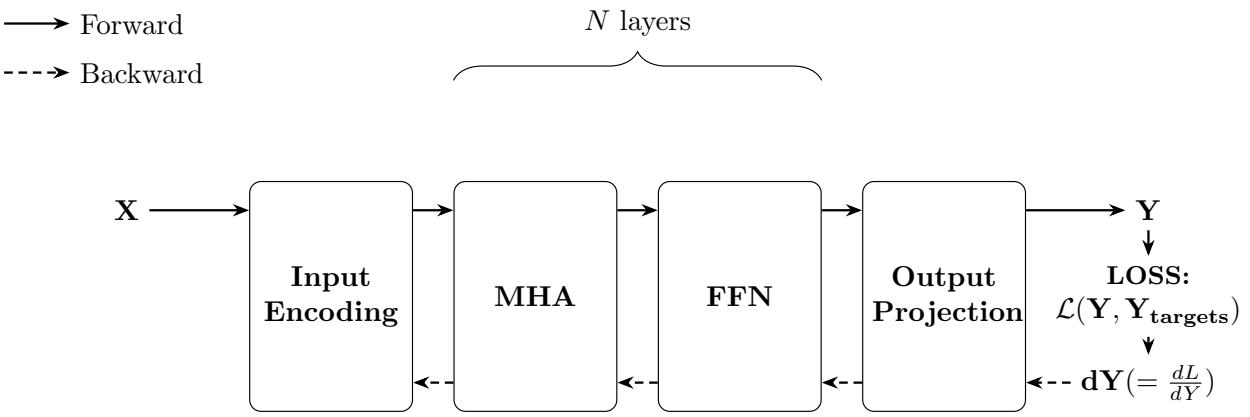


Figure 1: Single-node Transformer layer: overall forward and backward flow. Solid arrows denote forward activations, and dashed arrows denote gradients flowing backward from the loss $\mathcal{L}$ through the output projection, MLP, MHA, and back to the input embeddings.

5.2 Input Embedding Layer

The input embedding layer converts discrete token IDs into continuous vector representations, producing the initial hidden states $\mathbf{X} \in \mathbb{R}^{B \times S \times D}$ that enter the Transformer stack. We assume:

- A token embedding matrix $\mathbf{E} \in \mathbb{R}^{V \times D}$, where V is the vocabulary size.
- A positional embedding table $\mathbf{P} \in \mathbb{R}^{S \times D}$, either learned or fixed (e.g., sinusoidal).

Given token IDs $\mathbf{T} \in [B, S]$, the embedding layer performs:

1. **Token embedding lookup**: each token ID indexes into $\mathbf{E}$ to form $\mathbf{X}_{\text{tok}} \in [B, S, D]$.

2. **Positional embedding addition**: positional vectors $\mathbf{P}[s, :]$ are added to each time step s.

3. **Dropout (optional)**: a dropout layer may be applied to the resulting embeddings for regularization during training.

The resulting hidden states $\mathbf{X}$, with shape $[B, S, D]$, appear on the left of Figure 1 as the starting point of the layer, and the detailed dataflow inside the embedding block is shown in Figure 2.

5.2.1 Forward Pass

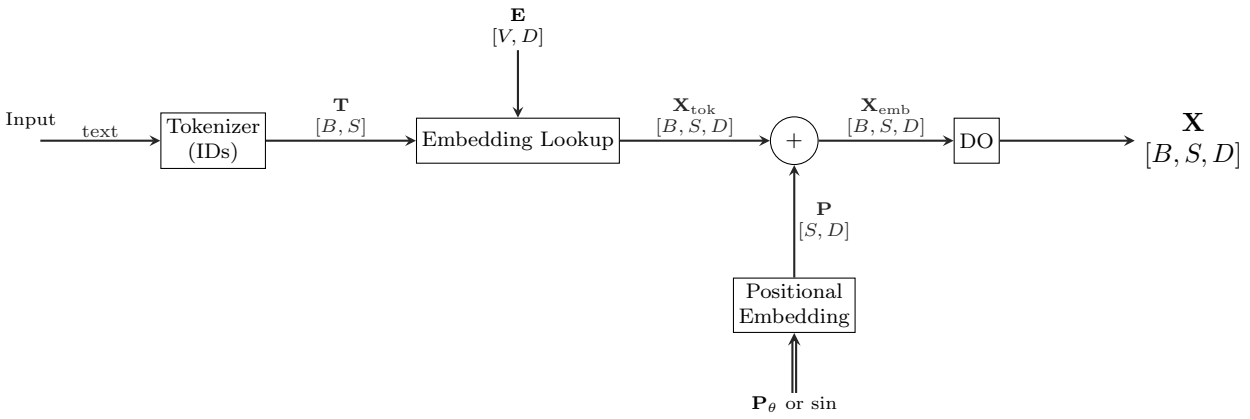


Figure 2: Input embedding forward pass. Token IDs are mapped to token embeddings using $\mathbf{E}$, positional embeddings $\mathbf{P}$ are added, and optional dropout yields the initial hidden states $\mathbf{X} \in [B, S, D]$.

In the backward pass (not shown as a separate figure), gradients with respect to the embeddings are accumulated into the token embedding matrix $\mathbf{E}$ and positional table $\mathbf{P}$ by summing over all positions where each embedding is used.

5.3 Multi-Head Attention (MHA)

Multi-Head Attention enables the model to jointly attend to information from different representation subspaces. The mechanism involves:

- **Linear projections**: mapping inputs to queries (Q), keys (K), and values (V).
- **Scaled dot-product attention**: computing attention scores and weighted sums of values.
- **Multi-head splitting**: processing N_H attention heads in parallel, each with dimension D_h such that $D = N_H D_h$.
- **Output projection**: concatenating heads and projecting back to the model dimension.

Given input activations $\mathbf{X} \in [B, S, D]$, the forward pass uses layer normalization, QKV projections, attention over the sequence, and a residual connection back to $\mathbf{X}$. The overall structure of this computation is visualized in Figure 3, while the corresponding gradient flow during training is shown in Figure 4.

5.3.1 Forward Pass

Given input activations $\mathbf{X} \in \mathbb{R}^{B \times S \times D}$, the multi-head self-attention block computes an output $\mathbf{A}_{\text{out}} \in \mathbb{R}^{B \times S \times D}$ that will be fed into the following MLP block. The same tensor shapes and operations are annotated along the edges of Figure 3 for quick visual reference.

(1) Layer normalization on the input. We first normalize the input along the hidden dimension:

$$\mathbf{X}_{\text{norm}} = \text{LN}(\mathbf{X}) \in \mathbb{R}^{B\times S\times D}.$$

Layer normalization is parameterized by learnable scale and bias vectors $\boldsymbol{\gamma}, \boldsymbol{\beta} \in \mathbb{R}^D$, but for brevity we only track the normalized activations here. In Figure 3, this corresponds to the leftmost normalization block before the Q/K/V projections.

(2) Linear projections to Q, K, V. From $\mathbf{X}_{\text{norm}}$ we compute queries, keys, and values via three independent linear layers:

$$\mathbf{Q} = \mathbf{X}_{\text{norm}} W_Q, \quad \mathbf{K} = \mathbf{X}_{\text{norm}} W_K, \quad \mathbf{V} = \mathbf{X}_{\text{norm}} W_V,$$

where $W_Q, W_K, W_V \in \mathbb{R}^{D\times D}$. Each of these tensors is then reshaped into a head-major representation,

$$\mathbf{Q}, \mathbf{K}, \mathbf{V} \ \in \ \mathbb{R}^{B\times N_H\times S\times D_h}, \qquad D = N_H \cdot D_h.$$

These projections and reshapes are drawn explicitly in the upper left of Figure 3, where each head receives its own D_h-dimensional slice.

(3) Scaled dot-product attention per head. For each head, we form attention scores by a scaled dot-product between queries and keys:

$$\mathbf{S} = \frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{D_h}} \ \in \ \mathbb{R}^{B\times N_H\times S\times S},$$

where the last two dimensions correspond to (query position, key position). A causal mask or padding mask is applied to $\mathbf{S}$ to prevent attending to invalid positions. After masking, a softmax along the last dimension produces attention weights:

$$\mathbf{A}_S = \text{Softmax}(\text{Mask}(\mathbf{S})) \ \in \ \mathbb{R}^{B\times N_H\times S\times S}.$$

These weights are then used to aggregate values:

$$\mathbf{A}_{\text{heads}} = \mathbf{A}_S \mathbf{V} \ \in \ \mathbb{R}^{B\times N_H\times S\times D_h},$$

as depicted in the central part of Figure 3, where each head computes a weighted sum over the value vectors.

(4) Head concatenation and output projection. The per-head outputs are concatenated along the head dimension and reshaped back to the model dimension:

$$\mathbf{A}_{\text{cat}} = \text{ConcatHeads}(\mathbf{A}_{\text{heads}}) \ \in \ \mathbb{R}^{B\times S\times D}.$$

A final output projection maps this back into the same hidden space:

$$\mathbf{A}_{\text{lin}} = \mathbf{A}_{\text{cat}} W_O + \mathbf{b}_O, \qquad W_O \in \mathbb{R}^{D\times D}, \ \mathbf{b}_O \in \mathbb{R}^D.$$

This corresponds to the right side of Figure 3, where all heads are merged and projected back to $[B, S, D]$.

(5) Dropout and residual connection. During training, dropout may be applied to $\mathbf{A}_{\text{lin}}$:

$$\mathbf{A}_{\text{drop}} = \text{Dropout}(\mathbf{A}_{\text{lin}}),$$

and the result is added back to the original input via a residual connection:

$$\mathbf{A}_{\text{out}} = \mathbf{X} + \mathbf{A}_{\text{drop}} \ \in \ \mathbb{R}^{B\times S\times D}.$$

The final sum node and residual edge are explicitly shown on the far right of Figure 3. This $\mathbf{A}_{\text{out}}$ becomes the input to the subsequent MLP block.

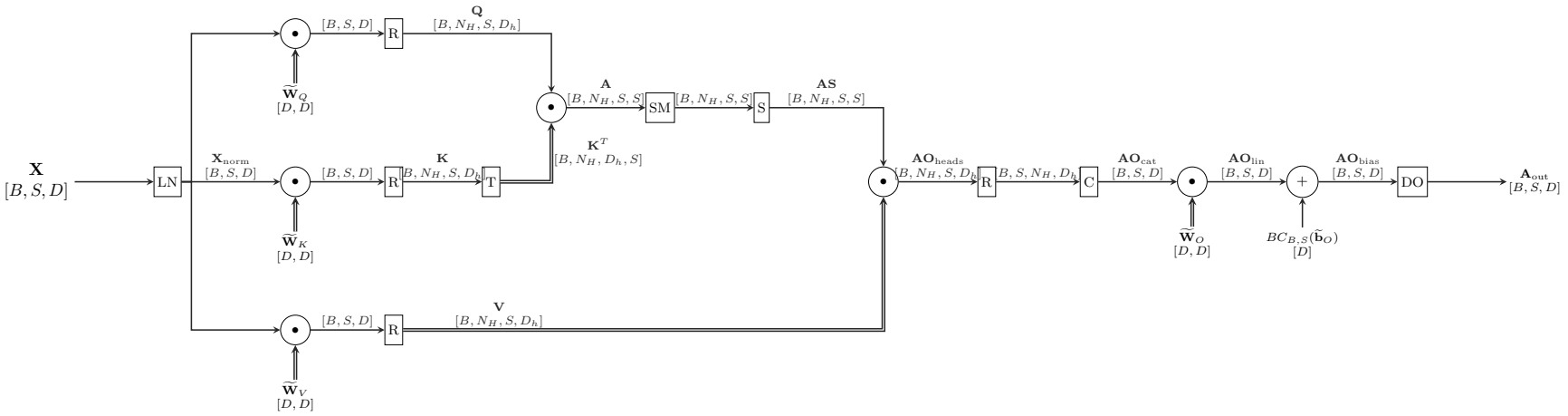


Figure 3: Multi-head attention forward pass on a single node. Input activations $\mathbf{X}$ are layer-normalized, projected to $\mathbf{Q}$, $\mathbf{K}$, and $\mathbf{V}$, processed by scaled dot-product attention over the sequence, concatenated across heads, and projected back to $[B, S, D]$ with an output projection and residual connection.

5.3.2 Backward Pass

In the backward pass, gradients arrive from the residual output and propagate through the output projection, attention mechanism, QKV projections, and layer normalization. Figure 4 mirrors Figure 3 but with dashed arrows showing gradient flow and explicit nodes for each matrix multiplication in the backward path.

At a high level:

- Gradients from the loss provide $\mathrm{d}\mathbf{A}_{\text{out}}$ at the output of the attention block, which are split between the residual branch and the main path, as shown on the right of Figure 4.
- The output projection is backpropagated to obtain gradients for weights and biases and to recover $\mathrm{d}\mathbf{A}_{\text{cat}}$ before head concatenation.
- The per-head attention computation is differentiated to obtain $\mathrm{d}\mathbf{V}$ and $\mathrm{d}\mathbf{A}_S$, and then $\mathrm{d}\mathbf{Q}$ and $\mathrm{d}\mathbf{K}$ through the scaled dot-product and softmax, corresponding to the central region of Figure 4.
- QKV projection layers accumulate gradients for $\mathbf{W}_Q, \mathbf{W}_K, \mathbf{W}_V$ and produce $\mathrm{d}\mathbf{X}_{\text{norm}}$, shown on the left side of Figure 4.
- Layer normalization backward finally produces $\mathrm{d}\mathbf{X}$ that flows to the previous block and parameter gradients for the LN parameters.

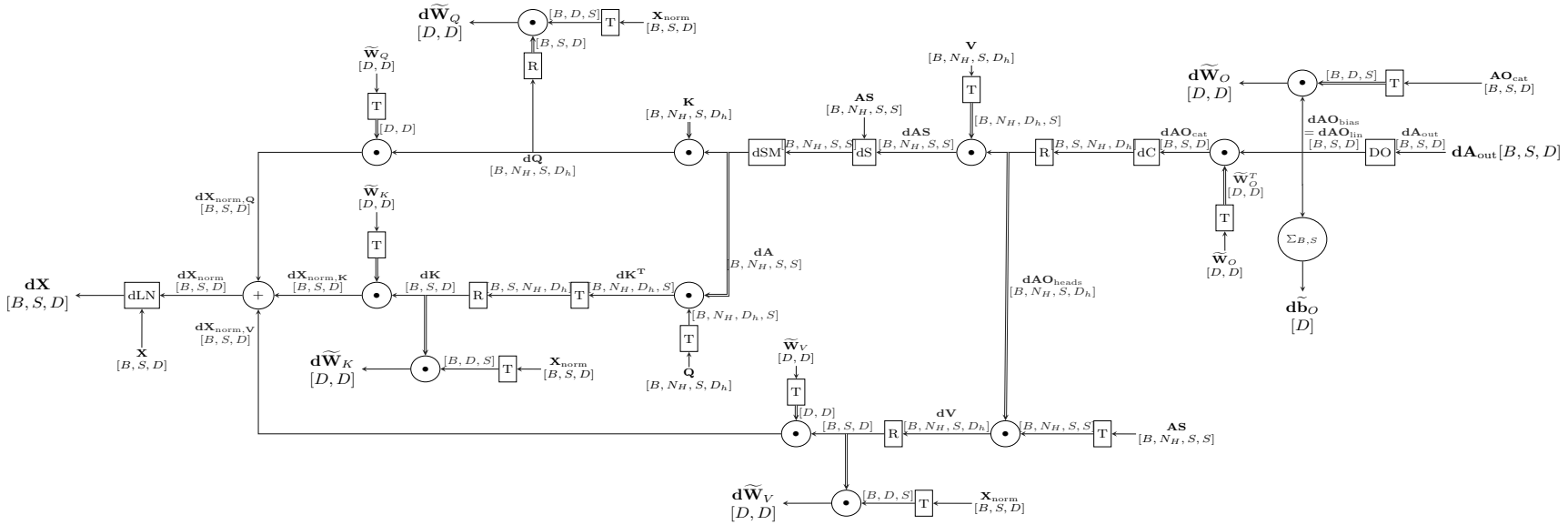


Figure 4: Multi-head attention backward pass. The diagram shows how gradients from $\mathrm{d}\mathbf{A}_{\text{out}}$ are propagated through the output projection, attention heads, QKV projection matrices, and layer normalization to yield $\mathrm{d}\mathbf{X}$ and all associated weight and bias gradients.

5.4 Feed-Forward Network (MLP / FFN)

The feed-forward network (FFN) applies two linear transformations with a non-linear activation in between, independently at each position in the sequence. Starting from $\mathbf{X}' \in [B, S, D]$ (the output of MHA with residual), the FFN performs:

- **Up-projection**: $[B, S, D] \to [B, S, D_{\text{ff}}]$ (expanding the dimension).
- **Activation**: GELU or ReLU non-linearity applied elementwise.
- **Down-projection**: $[B, S, D_{\text{ff}}] \to [B, S, D]$ (projecting back).
- **Dropout and residual**: dropout on the FFN output followed by a residual connection back to $\mathbf{X}'$.

The forward structure of this block is illustrated in Figure 5, and the corresponding backward matmuls are expanded in Figure 6.

5.4.1 Forward Pass

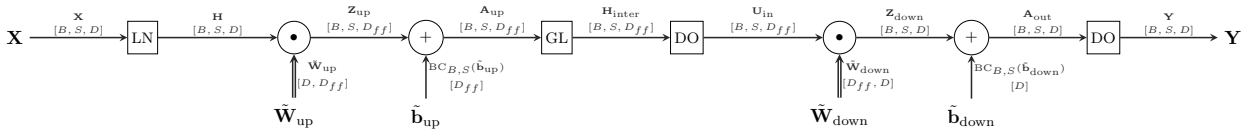


Figure 5: Feed-forward (MLP) forward pass. Hidden states are layer-normalized, projected up to dimension D_{ff}, passed through a non-linearity and dropout, projected back to D, and combined with the input via a residual connection.

5.4.2 Backward Pass

The MLP backward pass is a canonical example of the "2× cost" rule: each forward matrix multiplication generates two backward matmuls (one for activations, one for weights). Starting from $\mathrm{d}\mathbf{Y}$ at the FFN output, the backward pass proceeds as:

1. **Residual and dropout backward**: split $\mathrm{d}\mathbf{Y}$ into the identity path and the path through the final dropout to obtain $\mathrm{d}\mathbf{Z}_{\text{down}}$.
2. **Down-projection backward**: compute $\mathrm{d}\mathbf{U}$ (w.r.t. activations) and obtain gradients for $\mathbf{W}_{\text{down}}$ and $\mathbf{b}_{\text{down}}$.
3. **Activation and dropout backward**: backpropagate through dropout and the non-linearity (e.g., GELU) to obtain $\mathrm{d}\mathbf{Z}_{\text{up}}$.
4. **Up-projection backward**: compute $\mathrm{d}\mathbf{H}$ and gradients for $\mathbf{W}_{\text{up}}$ and $\mathbf{b}_{\text{up}}$.
5. **Layer normalization backward**: propagate $\mathrm{d}\mathbf{H}$ through layer normalization to produce $\mathrm{d}\mathbf{X}'$ (which flows into the MHA block) and gradients for LN parameters.

As with MHA, every matmul in the forward path corresponds to two matmuls in the backward path, which are shown explicitly in Figure 6.

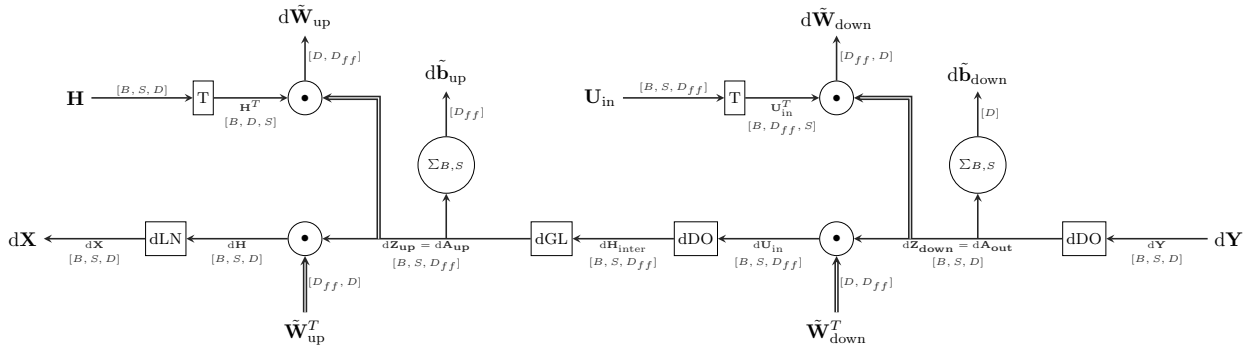


Figure 6: Feed-forward (MLP) backward pass. The figure shows how $d\mathbf{Y}$ splits through the residual and FFN path, and how gradients flow through the down-projection, activation, up-projection, and layer normalization to yield $d\mathbf{X}'$ and the corresponding parameter gradients.

5.5 Output Projection and Loss

The final stage converts the Transformer's hidden representations into predictions over the vocabulary and computes the training loss. We assume the last layer output is $\mathbf{A}_{\text{out}} \in [B, S, D]$, and the training targets are token IDs $\mathbf{Y}_{\text{targets}} \in [B, S]$.

- **Linear projection**: $[B, S, D] \to [B, S, V]$ to produce logits.
- **Softmax**: converting logits into probability distributions.
- **Cross-Entropy Loss**: comparing probabilities with target tokens to produce a scalar loss $\mathcal{L}$.

The forward and backward views of this stage are depicted in Figures 7 and 8, respectively.

5.5.1 Forward Pass (Logits, Softmax, Loss)

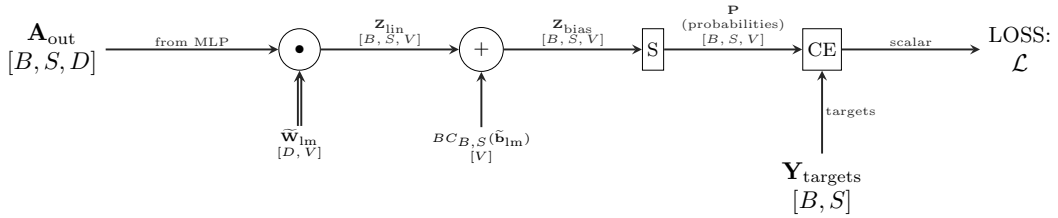


Figure 7: Output projection and loss: the final hidden states $\mathbf{A}_{\text{out}}$ are mapped to logits by the language model head $(\mathbf{W}_{\text{lm}}, \mathbf{b}_{\text{lm}})$, converted to probabilities by softmax, and compared with target tokens using cross-entropy.

5.5.2 Backward Pass

The backward pass starts from $d\mathcal{L} = 1$ and propagates gradients through the loss, softmax, and linear projection:

1. **Cross-entropy + softmax backward**: for each position, softmax and cross-entropy combine to give $d\mathbf{Z} = \mathbf{P} - \mathbf{Y}_{\text{onehot}}$, where $\mathbf{P}$ is the softmax output and $\mathbf{Y}_{\text{onehot}}$ is the one-hot encoding of the target token.
2. **Bias and weight gradients**: accumulate $d\mathbf{b}_{\text{lm}} = \sum_{b,s} d\mathbf{Z}[b, s, :]$ and compute $d\mathbf{W}_{\text{lm}} = \mathbf{A}_{\text{out}}^{\top} d\mathbf{Z}$.

3. **Gradient to hidden states**: propagate gradients back to the Transformer by $\mathrm{d}\mathbf{A}_{\text{out}} = \mathrm{d}\mathbf{Z}\,\mathbf{W}_{\text{lm}}^{\top}$.

These steps correspond directly to the blocks and arrows in Figure 8, where the softmax and cross-entropy are fused into a single gradient expression with respect to the logits.

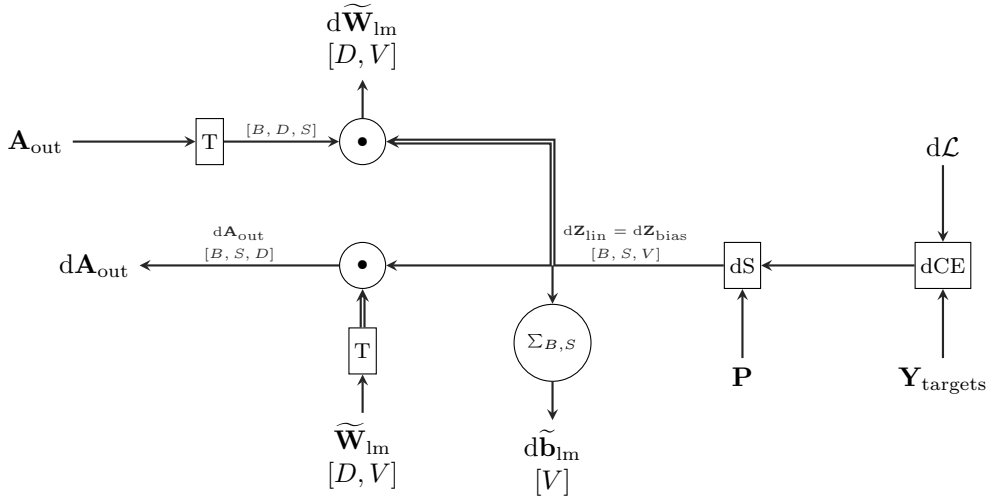


Figure 8: Output projection backward pass. Gradients from the loss are propagated through cross-entropy and softmax to the logits, then through the language model projection to produce $\mathrm{d}\mathbf{A}_{\text{out}}$ and parameter gradients $\mathrm{d}\mathbf{W}_{\text{lm}}$ and $\mathrm{d}\mathbf{b}_{\text{lm}}$.

6 Tensor Parallelism (TP)

In tensor parallelism, the parameters of each layer are partitioned across multiple devices along one or more tensor dimensions. Instead of replicating the full weight matrix on every device, each device holds only a shard of the weights and computes on a corresponding shard of the activations. Collective communication (e.g., All-Reduce, All-Gather) is then used to assemble partial results into the same shapes as in the single-node model of Section 5.

We denote the tensor-parallel degree by N_T, and index devices by $t \in \{0, \dots, N_T - 1\}$. Throughout this section we focus on how the single-node computations from Section 5 are decomposed across these N_T devices.

Key ideas.

- Large weight matrices are split along rows or columns so that each device holds a sub-matrix $W^{(t)}$ instead of the full matrix W.
- Each device computes local partial outputs using its own shard.
- Collective operations (All-Reduce, All-Gather) combine local partial results across devices into the same tensors that appear in the single-node computation graph.
- The backward pass mirrors the sharding and communication pattern of the forward pass so that gradients with respect to parameters and inputs are correctly accumulated.
- Memory usage per device is reduced approximately by a factor of N_T, but each global matmul incurs at least one collective communication.

6.1 Overview of Tensor-Parallel Sharding

At a high level, tensor parallelism can be seen as replacing every large matrix multiplication in Section 5 with a collection of smaller matmuls that run in parallel on different devices. There are two basic patterns for sharding a linear layer

$$\mathbf{Y} = \mathbf{X}W + \mathbf{b}, \qquad \mathbf{X} \in \mathbb{R}^{B \times D_{\text{in}}},\ W \in \mathbb{R}^{D_{\text{in}} \times D_{\text{out}}}:$$

- **Column-parallel (output-sharded) linear**: split W along the output dimension, $W = [W^{(0)}, \dots, W^{(N_T-1)}]$ with $W^{(t)} \in \mathbb{R}^{D_{\text{in}} \times D_{\text{out}}^{(t)}}$. Each device computes $\mathbf{Y}^{(t)} = \mathbf{X}W^{(t)} + \mathbf{b}^{(t)}$, and the global output is obtained by concatenation:
$$\mathbf{Y} = \text{Concat}_t \mathbf{Y}^{(t)}.$$
No collective is needed on the forward path, but later layers may require an All-Gather or All-Reduce to reassemble or sum over shards.
- **Row-parallel (input-sharded) linear**: split W along the input dimension and shard $\mathbf{X}$ accordingly. Each device holds $X^{(t)}$ and $W^{(t)}$ with $W^{(t)} \in \mathbb{R}^{D_{\text{in}}^{(t)} \times D_{\text{out}}}$, computes a partial product $\mathbf{Y}^{(t)} = \mathbf{X}^{(t)}W^{(t)}$, and an All-Reduce across t produces the full output:
$$\mathbf{Y} = \sum_{t=0}^{N_T-1} \mathbf{Y}^{(t)}.$$

These two patterns are composed inside the Transformer so that most operations remain local to each device and only a small number of All-Reduce/All-Gather calls are required per layer. An overview of this scheme for a single Transformer block is shown in Figure 9, which should be compared to the single-node overview in Figure 1.

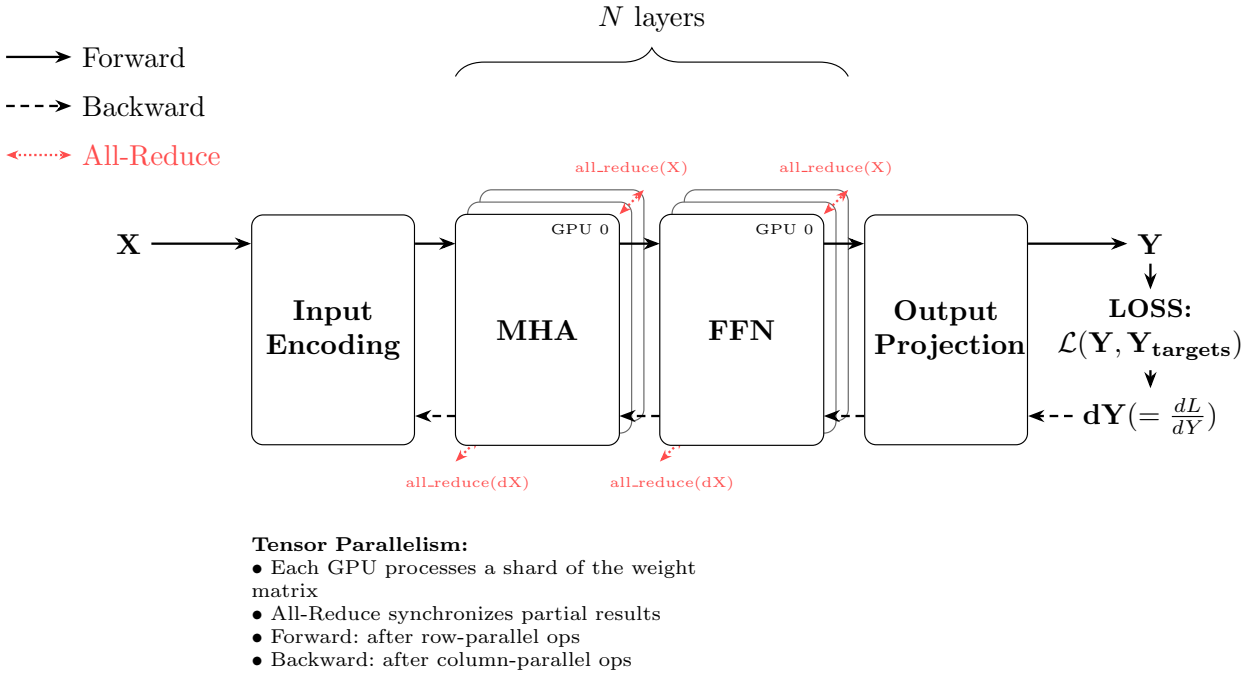


Figure 9: Overall Transformer layer with tensor parallelism. Each large linear layer from the single-node model is replaced by N_T smaller matmuls on different devices. Colored arrows indicate where collective communication (e.g. All-Reduce, All-Gather) is required to assemble partial results, while local computations remain identical to those in Figure 1.

6.2 MHA with Tensor Parallelism

We now apply tensor parallelism to the multi-head attention (MHA) block. Recall from Section 5.2 that the single-node attention block maps $\mathbf{X} \in [B, S, D]$ to $\mathbf{A}_{\text{out}} \in [B, S, D]$ via Q/K/V projections, scaled dot product attention, an output projection, and a residual connection.

In tensor parallelism we partition these computations across N_T devices in such a way that:

- each device is responsible for a subset of the attention heads, or equivalently a subset of the output channels of the Q/K/V projections;
- the softmax and value-weighted sums are computed locally on each device for its own heads;
- the final output projection is implemented as a row-parallel linear, requiring an All-Reduce across devices to recover the same $\mathbf{A}_{\text{out}}$ as in the single-node case.

Figure 10 shows the forward data flow under this sharding scheme and should be read as a "TP-annotated" version of Figure 3. The corresponding backward graph is shown in Figure 11.

6.2.1 Forward Pass

For clarity we focus on one Transformer block and omit batch/sequence dimensions in some formulas. We write the per-device Q/K/V projections as column-parallel linears:

$$W_Q = [W_Q^{(0)}, \dots, W_Q^{(N_T-1)}], \quad W_K = [W_K^{(0)}, \dots, W_K^{(N_T-1)}], \quad W_V = [W_V^{(0)}, \dots, W_V^{(N_T-1)}],$$

where, on device t,

$$W_Q^{(t)}, W_K^{(t)}, W_V^{(t)} \in \mathbb{R}^{D \times D_h^{(t)}}, \qquad \sum_t D_h^{(t)} = D.$$

Each device computes local projections:

$$\mathbf{Q}^{(t)} = \mathbf{X}_{\text{norm}} W_Q^{(t)}, \quad \mathbf{K}^{(t)} = \mathbf{X}_{\text{norm}} W_K^{(t)}, \quad \mathbf{V}^{(t)} = \mathbf{X}_{\text{norm}} W_V^{(t)},$$

and reshapes them into its own subset of attention heads. Because heads are independent, each device can compute the scaled dot-product attention for its local heads without communication:

$$\mathbf{S}^{(t)} = \frac{\mathbf{Q}^{(t)} (\mathbf{K}^{(t)})^\top}{\sqrt{D_h^{(t)}}}, \qquad \mathbf{A}_S^{(t)} = \text{Softmax}(\text{Mask}(\mathbf{S}^{(t)})),$$

$$\mathbf{A}_{\text{heads}}^{(t)} = \mathbf{A}_S^{(t)} \mathbf{V}^{(t)}.$$

After computing local head outputs, each device forms a local concatenation $\mathbf{A}_{\text{cat}}^{(t)} \in \mathbb{R}^{B \times S \times D^{(t)}}$ over its subset of heads. The final output projection is implemented as a *row-parallel* linear:

$$W_O = \begin{bmatrix} W_O^{(0)} \\ \vdots \\ W_O^{(N_T-1)} \end{bmatrix}, \quad W_O^{(t)} \in \mathbb{R}^{D^{(t)} \times D}.$$

Each device computes a partial contribution

$$\mathbf{A}_{\text{lin}}^{(t)} = \mathbf{A}_{\text{cat}}^{(t)} W_O^{(t)} + \mathbf{b}_O^{(t)},$$

and an All-Reduce across t yields the full projected tensor

$$\mathbf{A}_{\text{lin}} = \sum_{t=0}^{N_T-1} \mathbf{A}_{\text{lin}}^{(t)}.$$

Dropout and the residual connection with $\mathbf{X}$ are then applied locally on each device, since after the All-Reduce every device holds the same $\mathbf{A}_{\text{lin}}$ and $\mathbf{X}$.

These steps, together with the location of the All-Reduce, are annotated explicitly in Figure 10.

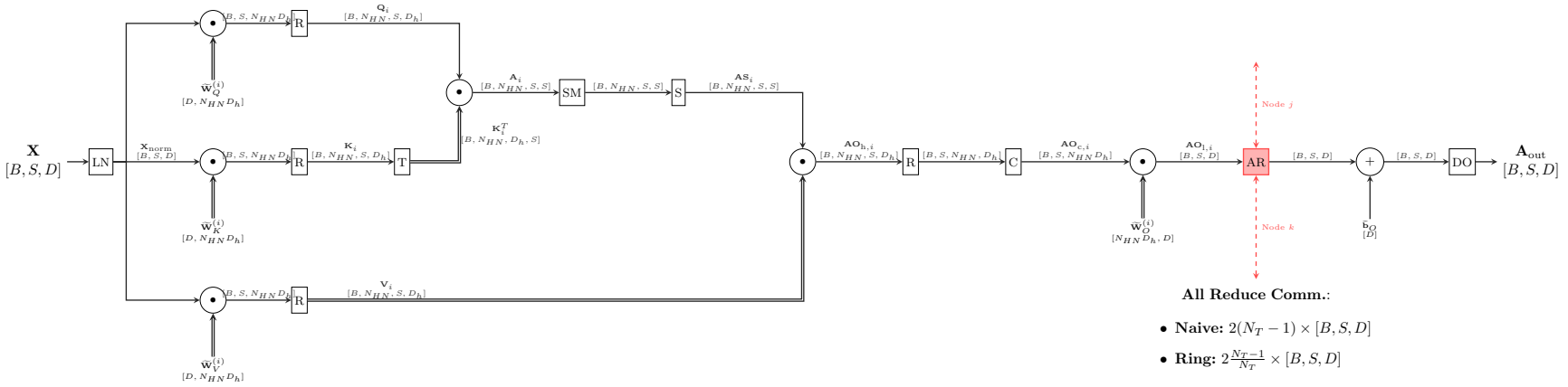


Figure 10: Multi-head attention forward pass with tensor parallelism. Q/K/V projections are implemented as column-parallel linears so that each device owns a subset of the heads. Attention for each local head is computed independently, and the resulting head outputs are combined using a row-parallel output projection followed by an All-Reduce to recover the same $\mathbf{A}_{\text{out}}$ as in the single-node computation.

6.2.2 Backward Pass

The backward pass for tensor-parallel MHA follows the same high-level structure as in Section 5.2, but with sharded gradients and explicit collectives. Starting from $\mathrm{d}\mathbf{A}_{\text{out}}$ on each device, we have:

- **Output projection backward.** The row-parallel output projection produces local gradients $\mathrm{d}\mathbf{A}_{\text{cat}}^{(t)}$ and parameter gradients $\mathrm{d}W_O^{(t)}, \mathrm{d}\mathbf{b}_O^{(t)}$ from $\mathrm{d}\mathbf{A}_{\text{lin}}$.
- **Head backward.** Each device backpropagates through its local heads, computing $\mathrm{d}\mathbf{V}^{(t)}$ and $\mathrm{d}\mathbf{A}_S^{(t)}$, and then $\mathrm{d}\mathbf{Q}^{(t)}$ and $\mathrm{d}\mathbf{K}^{(t)}$ through the scaled dot-product and softmax.
- **Q/K/V projection backward.** The column-parallel Q/K/V linears produce local gradients $\mathrm{d}W_Q^{(t)}, \mathrm{d}W_K^{(t)}, \mathrm{d}W_V^{(t)}$ and partial contributions to the gradient w.r.t. the normalized input, $\mathrm{d}\mathbf{X}_{\text{norm}}^{(t)}$.
- **All-Reduce for input gradient.** Because $\mathbf{X}_{\text{norm}}$ is shared across devices, gradients from all Q/K/V shards must be summed:
$$\mathrm{d}\mathbf{X}_{\text{norm}} = \sum_{t=0}^{N_T-1} \mathrm{d}\mathbf{X}_{\text{norm}}^{(t)},$$
implemented as an All-Reduce over t.
- **Layer normalization backward.** Finally, layer normalization backward is applied locally on each device to produce $\mathrm{d}\mathbf{X}$ and gradients for the LN parameters, exactly as in the single-node setting.

Figure 11 expands these steps into a full computation graph, with each All-Reduce/All-Gather explicitly indicated.

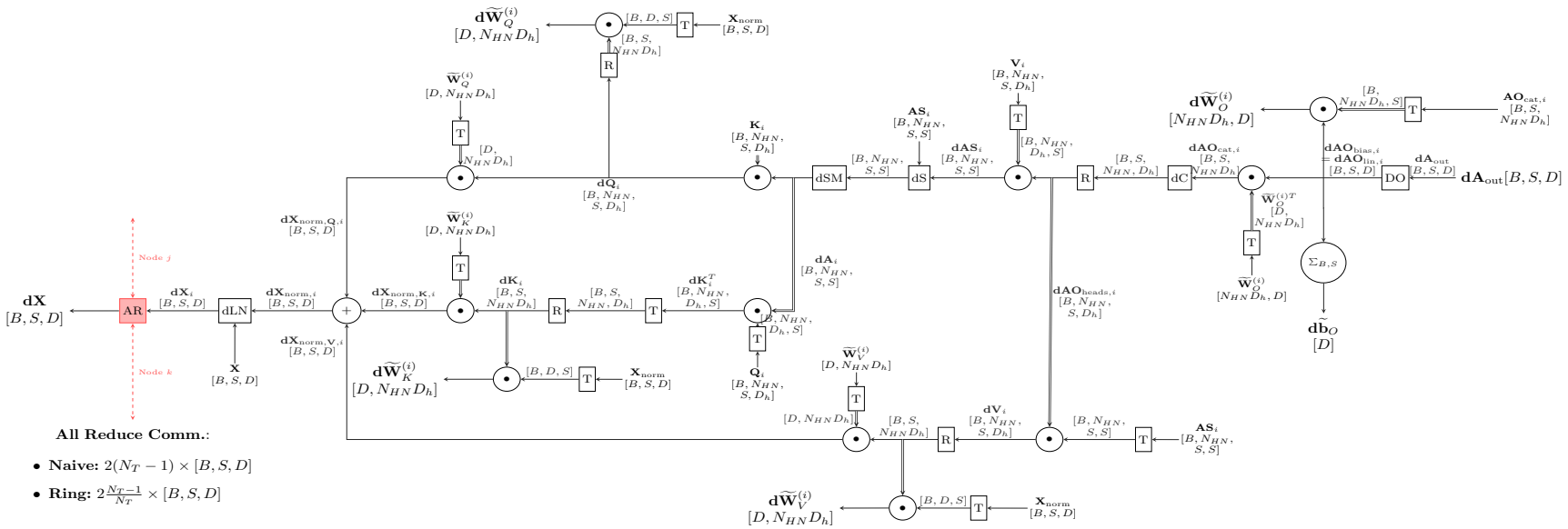


Figure 11: Multi-head attention backward pass with tensor parallelism. Each device backpropagates through its local Q/K/V projections and attention heads. Gradients with respect to the shared normalized input are summed across devices via All-Reduce, and parameter gradients are accumulated locally for each shard $W_Q^{(t)}, W_K^{(t)}, W_V^{(t)}$ and $W_O^{(t)}$.

6.3 MLP with Tensor Parallelism

The feed-forward (MLP) block is particularly well-suited to tensor parallelism because its two linear layers can be implemented as a column-parallel / row-parallel pair. Recall from Section 5.3 that the single-node MLP maps $\mathbf{H} \in [B, S, D]$ to $\mathbf{Y} \in [B, S, D]$ via

$$\mathbf{Z}_{\text{up}} = \mathbf{H}W_{\text{up}} + \mathbf{b}_{\text{up}}, \quad \mathbf{U} = \phi(\mathbf{Z}_{\text{up}}),$$

$$\mathbf{Z}_{\text{down}} = \mathbf{U}W_{\text{down}} + \mathbf{b}_{\text{down}}, \quad \mathbf{Y} = \mathbf{H} + \text{Dropout}(\mathbf{Z}_{\text{down}}),$$

where $W_{\text{up}} \in \mathbb{R}^{D \times D_{\text{ff}}}$ and $W_{\text{down}} \in \mathbb{R}^{D_{\text{ff}} \times D}$.

In tensor parallelism we choose:

- **Up-projection**: column-parallel (output-sharded).
- **Down-projection**: row-parallel (input-sharded).

This combination ensures that only one collective is required in the forward pass and one in the backward pass, while keeping most of the activation tensors local.

Figure 12 shows the resulting forward data flow, and Figure 13 shows the corresponding backward computation.

6.3.1 Forward Pass

We split the up-projection weight along its output dimension:

$$W_{\text{up}} = [W_{\text{up}}^{(0)}, \ldots, W_{\text{up}}^{(N_T-1)}], \quad W_{\text{up}}^{(t)} \in \mathbb{R}^{D \times D_{\text{ff}}^{(t)}}, \qquad \sum_t D_{\text{ff}}^{(t)} = D_{\text{ff}}.$$

Each device computes a local up-projection:

$$\mathbf{Z}_{\text{up}}^{(t)} = \mathbf{H}W_{\text{up}}^{(t)} + \mathbf{b}_{\text{up}}^{(t)}, \qquad \mathbf{U}^{(t)} = \phi(\mathbf{Z}_{\text{up}}^{(t)}),$$

where ϕ is the non-linearity (e.g., GELU). No communication is required at this stage, because each device only needs its own local slice $\mathbf{U}^{(t)}$ to proceed.

Next, we implement the down-projection as a row-parallel linear:

$$W_{\text{down}} = \begin{bmatrix} W_{\text{down}}^{(0)} \\ \vdots \\ W_{\text{down}}^{(N_T-1)} \end{bmatrix}, \quad W_{\text{down}}^{(t)} \in \mathbb{R}^{D_{\text{ff}}^{(t)} \times D}.$$

Each device computes a partial contribution

$$\mathbf{Z}_{\text{down}}^{(t)} = \mathbf{U}^{(t)}W_{\text{down}}^{(t)} + \mathbf{b}_{\text{down}}^{(t)},$$

and an All-Reduce across t yields the full down-projection output:

$$\mathbf{Z}_{\text{down}} = \sum_{t=0}^{N_T-1} \mathbf{Z}_{\text{down}}^{(t)}.$$

Dropout and the residual connection with $\mathbf{H}$ are then applied locally, as every device has the same $\mathbf{Z}_{\text{down}}$ and $\mathbf{H}$ after the All-Reduce.

These steps are summarized in Figure 12, where the only collective in the forward path is the All-Reduce after the down-projection.

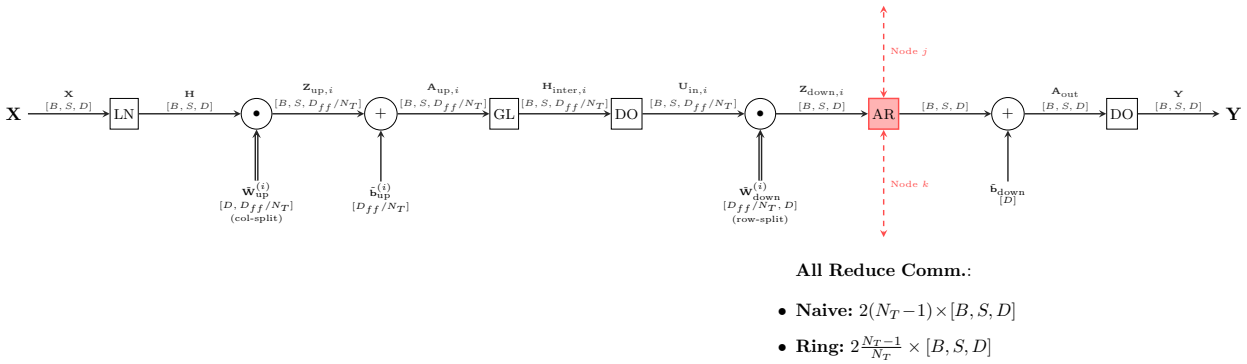


Figure 12: MLP forward pass with tensor parallelism. The up-projection is implemented as a column-parallel linear so that each device holds a subset of the intermediate features. The down-projection is row-parallel, and an All-Reduce over devices reconstructs the full $\mathbf{Z}_{\text{down}}$, after which dropout and the residual connection are applied locally.

6.3.2 Backward Pass

The backward pass for the tensor-parallel MLP reuses the same structure as the single-node backward graph (Figure 6), but with sharded parameters and explicit All-Reduce operations.

Starting from $\mathrm{d}\mathbf{Y}$ on each device:

1. **Residual and dropout backward**: as before, gradients are split between the identity path and the path through the final dropout, yielding $\mathrm{d}\mathbf{Z}_{\text{down}}$ on each device.
2. **Down-projection backward (row-parallel)**: each device computes local gradients $\mathrm{d}\mathbf{U}^{(t)}$ and parameter gradients $\mathrm{d}W_{\text{down}}^{(t)}, \mathrm{d}\mathbf{b}_{\text{down}}^{(t)}$ using its own shard $W_{\text{down}}^{(t)}$ and local activations $\mathbf{U}^{(t)}$. No communication is needed to form $\mathrm{d}\mathbf{U}^{(t)}$.
3. **Activation backward**: the non-linearity ϕ is inverted elementwise on each device to obtain $\mathrm{d}\mathbf{Z}_{\text{up}}^{(t)}$.
4. **Up-projection backward (column-parallel)**: for the column-parallel up-projection, each device computes local contributions to the gradient with respect to $W_{\text{up}}^{(t)}$ and a partial gradient w.r.t. the input $\mathrm{d}\mathbf{H}^{(t)}$. Because $\mathbf{H}$ is shared across devices, we must sum these partial gradients:

$$\mathrm{d}\mathbf{H} = \sum_{t=0}^{N_T-1} \mathrm{d}\mathbf{H}^{(t)},$$

   implemented as an All-Reduce over t.
5. **Layer normalization backward (if present)**: if the MLP is preceded by a layer normalization (as in a pre-LN Transformer), its backward pass is applied locally using the aggregated $\mathrm{d}\mathbf{H}$.

Figure 13 shows these steps, highlighting the All-Reduce on $\mathrm{d}\mathbf{H}$ (the only collective required in the MLP backward pass).

6.4 Communication Patterns and Costs

Finally, we summarize where collective communication appears in the tensor-parallel Transformer layer:

- **MHA forward:** an All-Reduce after the row-parallel output projection to obtain $\mathbf{A}_{\text{lin}}$ on every device (Figure 10).
- **MHA backward:** an All-Reduce on $\mathrm{d}\mathbf{X}_{\text{norm}}$ to sum gradient contributions from all Q/K/V shards (Figure 11).

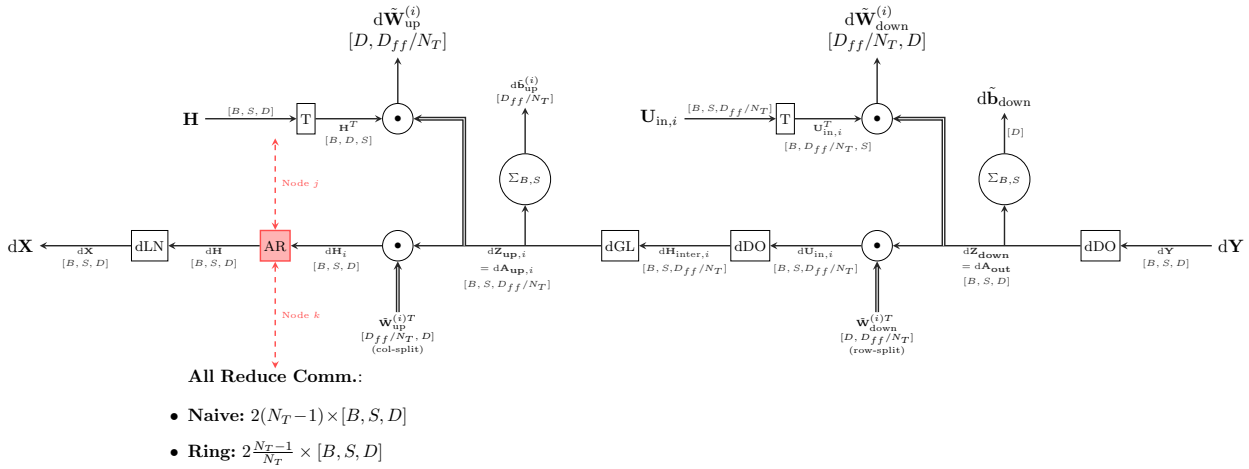


Figure 13: MLP backward pass with tensor parallelism. Each device backpropagates through its local up- and down-projection shards. The only collective in the backward path is an All-Reduce that sums the partial input gradients $\mathrm{d}\mathbf{H}^{(t)}$ across devices to obtain the full $\mathrm{d}\mathbf{H}$. Parameter gradients remain local to each shard.

- **MLP forward:** an All-Reduce after the row-parallel down-projection to form $\mathbf{Z}_{\text{down}}$ (Figure 12).
- **MLP backward:** an All-Reduce on the input gradient $\mathrm{d}\mathbf{H}$ from the column-parallel up-projection (Figure 13).

In all cases, the local computation on each device is structurally identical to the single-node computation described in Section 5; only the placement of collectives and the sharding of weights and activations differ. This makes tensor parallelism a natural extension of the single-node Transformer, well suited for large models that do not fit into the memory of a single accelerator.

7 Data Parallelism (DP)

In data parallelism, each replica holds a full copy of the model, but processes a different subset of the batch. Conceptually, each device runs the same forward and backward computation as in the single-node setting (Section 5), but on a different mini-batch. After the backward pass, gradients are synchronized across replicas via All-Reduce so that the optimizer update is identical to what would have been obtained on a single node with the full batch.

From the point of view of a single replica, the forward graph inside each Transformer block is therefore identical to the one in Section 5 (single-node), or to Section 6 (tensor-parallel) if TP is also enabled. The only change is the input: instead of the full batch $\mathbf{X} \in \mathbb{R}^{B\times S\times D}$, replica d sees a local shard $\mathbf{X}_d \in \mathbb{R}^{B_{\text{local}}\times S\times D}$, with $B_{\text{local}} = B/N_D$.

We denote the number of data-parallel replicas by N_D. Devices are indexed by $d \in \{0, \dots, N_D - 1\}$.

Key ideas:

- Each device has a full copy of the model parameters (weights and optimizer state).
- The global batch of size B is split into N_D local batches of size $B_{\text{local}} = B/N_D$.
- The forward pass on each device is identical to the single-node computation in Section 5, but uses only its local batch.
- The backward pass computes local gradients $\nabla W^{(d)}$ on each device.
- All-Reduce over all data-parallel replicas averages or sums the gradients, producing the same effective update as a single-node run with batch size B.

In contrast to tensor parallelism (Section 6), DP does not shard weight matrices or activations; instead, it replicates the entire model and splits only the data.

7.1 Overall DP Training Flow

Figure 14 shows a high-level view of data parallelism applied to a Transformer layer. Compared to the single-node overview (Figure 1), the main difference is that we now have N_D identical copies of the block, each processing a different input shard $\mathbf{X}_d$ and producing local predictions and losses.

For a single training step, the sequence of operations is:

1. **Batch sharding.** The input batch of size B is split into N_D local batches, each of size B_{local}:

$$\{\mathbf{X}_0, \dots, \mathbf{X}_{N_D-1}\}, \qquad \mathbf{X}_d \in \mathbb{R}^{B_{\text{local}}\times S\times D_{\text{in}}}.$$

   Each device d also receives the corresponding target tokens $\mathbf{Y}_d$.

2. **Local forward pass.** On each device, the full Transformer stack (input embedding, MHA, MLP, output projection) is applied to its local shard $\mathbf{X}_d$ exactly as in Section 5. If tensor parallelism from Section 6 is also used, then each replica runs the same TP-augmented forward graph on $\mathbf{X}_d$. In either case, the structure of the forward pass is unchanged; only the batch dimension and the presence of gradient synchronization differ. Each device produces local logits, probabilities, and loss $\mathcal{L}_d$.

3. **Local backward pass.** The loss $\mathcal{L}_d$ is backpropagated locally, yielding gradients $\nabla W^{(d)}$ for all parameters on device d.

4. **Gradient synchronization (All-Reduce).** For each parameter tensor W, an All-Reduce over the data-parallel group is performed:

$$\nabla W = \frac{1}{N_D} \sum_{d=0}^{N_D-1} \nabla W^{(d)}.$$

   After this step, all replicas hold identical averaged gradients ∇W.

5. **Optimizer update.** Each device applies the same optimizer (SGD, Adam, etc.) to its local copy of the parameters using the synchronized gradients. Because the gradients are identical across devices, the updated weights are also identical.

Apart from the All-Reduce in step 4, the computations on each device are identical copies of the single-node graph.

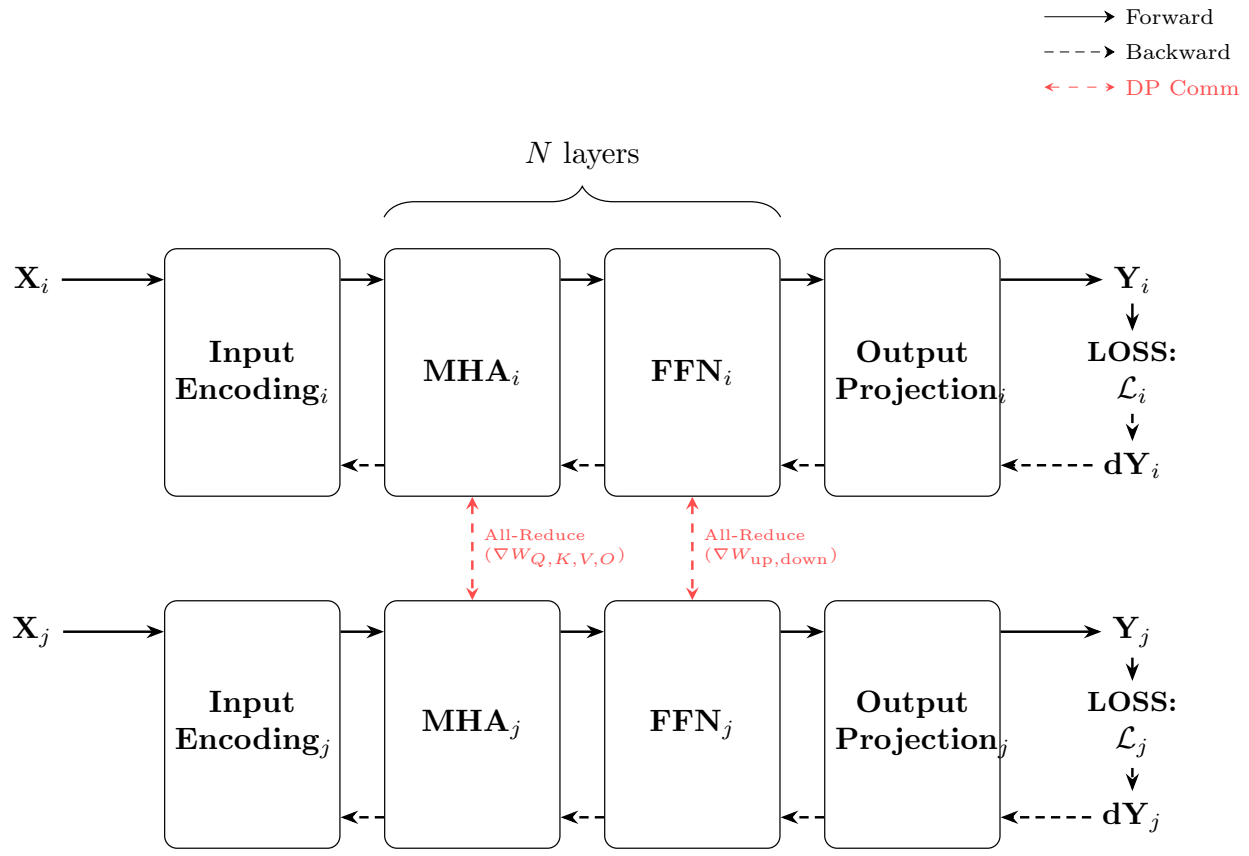


Figure 14: Overall Transformer layer under data parallelism. Each device holds a full copy of the model and processes a different shard of the input batch ($\mathbf{X}_i$, $\mathbf{X}_j$, ...). Forward and backward passes are performed locally as in the single-node case, and gradients for each block (MHA, MLP, output projection) are synchronized across replicas via All-Reduce.

7.2 Relationship to the Single-Node Computation

It is often useful to think of DP as a pure "wrapper" around the single-node computation from Section 5:

- **Same computation graph per replica.** For any given layer (MHA, MLP, output projection), the forward and backward graphs on a single device are identical to those in the single-node diagrams (Figures 3, 4, 5, 6, etc.), or to their tensor-parallel counterparts in Section 6 when TP is enabled. Only the input batch is different: $\mathbf{X}_d$ is a shard of the global batch.
- **Different mini-batches.** The only difference in the forward pass is that each replica sees a different chunk of the global batch. There is no communication needed for forward activations in DP.
- **Gradient aggregation only.** In the backward pass, each replica computes local gradients as if it were running independently. Only after all local gradients have been computed do we introduce communication: an All-Reduce over all replicas for each parameter tensor.

- **Equivalence to larger batch.** When gradients are averaged across replicas, the resulting update is mathematically equivalent to a single-node run with batch size $B = N_D \cdot B_{\text{local}}$ (ignoring subtle differences from e.g. dropout noise).

In contrast, tensor parallelism (Section 6) changes the graph inside each layer by sharding weight matrices and adding collectives in the middle of the forward and backward passes. DP keeps the layer graphs intact and adds communication only at the level of gradients.

7.3 MHA Backward under DP

For the multi-head attention block, the local backward pass on each device is exactly the same as in Figure 4: gradients flow from the loss through the output projection, attention heads, Q/K/V projections, and layer normalization back to the input $\mathbf{X}$.

The only DP-specific difference is how gradients with respect to the MHA parameters are combined across replicas. Let W_Q, W_K, W_V, W_O be the query, key, value, and output projection matrices. On device d, the local backward pass computes $\nabla W_Q^{(d)}, \nabla W_K^{(d)}, \nabla W_V^{(d)}, \nabla W_O^{(d)}$. An All-Reduce across all data-parallel replicas then aggregates these gradients:

$$\nabla W_Q = \frac{1}{N_D} \sum_{d=0}^{N_D-1} \nabla W_Q^{(d)}, \quad \nabla W_K = \frac{1}{N_D} \sum_{d=0}^{N_D-1} \nabla W_K^{(d)},$$

$$\nabla W_V = \frac{1}{N_D} \sum_{d=0}^{N_D-1} \nabla W_V^{(d)}, \quad \nabla W_O = \frac{1}{N_D} \sum_{d=0}^{N_D-1} \nabla W_O^{(d)}.$$

After this synchronization, each device holds the same averaged gradients $\nabla W_Q, \nabla W_K, \nabla W_V, \nabla W_O$.

Figure 15 illustrates this process: the internal nodes and shapes inside the MHA block are identical to Figure 4, but additional DP-specific boxes and red dashed arrows mark the All-Reduce operations over replica gradients.

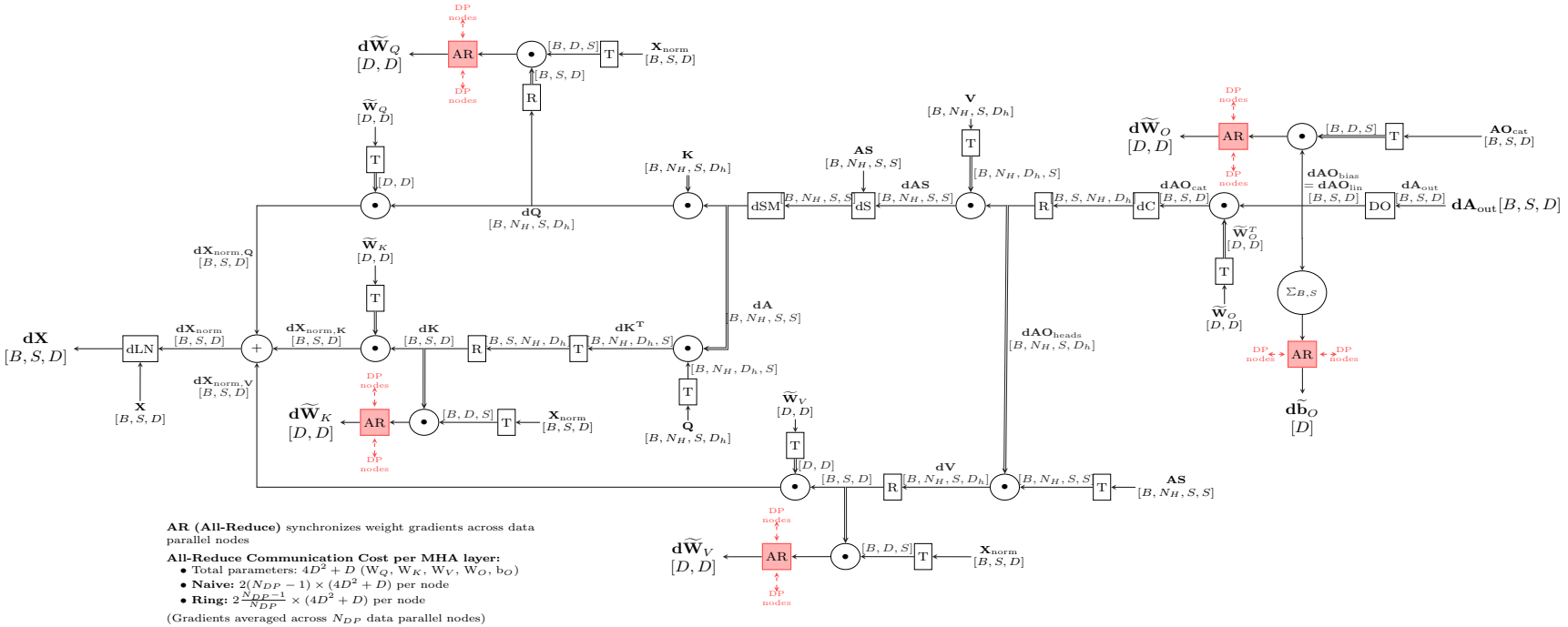


Figure 15: Multi-head attention backward pass under data parallelism. Each replica computes local gradients w.r.t. W_Q, W_K, W_V, and W_O using its own mini-batch. Red dashed arrows indicate All-Reduce operations that aggregate these local gradients across all data-parallel replicas to form the global gradients used by the optimizer. The internal structure of the backward graph matches the single-node case.

7.4 MLP Backward under DP

The situation for the MLP block is similar. Locally, each replica performs exactly the same backward computations as in Figure 6: gradients propagate from d$\mathbf{Y}$ through the down-projection, activation, up-projection, and (optional) layer normalization back to $\mathbf{H}$, while accumulating parameter gradients for $W_{\text{up}}, W_{\text{down}}, \mathbf{b}_{\text{up}}, \mathbf{b}_{\text{down}}$.

Under data parallelism, each replica d obtains its own local gradients $\nabla W_{\text{up}}^{(d)}, \nabla W_{\text{down}}^{(d)}$ and similarly for the biases. These are then synchronized across replicas via All-Reduce:

$$\nabla W_{\text{up}} = \frac{1}{N_D} \sum_{d=0}^{N_D-1} \nabla W_{\text{up}}^{(d)}, \quad \nabla W_{\text{down}} = \frac{1}{N_D} \sum_{d=0}^{N_D-1} \nabla W_{\text{down}}^{(d)}.$$

Bias gradients are treated analogously. Because the MLP input $\mathbf{H}$ is replicated across devices, no additional communication is needed for d$\mathbf{H}$ itself: each replica computes the same functional backward map but with different data, and the effect of aggregation is entirely captured by averaging the parameter gradients.

Figure 16 marks the locations of these All-Reduce operations in the MLP backward graph.

7.5 Communication and Memory Considerations

Compared to the single-node model in Section 5:

- **Memory per device.** Each device stores a full copy of the model parameters and optimizer states. Activation memory per device is reduced by a factor of N_D because each replica sees only a fraction of the global batch.
- **Communication pattern.** DP introduces communication only at gradient synchronization time (All-Reduce over parameter gradients). There is no communication on the forward path and no communication for activations.
- **Scalability.** Increasing N_D increases the effective batch size and reduces the per-device compute and activation memory, but the cost of All-Reduce grows with the number of replicas and the total parameter size.
- **Combination with other parallelism.** In practice, DP is often combined with tensor parallelism (and sometimes pipeline parallelism). In such settings, each data parallel group contains several tensor-parallel shards; gradient synchronization is then performed across data-parallel groups, while tensor-parallel collectives remain local to each group.

From the perspective of the computation graphs in Section 5, data parallelism is therefore the least invasive form of parallelism: it keeps the per-layer forward and backward structure unchanged and adds only gradient All-Reduces on top.

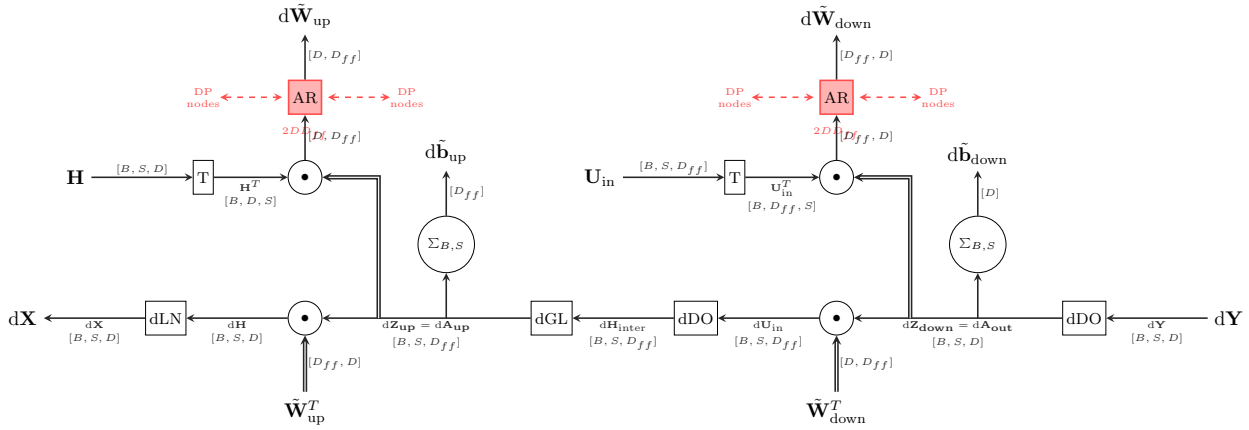


Figure 16: MLP backward pass under data parallelism. Each replica computes local gradients for the up- and down-projection weights and biases using its own mini-batch. Red boxes and dashed arrows indicate All-Reduce operations that aggregate these local parameter gradients across replicas. The structure of the backward computation inside each replica is identical to the single-node graph.

8 Combining Data Parallelism (DP) and Tensor Parallelism (TP)

In previous sections we have treated tensor parallelism (TP, Section 6) and data parallelism (DP, Section 7) separately. In practice, large-scale training almost always combines both:

- **Tensor parallelism** splits large weight matrices and activations across N_T devices inside a *model shard*.
- **Data parallelism** replicates that model shard across N_D different mini-batches and synchronizes gradients.

The total number of devices is therefore

$$N_{\text{devices}} = N_D \times N_T.$$

From the point of view of the computation graph:
- Inside one TP group, the forward and backward graphs are identical to those in Section 6 (TP-only). - Across DP replicas, the wrapping logic is identical to Section 7 (DP-only): each replica sees a different mini-batch and gradients are averaged across replicas.

The main difference from the previous chapters is therefore not in the shape of the per-layer graphs, but in how devices are organized into parallel groups and how multiple kinds of collective communication (All-Reduce and All-Gather) interact.

8.1 Parallel Groups and Layout

We conceptually arrange devices into a 2D grid:

- **TP dimension**: within each row, N_T devices form a tensor-parallel group and jointly store a single model shard.
- **DP dimension**: along the column dimension, N_D copies of that TP group process different shards of the batch.

In other words:
- Each TP group behaves like the TP-only model of Section 6, but only sees a local shard of the batch. - Each DP "column" behaves like the DP-only setup of Section 7, but each replica is itself a TP group instead of a single device.

If the global batch has size B, we split it across data-parallel groups as

$$B = N_D \cdot B_{\text{local}},$$

so that DP replica d sees $\mathbf{X}_d \in \mathbb{R}^{B_{\text{local}} \times S \times D}$ as its input. Within each replica, $\mathbf{X}_d$ is fed into the TP-sharded model exactly as in Section 6: single-node matmuls are replaced by column-parallel / row-parallel pairs across the N_T shards.

Figure 17 summarizes this layout and should be read as a DP+TP version of the single-node overview from Figure 1 and the TP overview in Figure 9.

8.2 From Single-Node to TP to DP+TP

It is useful to view DP+TP as a sequence of incremental modifications to the single-node computation of Section 5:

Single node (Section 5). All parameters and activations live on a single device. Each linear layer is a single matmul, and no collectives are needed.

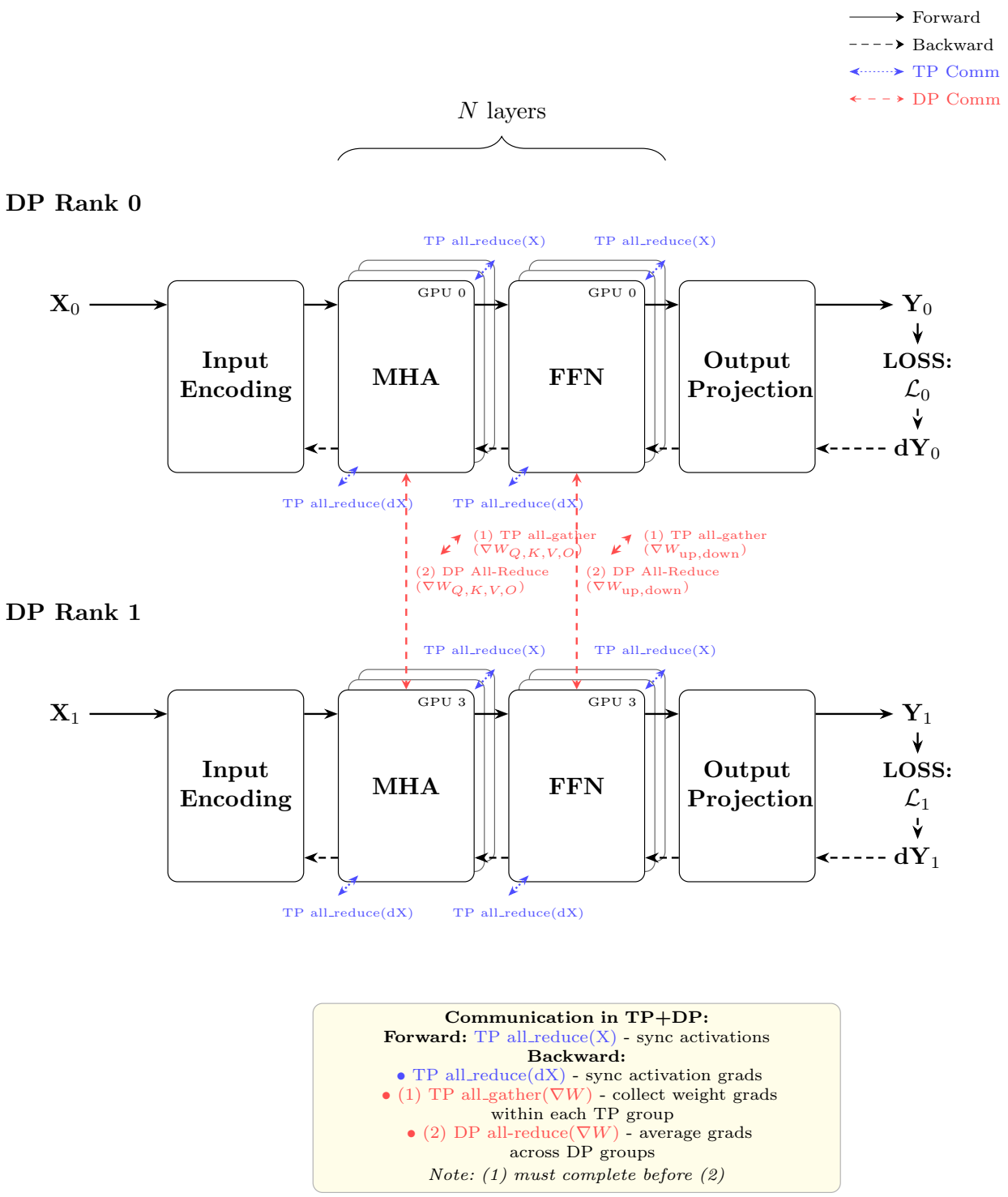


Figure 17: Overall Transformer layer with combined data and tensor parallelism. Horizontally, each tensor-parallel group of size N_T jointly implements a sharded version of the layer (as in Section 6). Vertically, N_D such groups form a data-parallel grid: each row processes a different mini-batch shard, and gradients are synchronized across rows. All-Reduce and All-Gather collectives are annotated where they appear.

Tensor parallelism only (Section 6). We split large matrices across N_T devices:

- **Column-parallel** linears: weights are split along the output dimension. Each shard computes a slice of the output; later operations either keep that sharded representation or perform an All-Gather to reconstruct the full tensor.
- **Row-parallel** linears: weights and inputs are split along the input dimension. Each shard computes a partial result, and an All-Reduce is used to sum partial outputs.

Within a single TP group, all devices see the same batch and activations are sharded along feature dimensions. Collectives (All-Reduce and possibly All-Gather) appear *inside* the layer graphs.

Data parallelism only (Section 7). We keep the single-node graphs intact but replicate them across N_D devices, each with a different mini-batch shard. The only collective is an All-Reduce over parameter gradients at the end of the backward pass.

Combining DP and TP. In DP+TP we apply both modifications at once:

- Each data-parallel replica is itself a TP group of N_T devices. Inside that group, all TP sharding and collectives from Section 6 apply as-is.
- Across data-parallel replicas, we perform gradient All-Reduces as in Section 7, now applied to *sharded* parameters (the TP shards of each weight matrix).

Thus, the internal structure of each MHA/MLP/output block on a *per shard* basis is unchanged; the only differences lie in how many devices share each tensor and which collectives connect them.

8.3 Forward Pass under DP+TP

From the perspective of a single TP shard inside DP+TP, the forward pass is the same as in the TP-only model (Section 6):

- the input is the local mini-batch shard $\mathbf{X}_d$ of shape $[B_{\text{local}}, S, D]$;
- column-parallel and row-parallel linears are applied as before;
- intermediate activations (e.g., per-head Q/K/V, intermediate MLP features) are sharded across the N_T devices.

The main new ingredient is the use of **All-Gather** on activations in certain places where a subsequent operation requires the full (unsharded) tensor.

Common examples include:

- **Vocabulary-parallel output projection.** If the final LM head weight $\mathbf{W}_{\text{lm}}$ is sharded across TP devices along the vocabulary dimension, each shard produces logits for a subset of the vocabulary. To compute the softmax and loss with the full vocabulary on each device, we perform an All-Gather of the partial logits slices to assemble the full $[B_{\text{local}}, S, V]$ tensor.
- **Embedding layers.** Similarly, if the token embedding matrix is sharded along the vocabulary dimension, embedding lookups may produce sharded embeddings that must be All-Gathered before being passed into a subsequent layer that expects an unsharded $[B_{\text{local}}, S, D]$ representation.
- **Non-sharded operations.** Any operation that conceptually acts on the full hidden dimension (e.g., a global normalization or a non-sharded residual branch) may require an All-Gather of the sharded activations before it can be applied, unless the model has been carefully arranged so that all such operations either run locally on shards or are replaced by TP-friendly variants.

All-Reduce and All-Gather thus play complementary roles inside a TP group:

- **All-Reduce** combines partial results that were computed on a sharded input (row-parallel linears, sharded gradients).
- **All-Gather** reassembles tensor slices that were produced by column-parallel linears or vocabulary-parallel projections when the next operation needs the full tensor.

Data parallelism does not change these intra-group collectives; it only changes which mini-batch $\mathbf{X}_d$ is fed into the TP group.

8.4 Backward Pass and Gradient Synchronization

In DP+TP, the backward pass combines:

- the TP-only backward graphs from Section 6, including all intra-group All-Reduce and All-Gather operations on activations and partial gradients, and
- the DP-only gradient synchronization from Section 7, now applied to the sharded parameters of each TP group.

Concretely, for a given parameter tensor W:

- In TP-only:
  - W is split into shards $W^{(t)}$ over $t = 0, \ldots, N_T - 1$, each stored on a different device inside a TP group.
  - Backward through the local matmul yields a local gradient $\nabla W^{(t)}$ on each shard.
  - If W appears in a row-parallel configuration, partial gradients w.r.t. inputs are All-Reduced across t.
- In DP+TP:
  - The above TP logic is applied independently within each DP replica d, yielding $\nabla W^{(t,d)}$ on shard t in replica d.
  - *Across* replicas, we perform an All-Reduce over d for each shard t:

$$\nabla W^{(t)} = \frac{1}{N_D} \sum_{d=0}^{N_D-1} \nabla W^{(t,d)}.$$

    This is a data-parallel gradient synchronization, but now replicated over all TP shards in parallel.

Thus, every parameter shard participates in two types of collectives:

- **TP collectives** inside a DP replica: All-Reduce/All-Gather across t (row-/column-parallel logic).
- **DP collectives** across replicas: All-Reduce across d for each shard index t.

The combination ensures that, at the end of the backward pass, all TP groups in all DP replicas hold identical, properly aggregated gradients for their local shards $W^{(t)}$, so that the optimizer update is consistent across the entire system.

8.5 Communication Summary and Differences from TP-only / DP-only

We summarize the main differences compared to TP-only (Section 6) and DP-only (Section 7):

Compared to TP-only.

- **Same per-layer sharding.** Inside one TP group, the forward/backward graphs, the placement of All-Reduce and All-Gather, and the tensor shapes are identical to the TP-only case.
- **Additional gradient All-Reduce over DP replicas.** Every parameter shard now participates in an extra All-Reduce over the DP dimension to average gradients across mini-batch shards.
- **No change in local activations.** Activation memory and compute patterns inside a TP group do not change when DP is added; DP only affects which samples are seen by each replica and how parameter gradients are combined.

Compared to DP-only.

- **Sharded model within each replica.** Instead of a full copy of the model per DP replica, each replica contains a TP-sharded model: parameters and some activations are split across N_T devices.
- **More frequent collectives.** DP-only uses All-Reduce once per step per parameter tensor. In DP+TP, we additionally have intra-replica All-Reduce/All-Gather operations inside most layers (MHA, MLP, embeddings, LM head).
- **Per-device memory and compute.** TP reduces per-device parameter and activation memory by approximately $1/N_T$, at the cost of extra intra-layer communication. DP further reduces per-device activation memory by a factor of $1/N_D$ by splitting the batch.

Overall, DP+TP can be seen as:

> "Take the TP-sharded Transformer layer of Section 6, run it on different mini-batch shards as in Section 7, and add one more All-Reduce per parameter shard to average gradients across replicas."

All-Gather operations appear wherever TP has produced sharded activations that must be globally visible (e.g., logits over the full vocabulary), while All-Reduce is used both to combine partial TP results and to aggregate gradients across data-parallel replicas.

9 Summary and Practical Takeaways

This chapter summarizes the main ideas of the previous sections and connects them to practical questions about how Transformer training actually runs on hardware. Rather than introducing new notation, we focus on a small number of mental models that are useful when reading execution diagrams, reasoning about cost, or deciding how to parallelize a model.

9.1 From Gradients to Graphs to Transformers

The early chapters of this document built up three layers of intuition:

- **Gradients as linear maps.** Backpropagation through a layer can be viewed as another computation graph, where each forward operation contributes one or more backward operations (often transposed matmuls). This leads to the simple rule of thumb that a linear layer roughly doubles the cost in the backward pass.
- **Graph conventions.** Forward tensors, gradients, and parameters are represented as nodes with explicit shapes. Arrows and labels make it possible to read off which matmuls and elementwise ops appear in each phase without looking at code.
- **Single-node Transformer.** A Transformer layer is just a composition of:
  - embedding lookup + positional encoding,
  - MHA (LN, QKV projections, attention, output projection, residual),
  - MLP (LN, up-projection, non-linearity, down-projection, residual),
  - output projection + softmax + loss.

  Each block has a forward graph and a matching backward graph with one or two matmuls for each forward matmul.

The single-node diagrams in Section 5 are therefore the "base case" for all later parallel variants: every parallel configuration is just a way of splitting, replicating, and reconnecting this same underlying graph.

9.2 Cost and Memory: What Really Matters

Several general patterns show up across all layers:

- **Matmuls dominate.** The vast majority of FLOPs live in linear layers (Q/K/V projections, attention output projection, MLP up/down projections, output projection). Elementwise ops (softmax, GELU, layernorm) are important for correctness but cheap by comparison.
- **Backward is $\approx 2\times$ forward for matmuls.** Each forward matmul XW produces at least:
  - one backward matmul to propagate gradients to X, and
  - one backward matmul to accumulate gradients into W.

  This is explicitly visible in the backward diagrams for MHA and MLP.
- **Residual connections do not add much cost.** They split gradients but do not change asymptotic complexity: a residual add introduces cheap elementwise ops in both directions.
- **Layer normalization is modest but ubiquitous.** LN adds some per-token compute and a small number of parameters. Its backward pass is a bit more involved than, say, ReLU, but still far cheaper than the surrounding matmuls.

When approximating the cost of a model, it is therefore usually enough to count the large matmuls and remember that the backward pass is roughly twice as expensive as the forward pass for those layers.

9.3 Single-Node vs. TP vs. DP vs. DP+TP

The later chapters compared four execution modes:

- **Single node (Section 5).** All parameters and activations live on one device. The diagrams in that section are the reference implementation: no sharding, no collectives, just a sequence of matmuls and elementwise ops.
- **Tensor parallelism (Section 6).** Weight matrices and some activations are split across N_T devices inside a *model shard*. Column-parallel and row-parallel linears replace the single-node matmuls. All-Reduce and All-Gather collectives appear *inside* the layer graphs to combine partial results or reassemble sharded tensors.
- **Data parallelism (Section 7).** The full model (or full TP shard) is replicated across N_D devices. Each replica sees a different mini-batch shard and runs the same forward/backward graph as in the single-node or TP case. The only new collective is an All-Reduce over parameter gradients at the end of the backward pass.
- **DP + TP (Section 8).** Each DP replica is itself a TP group. Inside a TP group, all intra-layer collectives from Section 6 (All-Reduce, All-Gather) still apply. Across replicas, additional All-Reduces average gradients for each TP shard. Some layers (e.g., vocabulary-parallel output heads) require All-Gather on activations when a subsequent operation needs a full (unsharded) tensor.

In other words:

Single node	no collectives, full weights on one device
TP	shard weights/activations, intra-layer collectives
DP	replicate model, gradient All-Reduce only
DP+TP	TP inside each replica + DP gradient All-Reduce

Conceptually, none of these modes change what the Transformer *is* computing; they only change where each piece of the graph runs and how partial results are stitched back together.

9.4 Reading and Using the Diagrams

The diagrams throughout this document are designed to answer a small set of recurring questions:

- **Where are the big matmuls?** Look for rectangular nodes with shapes like $[B, S, D]$ and weight matrices with shapes $[D, D_{\text{ff}}]$, $[D, D]$, or $[D, D_{\text{vocab}}]$. These are the main sources of FLOPs.
- **Where do gradients flow?** Dashed arrows and backward-specific nodes show how $\mathrm{d}\mathcal{L}$ propagates. Following these arrows makes it clear which tensors need to be stored for backward and where activations can be recomputed.
- **Where is memory used?** Any place where a forward tensor is needed again in backward contributes to activation memory. Attention score tensors $[B, N_H, S, S]$ and intermediate MLP activations $[B, S, D_{\text{ff}}]$ are particularly expensive.
- **Where do collectives occur?** In TP and DP diagrams, colored or labeled arrows denote All-Reduce and All-Gather. These positions determine the communication cost and the critical path length when scaling to many devices.

With this vocabulary, one can often look at a new architecture (e.g., a variant MLP, a different attention mechanism) and immediately sketch its core cost and parallelization pattern.

9.5 Practical Rules of Thumb

The following informal rules capture many of the practical lessons from the previous chapters:

- **Start with the single-node graph.** Always begin reasoning from the simplest view: what would the model do on a single device? Then layer TP and DP on top.
- **Think in matmuls, not lines of code.** Count how many large matmuls appear per token, and remember that each will be paid for again (roughly twice) in backward.
- **TP trades memory for intra-layer communication.** Sharding along feature dimensions reduces per-device memory but introduces All-Reduce/All-Gather inside layers. Use it when the model is too large for a single device.
- **DP trades global batch size for gradient communication.** Replicating the model is simple and keeps layer graphs intact, but every parameter must participate in a gradient All-Reduce.
- **DP + TP is the default for very large models.** In practice, most large-scale systems combine both: TP to make the model fit, DP to scale the batch and throughput.
- **Collectives define your scaling limits.** Once the model fits in memory, additional speedups are mostly limited by the number, size, and placement of All-Reduce and All-Gather operations.

9.6 Where to Go Next

This document deliberately focused on a single Transformer layer and a small set of parallel strategies. Extensions in real systems include:

- **Pipeline parallelism:** splitting layers across devices in depth and overlapping micro-batches.
- **Activation checkpointing:** trading recomputation for reduced activation memory.
- **Optimizer and parameter sharding:** partitioning optimizer state and even the parameters themselves across nodes in more complex ways.

However, the core mental model remains the same: start from a precise computation graph (as in the single-node diagrams), then ask how that graph is sliced across devices, where communication appears, and how gradients are aggregated. If those three questions are clear, most seemingly complicated parallel configurations reduce to simple, familiar building blocks.