Chapter 8: The Transformer Architecture

Learning Objectives

After reading this chapter, the reader should be able to:

1. Derive scaled dot-product attention from first principles, explaining why the $1/\sqrt{d_k}$ scaling factor is necessary to prevent softmax saturation for large key dimensions.
2. Explain multi-head attention: how splitting queries, keys, and values into h heads enables the model to attend to information from different representation subspaces simultaneously.
3. Compare positional encoding strategies — sinusoidal, learned, rotary (RoPE), and ALiBi — and articulate the tradeoffs between absolute, relative, and extrapolatable position representations.
4. Trace the forward pass through a complete Transformer block (self-attention, add-and-norm, feed-forward, add-and-norm) and explain the roles of residual connections, layer normalization, and the feed-forward network.

8.1 From Recurrence to Attention

Attention lets every position look at every other position. But if attention alone can capture all relationships, why do we still need the RNN?

As we established in Chapter 1, the central question of this book is how to estimate $P(w_{\text{next}} \mid w_{<t})$ — the probability of the next word given all previous words. The Transformer provides the most powerful answer yet by replacing sequential recurrence with fully parallelized self-attention, enabling training on vast amounts of data at unprecedented scale. We now explain how this is possible.

In Chapter 7, we constructed a complete encoder-decoder system for conditional text generation. The encoder, built from stacked LSTM cells, compressed an input sequence into a series of hidden states. The attention mechanism then allowed the decoder to selectively consult those states at each generation step, bypassing the information bottleneck of the fixed-length context vector. The result was a significant improvement in translation quality — the model could now align source and target tokens dynamically rather than relying on a single compressed representation. Yet one fundamental bottleneck remained untouched: the recurrence itself. The encoder still had to process every token sequentially, one hidden state update at a time. For a sentence of T tokens, this meant T sequential operations that no amount of parallel hardware could compress. The question that motivated the Transformer architecture is, in retrospect, obvious: if attention already gives us direct access to all encoder states, why do we need the sequential hidden state chain that produced them in the first place?

8.1.1 The Parallelization Problem with RNNs

The $\mathcal{O}(T)$ sequential bottleneck in recurrent networks is not an implementation detail that faster hardware can dissolve — it is a logical dependency baked into the architecture. Recall from Chapter 5 that the hidden state update rule for an LSTM is $\mathbf{h}_t = \text{LSTM}(\mathbf{h}_{t-1}, \mathbf{x}_t)$. The state at time t depends directly on the state at time $t-1$, which depends on the state at $t-2$, and so on back to the beginning of the sequence. This chain of dependencies is not an accident of implementation but a logical necessity: an RNN's power comes precisely from the fact that each state summarizes all

information up to that point. The price of this power is sequential processing. No matter how many processors are available, one cannot compute $\mathbf{h}_t$ until $\mathbf{h}_{t-1}$ is complete. For a sequence of length $T = 1000$, the encoder requires 1000 sequential operations with the minimum possible latency equal to the time required for 1000 matrix-vector multiplications in serial.

The contrast with matrix operations is instructive. A single matrix multiplication $\mathbf{AB}$, where $\mathbf{A} \in \mathbb{R}^{T \times d}$ and $\mathbf{B} \in \mathbb{R}^{d \times d}$, can exploit the full parallelism of a graphics processing unit (GPU). Modern GPU architectures execute thousands of floating-point operations simultaneously, meaning that a matrix multiplication of scale $T \times d$ can be completed in depth proportional to $\log T$ rather than T linear steps. An RNN encoder, by contrast, requires exactly T sequential steps regardless of model dimension or hardware capacity. It is tempting to think that the attention mechanism introduced in Chapter 6 already solved this problem. Attention solved the information bottleneck — the difficulty of compressing an entire source sentence into a single vector — but left the sequential processing bottleneck completely intact. The RNN encoder in an attention-augmented model still processes all T tokens sequentially; attention only changes how the decoder consults the resulting hidden states. The wall-clock time for encoding scales linearly with sequence length in both the attentional and non-attentional RNN. This is the bottleneck that the Transformer eliminates.

8.1.2 Self-Attention as a Replacement for Recurrence

If we no longer update hidden states sequentially, how can each position acquire information about the rest of the sequence?

Chapter 6 introduced self-attention as a mechanism by which each position in a sequence attends to all other positions of the same sequence. At that stage, self-attention was presented as an augmentation to a recurrent model, providing a shortcut for long-range dependencies that would otherwise have to travel through many sequential steps. The key insight of the Transformer is to go further: self-attention can replace recurrence entirely. Instead of building contextual representations through sequential state updates, each position directly computes a weighted combination of all other positions in a single, massively parallel operation. One useful mental picture is that of a conference call in which every participant can hear every other participant simultaneously, rather than a relay chain in which information passes person-to-person. The analogy is imperfect — a conference call has no notion of speaker order, the positional encoding problem addressed in Section 8.4 — but it captures the shift from sequential to simultaneous exchange. The computation for all T positions can be expressed as a single matrix multiplication, $\mathbf{QK}^\top$, which produces a $T \times T$ matrix of pairwise compatibility scores in one step. This eliminates the sequential dependency entirely: every position can be updated in parallel, reducing the number of serial operations from T to $\mathcal{O}(1)$.

The resulting representations differ fundamentally from those produced by RNNs. In an LSTM, the representation at position t encodes context accumulated incrementally from position 1 to t: early positions in a long sentence have, in effect, been "processed earlier" than late positions, and information about early tokens must travel through many timesteps to influence late tokens. In self-attention, every output position attends to every input position in a single layer: there is no notion of "earlier" or "later" processing. This global connectivity at every layer means that self-attention can, in principle, capture any pairwise relationship regardless of distance, without the vanishing-gradient problem that makes long-range dependencies difficult for RNNs. The trade-off is computational: self-attention requires $\mathcal{O}(T^2)$ space and time to compute the full attention matrix, whereas an RNN requires only $\mathcal{O}(T)$ sequential operations of $\mathcal{O}(d^2)$ cost each. For typical sequence lengths ($T < 2048$) and modern hardware, the quadratic attention cost is acceptable and is vastly

outweighed by the parallelization benefit.

8.1.3 The Transformer Hypothesis: Attention Is All You Need

Is it truly possible to build a powerful sequence model with no recurrence and no convolution?

Vaswani et al. [2017] made a bold architectural bet: remove all recurrence and convolution from sequence modeling, relying entirely on self-attention — supplemented by positional encodings, feed-forward layers, residual connections, and layer normalization — to build contextual representations. The authors chose the title "Attention Is All You Need" — a claim that was approximately as understated as the architectural revolution it announced. The paper was deliberately provocative: it asserted that the recurrent component, considered indispensable for sequence modeling for 25 years, was unnecessary. The claim was empirically validated immediately. The base Transformer model achieved 28.4 BLEU on WMT 2014 English-German translation, surpassing all previous state-of-the-art results, and trained in 3.5 days on 8 P100 GPUs — a small fraction of the time required for comparable recurrent models. The large Transformer achieved 41.0 BLEU on WMT 2014 English-French, the best result reported at the time. The result was, frankly, striking: what made it remarkable was not merely the quality of the translations but the speed of training. We remember the surprise in the community when these results first circulated — the idea that recurrence could simply be removed struck many as reckless, and the paper's title was viewed as provocative rather than descriptive.

It is important to be precise about what the Transformer hypothesis claims. The title "Attention Is All You Need" refers specifically to the claim that recurrence is not needed: attention can replace the recurrent chain. The Transformer does still require several other components beyond attention: positional encodings inject the word-order information that self-attention cannot represent (Section 8.4), feed-forward networks add per-position nonlinearity that pure attention cannot provide (Section 8.5.4), residual connections enable deep stacking without gradient vanishing (Section 8.5.2), and layer normalization stabilizes training (Section 8.5.3). The architecture is thus not "attention and nothing else" but rather "attention instead of recurrence." This distinction matters because students sometimes expect a Transformer to function with only the attention mechanism and are puzzled when the other components turn out to be equally important. You might reasonably assume that a model titled "Attention Is All You Need" requires only the attention module; without positional encodings, residual connections, and layer normalization, the attention layers would quickly collapse into untrainable representations.

> **Sidebar: Why "Transformer"?**
>
> The paper that introduced the Transformer had one of the most memorable titles in machine learning: "Attention Is All You Need." But where did the name "Transformer" itself come from? The model was developed in 2017 by a team at Google Brain and Google Research, led by Ashish Vaswani, in a collaboration that bridged research and engineering. According to retrospective accounts by the authors, the name "Transformer" was chosen to evoke the idea of transforming representations — each layer transforms input representations into progressively more abstract output representations through layers of attention. The name also carried an aspirational quality: the team believed the architecture would transform the field of natural language processing (NLP). What the authors could not have fully anticipated was the magnitude of the transformation. Within two years, the Transformer had become the basis for Bidirectional Encoder Representations from Transformers (BERT), Generative Pre-trained Transformer 2

(GPT-2), and T5, effectively replacing all previous architectures for NLP tasks. Within five years, it powered GPT-4, Gemini, and Claude, becoming the computational substrate for large-scale AI research. The original base model had 65 million parameters and the large model 213 million. Modern frontier models are estimated to have hundreds of billions or more. The architecture has scaled by orders of magnitude in under a decade, and the end of that scaling is not yet in sight.

Section 8.2 now derives the core computation of the Transformer — scaled dot-product attention — with full mathematical rigor, including the critical scaling argument that makes training stable.

8.2 Scaled Dot-Product Attention

How do we formalize the intuition of "looking up relevant information" as a matrix computation, and why does a single square root stand between a working model and a broken one?

The attention mechanism introduced in Chapter 6 was presented in terms of scalar compatibility scores between a query vector and a sequence of key vectors, with the scores normalized through softmax to produce weights over a sequence of value vectors. The Transformer reformulates this computation entirely in terms of matrix operations, allowing all queries to be processed simultaneously. The result is the scaled dot-product attention formula, which is simultaneously the Transformer's most important equation and its most elegant one.

8.2.1 Queries, Keys, and Values as Linear Projections

The terminology of queries, keys, and values was borrowed from information retrieval — a query is what you seek, a key identifies what is stored, and a value is the stored content — and the metaphor predates the Transformer by several years. In the Transformer, every input position is associated with three vectors: a query $\mathbf{q}$, a key $\mathbf{k}$, and a value $\mathbf{v}$. These three vectors are computed by applying distinct learned linear projections to the same input representation. If $\mathbf{X} \in \mathbb{R}^{T \times d_{\text{model}}}$ is the matrix of input representations (one row per token), then the three projection matrices $\mathbf{W}^Q \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $\mathbf{W}^K \in \mathbb{R}^{d_{\text{model}} \times d_k}$, and $\mathbf{W}^V \in \mathbb{R}^{d_{\text{model}} \times d_v}$ produce the query, key, and value matrices as $\mathbf{Q} = \mathbf{X}\mathbf{W}^Q$, $\mathbf{K} = \mathbf{X}\mathbf{W}^K$, and $\mathbf{V} = \mathbf{X}\mathbf{W}^V$, where $\mathbf{Q} \in \mathbb{R}^{T \times d_k}$, $\mathbf{K} \in \mathbb{R}^{T \times d_k}$, and $\mathbf{V} \in \mathbb{R}^{T \times d_v}$.

The three roles are conceptually distinct. The query $\mathbf{q}_i$ represents what position i is looking for: it encodes the information need of that position. The key $\mathbf{k}_j$ represents what position j contains or offers: it encodes the type of information available at that position. The value $\mathbf{v}_j$ represents the actual content that position j contributes to the output: it is the information retrieved when position j is attended to. The dot product $\mathbf{q}_i^\top \mathbf{k}_j$ measures the compatibility between the information need of position i and the information offered by position j. Crucially, a natural but incorrect intuition is that $\mathbf{Q}$, $\mathbf{K}$, and $\mathbf{V}$ must come from three different inputs. In self-attention, all three are projections of the same input matrix $\mathbf{X}$: each position simultaneously asks a question (query), advertises its content (key), and prepares information for retrieval (value). The three separate projections allow the model to learn different, task-adapted representations for each role. In cross-attention — used in encoder-decoder Transformers — the queries come from the decoder sequence while the keys and values come from the encoder output, implementing the bridging mechanism from Chapter 7.

8.2.2 The Dot-Product Attention Formula

What is the mathematical form of the complete attention computation, and how do the dimensions flow from input to output?

Given the query, key, and value matrices, the attention output is computed as a single formula. The matrix product $\mathbf{QK}^\top \in \mathbb{R}^{T\times T}$ computes all T^2 pairwise compatibility scores simultaneously: entry (i, j) of this matrix equals $\mathbf{q}_i^\top \mathbf{k}_j$, the dot product between the query at position i and the key at position j. Applying the softmax function row-wise to this matrix normalizes each row into a probability distribution over the T positions, producing the attention weight matrix $\mathbf{A} \in \mathbb{R}^{T\times T}$, where $\sum_j \mathbf{A}_{ij} = 1$ for every i. Multiplying by the value matrix $\mathbf{V}$ then produces the output matrix $\mathbf{Z} \in \mathbb{R}^{T\times d_v}$, where row i is the weighted sum $\sum_j \mathbf{A}_{ij}\mathbf{v}_j$: each output position receives a convex combination of all value vectors, weighted by attention. The full scaled dot-product attention formula is:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{QK}^\top}{\sqrt{d_k}}\right)\mathbf{V} \tag{8.1}$$

A common mistake is to think of the $\mathbf{QK}^\top$ product as a vector operation. It is a full matrix multiplication producing a $T \times T$ matrix: every position's query is compared against every position's key, yielding T^2 scores. For a 512-token sequence, this means computing 262,144 scalar compatibility scores in one pass. The output dimension is d_v, matching the value dimension rather than the key dimension, which is an important detail for parameter counting in multi-head attention.

8.2.3 Why Scale by $\sqrt{d_k}$?

What goes wrong without the scaling factor, and why does the square root of the key dimension appear specifically?

The scaling factor $1/\sqrt{d_k}$ in Equation (8.1) is one of the most important practical decisions in the Transformer architecture, and its justification is a clean exercise in probability theory. In our experience, the scaling factor is the single concept that most separates students who understand the Transformer mechanically from those who understand it deeply. Assume that the components of the query vector $\mathbf{q}$ and key vector $\mathbf{k}$ are independent random variables with mean zero and unit variance. The dot product $\mathbf{q}^\top\mathbf{k} = \sum_{i=1}^{d_k} q_i k_i$ is then a sum of d_k terms. For each term, $\mathbb{E}[q_i k_i] = \mathbb{E}[q_i]\mathbb{E}[k_i] = 0$ and $\text{Var}(q_i k_i) = \text{Var}(q_i)\text{Var}(k_i) + \text{Var}(q_i)\mathbb{E}[k_i]^2 + \text{Var}(k_i)\mathbb{E}[q_i]^2 = 1$ for independent zero-mean unit-variance variables. By the linearity of variance for independent terms, $\text{Var}(\mathbf{q}^\top\mathbf{k}) = d_k$, and therefore the standard deviation of the dot product is $\sqrt{d_k}$.

For large d_k, the dot products can be extremely large in magnitude. For $d_k = 64$ (typical in the base Transformer), the standard deviation is 8, meaning dot products routinely fall in the range $[-24, +24]$. For $d_k = 512$, the standard deviation is approximately 22.6, and dot products span $[-68, +68]$. The softmax function is highly sensitive to the scale of its inputs: when one input is 24 while others are near zero, the softmax output places nearly all probability mass on that one input and near-zero mass on all others. This near-one-hot distribution has two catastrophic consequences for training. First, the gradient of the softmax with respect to all non-maximal inputs becomes vanishingly small, preventing the model from learning which other positions it should attend to. Second, the model becomes effectively deterministic in its attention patterns from early in training, eliminating the flexibility to discover useful soft combinations of multiple positions.

Dividing by $\sqrt{d_k}$ normalizes the dot product variance to 1, keeping the softmax inputs in the range $[-3, +3]$ where gradients flow freely and attention weights remain soft. The lesson is elegant: a single scalar normalization, justified by basic variance analysis, is the difference between a model that trains well and one that fails to learn. A question we frequently receive at this point is: "Why not normalize by the maximum dot product instead of $\sqrt{d_k}$?" The answer is that the maximum dot product is data-dependent and unknown at initialization, whereas $\sqrt{d_k}$ is a fixed architectural constant providing the correct expected normalization under standard initialization assumptions.

Sidebar: The $1/\sqrt{d_k}$ Factor

Of all the details in the original Transformer paper, one of the most important is also one of the easiest to overlook: the division by the square root of the key dimension in the attention formula. Why does this seemingly minor detail matter so much? The answer lies in the behavior of the softmax function at extreme values. When attention scores $\mathbf{q}^\top \mathbf{k}$ are large in magnitude — which happens naturally when d_k is large, since the dot product variance grows linearly with dimension — the softmax saturates: almost all probability mass concentrates on a single token, and the gradients for all other tokens become vanishingly small. This is catastrophic for learning because the model becomes trapped in near-deterministic attention patterns. The variance of the dot product for random unit-variance entries is exactly d_k, so its standard deviation is $\sqrt{d_k}$. Dividing by $\sqrt{d_k}$ normalizes the dot products to have unit variance, keeping attention weights in the soft regime where the model can learn meaningful patterns. Practitioners who have tried removing the scaling factor report that training either fails to converge or reaches a substantially worse solution. The lesson extends beyond attention: in deep learning, controlling the variance of intermediate computations — through initialization schemes, batch normalization, layer normalization, or explicit scaling — is often the decisive factor in whether a deep model trains at all. In the words of the original paper's Section 3.2.1: "We suspect that for large values of d_k, the dot products grow large in magnitude, pushing the softmax function into regions where it has extremely small gradients." The $1/\sqrt{d_k}$ factor is the simple, principled fix.

8.2.4 Masking for Autoregressive Models

How do we prevent a language model from "cheating" by reading future tokens during training?

For autoregressive language modeling, the model must predict the next token based only on previous tokens. During training with teacher forcing, we present the entire input sequence to the model simultaneously and train it to predict each token from its prefix. Without any additional mechanism, the self-attention computation at position t would attend to positions $t+1, t+2, \ldots, T$ — precisely the future tokens that should be unknown at generation time. This information leakage would produce a model that performs perfectly on training data but cannot generate text at inference time (since future tokens do not yet exist). The causal mask prevents this. We define a mask matrix $\mathbf{M} \in \mathbb{R}^{T \times T}$ such that $\mathbf{M}_{ij} = 0$ if $j \leq i$ and $\mathbf{M}_{ij} = -\infty$ if $j > i$. Adding this mask to the pre-softmax scores gives:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top + \mathbf{M}}{\sqrt{d_k}}\right)\mathbf{V}$$

Since $\exp(-\infty) = 0$, the softmax assigns exactly zero weight to all future positions. The causal

mask is a lower-triangular binary matrix: position i can attend to positions $1, 2, \ldots, i$ and is blocked from positions $i+1, \ldots, T$. This ensures that the model's prediction at each position depends only on the causal prefix, matching the autoregressive factorization that governs text generation. Encoder models (BERT-style) do not use the causal mask, since their task is to produce contextual representations of a fully observed input. Encoder-decoder models (T5-style) apply the mask in the decoder's self-attention sublayer but not in the cross-attention sublayer, since cross-attention reads from the encoder output, which is fully observed.

8.2.5 Numerical Walkthrough

Let us make this concrete with numbers. We work through a complete numerical example with $T = 4$ positions and $d_k = d_v = 3$ to make the matrix dimensions concrete. Suppose the query, key, and value matrices are as follows (with values chosen for illustrative clarity):

$$\mathbf{Q} = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad \mathbf{K} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}, \quad \mathbf{V} = \begin{pmatrix} 1 & 2 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 2 \\ 2 & 1 & 0 \end{pmatrix}$$

Each matrix has shape 4×3. We first compute the raw score matrix $\mathbf{Q}\mathbf{K}^\top \in \mathbb{R}^{4\times 4}$. Entry (i, j) is $\mathbf{q}_i^\top \mathbf{k}_j$. For example, $\mathbf{q}_1^\top \mathbf{k}_1 = 1 \cdot 1 + 0 \cdot 0 + 1 \cdot 0 = 1$, $\mathbf{q}_1^\top \mathbf{k}_3 = 1 \cdot 1 + 0 \cdot 0 + 1 \cdot 1 = 2$. Computing all 16 entries gives:

$$\mathbf{Q}\mathbf{K}^\top = \begin{pmatrix} 1 & 0 & 2 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 2 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix}$$

Next we scale by $1/\sqrt{d_k} = 1/\sqrt{3} \approx 0.577$:

$$\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{3}} \approx \begin{pmatrix} 0.577 & 0 & 1.155 & 0.577 \\ 0 & 0.577 & 0 & 0.577 \\ 0.577 & 0.577 & 1.155 & 0.577 \\ 0 & 0 & 0.577 & 0.577 \end{pmatrix}$$

Applying softmax row-wise to the first row $[0.577, 0, 1.155, 0.577]$: the exponentials are $e^{0.577} \approx 1.780$, $e^0 = 1$, $e^{1.155} \approx 3.174$, $e^{0.577} \approx 1.780$, summing to 7.734. The first row of the attention weight matrix is therefore approximately $[0.230, 0.129, 0.410, 0.230]$, and we can verify that these sum to 1. The output at position 1 is the weighted combination $0.230 \cdot \mathbf{v}_1 + 0.129 \cdot \mathbf{v}_2 + 0.410 \cdot \mathbf{v}_3 + 0.230 \cdot \mathbf{v}_4$, which is a 3-dimensional vector carrying weighted information from all four value vectors. The full attention matrix $\mathbf{A}$ has shape 4×4 and the output $\mathbf{Z} = \mathbf{A}\mathbf{V}$ has shape 4×3, matching the shape of the input value matrix. The output at each position is a convex combination of all value vectors, with weights determined by the compatibility of that position's query with each position's key. This walkthrough confirms the dimensional flow: input $\mathbf{X}$ of shape $T \times d_{\text{model}}$ is projected to $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ of shape $T \times d_k$ (or $T \times d_v$ for $\mathbf{V}$), the score matrix $\mathbf{Q}\mathbf{K}^\top$ has shape $T \times T$, and the final output has shape $T \times d_v$.

```python
import torch
import torch.nn.functional as F

def scaled_dot_product_attention(Q, K, V, mask=None):
    """Compute scaled dot-product attention."""
    d_k = Q.size(-1)
    # Compute attention scores: QK^T / sqrt(d_k)
    scores = torch.matmul(Q, K.transpose(-2, -1)) / (d_k ** 0.5)
    # Apply causal mask if provided
    if mask is not None:
        scores = scores.masked_fill(mask == 0, float('-inf'))
    # Softmax to get attention weights
    weights = F.softmax(scores, dim=-1)
    # Weighted sum of values
    output = torch.matmul(weights, V)
    return output, weights

# Example: 1 batch, 4 positions, d_k = 8
Q = torch.randn(1, 4, 8)
K = torch.randn(1, 4, 8)
V = torch.randn(1, 4, 8)
output, weights = scaled_dot_product_attention(Q, K, V)
print(f"Output shape: {output.shape}")      # [1, 4, 8]
print(f"Attention weights:\n{weights[0]}")   # [4, 4] matrix
```

The scaled dot-product attention formula, Equation (8.1), is the fundamental operation of the Transformer. In the following sections, we show how running this operation in parallel across multiple learned subspaces — multi-head attention — dramatically increases the model's representational capacity without a proportional increase in parameters.

8.3 Multi-Head Attention

If a single attention function computes one type of relationship, what do we gain by running multiple attention functions in parallel?

A single application of scaled dot-product attention, as described in Section 8.2, produces one weighted combination of value vectors for each position. The attention weights encode one set of pairwise relationships — perhaps syntactic dependencies, or perhaps semantic similarity, depending on what the projection matrices $\mathbf{W}^Q$, $\mathbf{W}^K$, $\mathbf{W}^V$ have learned to emphasize. But language is rich with simultaneous relationship types. In the sentence "The government raised the interest rate because inflation was rising," a parser attending to syntactic structure must link "government" to "raised," "rate" to "raised," and "inflation" to "was." Simultaneously, a coreference resolver must link pronouns to their antecedents, and a semantic role labeler must link arguments to predicates. A single attention function must learn to compromise among all these relationship types, potentially failing to capture any of them precisely. Multi-head attention solves this by running h independent attention functions in parallel, each operating in a lower-dimensional subspace, allowing each head to specialize in a different type of relationship.

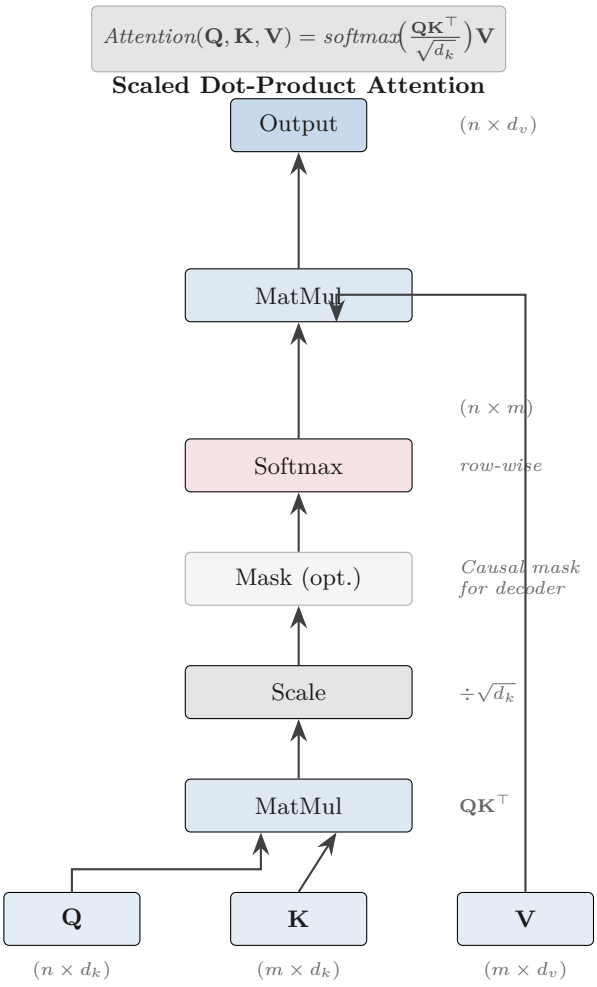

Figure 1: Figure 8.1 – Scaled Dot-Product Attention Computation Flow

8.3.1 The Motivation for Multiple Heads

Why is a single attention function an insufficient model of the multi-dimensional structure of language?

Consider the attention weight matrix produced by a single attention head attending to the sentence "The cat that sat on the mat was hungry." The word "was" must simultaneously establish a syntactic dependency with "cat" (the subject of "was hungry"), track that "that" is a relative pronoun modifying "cat," and recognize that "mat" and "cat" rhyme but are otherwise unrelated. A single attention head with a single set of query-key projections can emphasize only one type of relationship at a time: if it learns to compute syntactic subject-verb attention, it will suppress the other relationship types. The solution is to project the input into h different subspaces — each of dimension $d_k = d_{\text{model}}/h$ — and compute attention independently within each subspace. Head 1 might learn a query-key projection that highlights syntactic subject-verb dependencies. Head 2 might learn a projection that highlights semantic similarity. Head 3 might attend primarily to adjacent tokens, capturing local n-gram patterns. The outputs of all heads are then concatenated and projected back to the original dimension d_{model}, producing a representation that jointly encodes all the relationship types that the heads have individually learned. This is the core intuition behind multi-head attention: not a single global view of the input, but a collection of complementary, specialized views.

8.3.2 Projection into Subspaces

How do separate projection matrices enable each head to attend to a distinct aspect of the input?

For each of the h heads, we define three distinct projection matrices: $\mathbf{W}_i^Q \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $\mathbf{W}_i^K \in \mathbb{R}^{d_{\text{model}} \times d_k}$, and $\mathbf{W}_i^V \in \mathbb{R}^{d_{\text{model}} \times d_v}$, where $d_k = d_v = d_{\text{model}}/h$. Each head then independently computes projected query, key, and value matrices: $\mathbf{Q}_i = \mathbf{X}\mathbf{W}_i^Q$, $\mathbf{K}_i = \mathbf{X}\mathbf{W}_i^K$, $\mathbf{V}_i = \mathbf{X}\mathbf{W}_i^V$. Because each head has its own projection matrices, the inputs to head i's attention computation are genuinely different from those of head j: the projections select different linear combinations of the input features, effectively "looking at" different aspects of the representation. In the original Transformer base model, $d_{\text{model}} = 512$ and $h = 8$, so each head operates with $d_k = d_v = 64$. Each head's attention is therefore a 64-dimensional function, much lower-dimensional than the full 512-dimensional representation. The reduction in dimension per head ensures that the total computational cost of multi-head attention is comparable to that of single-head attention with full dimension d_{model} — the heads are narrower, but there are h of them. It is tempting to think that reducing d_k per head loses information. In fact, the combined outputs of all h heads span $h \cdot d_v = d_{\text{model}}$ dimensions, recovering the full representational capacity. The subspace decomposition is a factorization that enables diversity, not a compression that discards information.

8.3.3 Parallel Attention and Concatenation

How are the outputs of the parallel heads combined into a single output representation?

Once all h heads have computed their individual attention outputs $\text{head}_i \in \mathbb{R}^{T \times d_v}$, the multi-head attention output is produced by concatenating them along the feature dimension and projecting back to d_{model}. The complete multi-head attention formula is:

$$\text{MultiHead}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)\mathbf{W}^O \tag{8.2}$$

where $\text{head}_i = \text{Attention}(\mathbf{Q}\mathbf{W}_i^Q, \mathbf{K}\mathbf{W}_i^K, \mathbf{V}\mathbf{W}_i^V)$.

The concatenation $\text{Concat}(\text{head}_1, \ldots, \text{head}_h) \in \mathbb{R}^{T \times d_{\text{model}}}$ stacks the h output matrices side by side along the feature axis. The output projection $\mathbf{W}^O \in \mathbb{R}^{d_{\text{model}} \times d_{\text{model}}}$ then mixes information across heads, producing the final output of shape $T \times d_{\text{model}}$. In practice, all heads are computed in parallel using batched matrix operations: instead of computing h separate matrix multiplications, the projection matrices for all heads are stacked into a single tensor and all projections are computed in one batched operation. This batched computation is a key efficiency advantage: multi-head attention does not require h sequential attention computations, but rather h simultaneous ones, with the "parallel" in "parallel attention" referring to actual parallelism on GPU hardware.

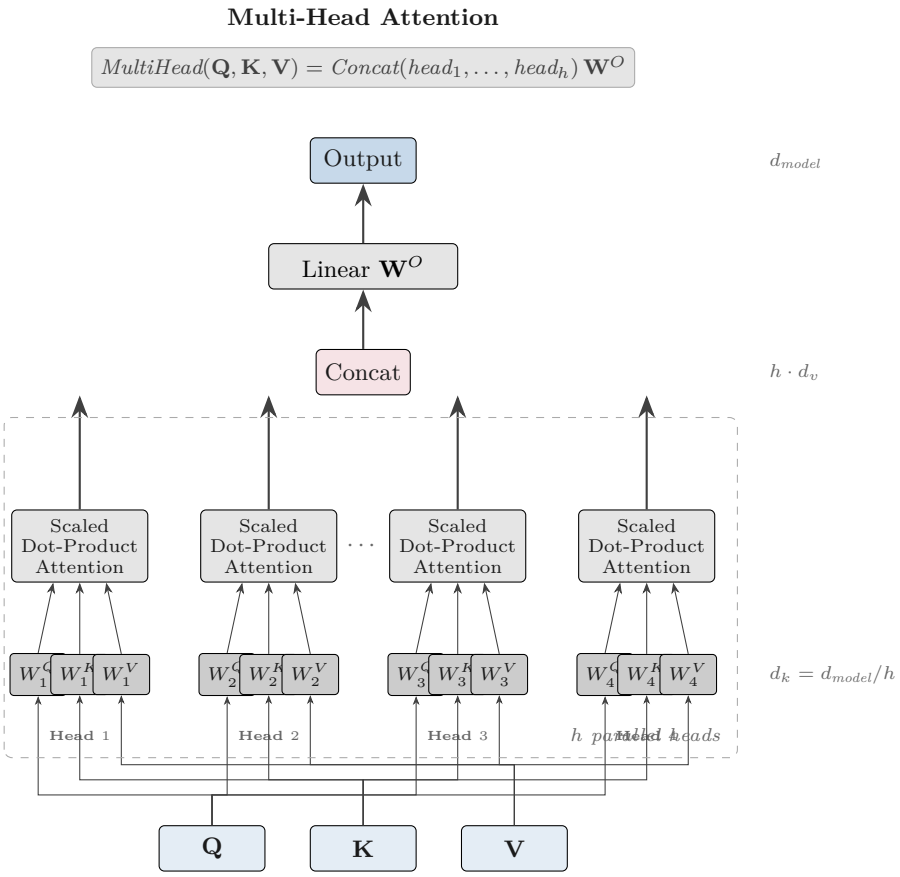

Figure 2: Figure 8.2 – Multi-Head Attention

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class MultiHeadAttention(nn.Module):
    def __init__(self, d_model, h):
        super().__init__()
        self.h = h
```

```
        self.d_k = d_model // h
        # Single projection matrices for all heads (batched)
        self.W_Q = nn.Linear(d_model, d_model, bias=False)
        self.W_K = nn.Linear(d_model, d_model, bias=False)
        self.W_V = nn.Linear(d_model, d_model, bias=False)
        self.W_O = nn.Linear(d_model, d_model, bias=False)

    def forward(self, X, mask=None):
        B, T, d_model = X.shape
        # Project and reshape into h heads: (B, T, d_model) -> (B, h, T, d_k)
        Q = self.W_Q(X).view(B, T, self.h, self.d_k).transpose(1, 2)
        K = self.W_K(X).view(B, T, self.h, self.d_k).transpose(1, 2)
        V = self.W_V(X).view(B, T, self.h, self.d_k).transpose(1, 2)
        # Scaled dot-product attention per head
        scores = torch.matmul(Q, K.transpose(-2, -1)) / (self.d_k ** 0.5)
        if mask is not None:
            scores = scores.masked_fill(mask == 0, float('-inf'))
        weights = F.softmax(scores, dim=-1)
        head_out = torch.matmul(weights, V)          # (B, h, T, d_k)
        # Concatenate heads and project
        out = head_out.transpose(1, 2).reshape(B, T, d_model)
        return self.W_O(out), weights

mha = MultiHeadAttention(d_model=16, h=4)
X = torch.randn(1, 6, 16)
output, weights = mha(X)
print(f"Output shape: {output.shape}")           # [1, 6, 16]
print(f"Head attention shapes: {weights.shape}") # [1, 4, 6, 6]
```

8.3.4 What Different Heads Learn

We should be candid: despite numerous visualization studies, the question of what individual attention heads "learn" remains only partially answered. Clark et al. [2019] provided one of the most illuminating empirical accounts of this question, and the results are simultaneously striking and humbling. Empirical analysis of trained Transformer models reveals that attention head specialization is a genuine phenomenon, not merely a theoretical possibility. Clark et al. [2019] analyzed the attention patterns of BERT, finding that specific heads correspond to specific linguistic relationships: one head in layer 5 consistently attends to the direct objects of verbs, another head attends to the immediately preceding token (tracking local sequential patterns), and several heads attend globally across all positions. Voita et al. [2019] found that certain heads are critical for model performance while others are largely redundant: in a 12-head BERT model, pruning 8 of the 12 heads in most layers degraded performance only slightly, with the remaining 4 heads per layer capturing nearly all the linguistically meaningful information. This finding has practical implications for model compression (discussed in Chapter 14): not all heads are equally useful, and attention head pruning can reduce model size significantly. The important insight, however, is that the mechanism of multi-head attention provides the structural opportunity for specialization to emerge through training, without any explicit supervision. The model is never told that head 3 should track subject-verb agreement; it discovers this specialization because it reduces training loss.

8.3.5 Parameter Count Analysis

Does using h heads instead of one head increase the total number of parameters?

Multi-head attention does not increase the total parameter count relative to single-head attention with the same d_{model}. Each of the h heads has three projection matrices of size $d_{\text{model}} \times d_k = d_{\text{model}} \times (d_{\text{model}}/h)$, contributing $3 \times d_{\text{model}} \times (d_{\text{model}}/h)$ parameters per head, and $3 \times h \times d_{\text{model}} \times (d_{\text{model}}/h) = 3 \times d_{\text{model}}^2$ parameters total across all heads. Adding the output projection $\mathbf{W}^O \in \mathbb{R}^{d_{\text{model}} \times d_{\text{model}}}$ contributes d_{model}^2 parameters. The total parameter count per multi-head attention layer is therefore $4 \times d_{\text{model}}^2$ — exactly the same as single-head attention with $d_k = d_{\text{model}}$ plus an output projection. For the base Transformer with $d_{\text{model}} = 512$, each attention layer has $4 \times 512^2 = 1{,}048{,}576$ parameters, approximately 1 million. For the large Transformer with $d_{\text{model}} = 1024$, each layer has $4 \times 1024^2 \approx 4.2$ million parameters. The factorization into h heads does not change the total parameter budget: it changes how those parameters are organized, from one large attention function to h smaller, specialized ones.

Multi-head attention is the central mechanism of the Transformer's representational power. In the next section, we turn to the problem that multi-head attention cannot solve: the complete absence of positional information in the attention computation.

8.4 Position Encodings

Self-attention treats the input as a set rather than a sequence. How do we restore the ordering information that language requires?

The scaled dot-product attention formula (Equation 8.1) is permutation-equivariant: if we permute the rows of the input matrix $\mathbf{X}$, the output rows are permuted in exactly the same way. To see this formally, let $\mathbf{P}$ be a permutation matrix. Then $\text{Attention}(\mathbf{PXW}^Q, \mathbf{PXW}^K, \mathbf{PXW}^V) = \mathbf{P} \cdot \text{Attention}(\mathbf{XW}^Q, \mathbf{XW}^K, \mathbf{XW}^V)$: the attention output is the same up to the same permutation applied to the rows. This means that without additional intervention, self-attention assigns identical contextual representations to "the cat sat on the mat" and "mat the sat cat the on" — any permutation of the same words produces the same set of output vectors, just in a different order. Word order is linguistically critical: "the dog bites the man" and "the man bites the dog" describe entirely different events, yet a permutation-equivariant attention model would assign the same representation to both. Position encodings are the mechanism by which the Transformer breaks this symmetry and restores sensitivity to word order.

8.4.1 Why Position Matters

Scaled dot-product attention is permutation-equivariant: any reordering of the input tokens produces an identically reordered output. This is an exact property, and its consequence is that word order is entirely invisible to the attention mechanism without explicit intervention. We establish this rigorously for the single-head case; the multi-head case follows identically. Let $\mathbf{X}' = \mathbf{PX}$ where $\mathbf{P}$ is a $T \times T$ permutation matrix. The projected matrices under the permuted input are $\mathbf{Q}' = \mathbf{X}'\mathbf{W}^Q = \mathbf{PXW}^Q = \mathbf{PQ}$, and similarly $\mathbf{K}' = \mathbf{PK}$, $\mathbf{V}' = \mathbf{PV}$. The score matrix is $\mathbf{Q}'\mathbf{K}'^\top = \mathbf{PQ}(\mathbf{PK})^\top = \mathbf{PQK}^\top\mathbf{P}^\top$. Applying softmax row-wise to this permuted score matrix and multiplying by $\mathbf{PV}$: since $\mathbf{P}$ is a permutation matrix, $\text{softmax}(\mathbf{PSP}^\top)\mathbf{PV} = \mathbf{P} \cdot \text{softmax}(\mathbf{S}) \cdot \mathbf{V}$, where $\mathbf{S} = \mathbf{QK}^\top/\sqrt{d_k}$. The output under the permuted input is precisely the permuted output under the original input. The model cannot distinguish different orderings of the same words. Contrary

to what one might expect, self-attention cannot "learn" positional information from training data without explicit position encodings. This is mathematically impossible: the attention computation is provably permutation-equivariant, and no amount of training data can endow it with positional sensitivity. Position information must be injected explicitly, and the four approaches discussed in this section represent four different philosophies for how to do so. We omit the formal proof that this extends to the multi-head case, which follows by applying the single-head argument to each head independently.

8.4.2 Sinusoidal Position Encodings

Can a fixed mathematical function provide position information that generalizes to any sequence length?

Vaswani et al. [2017] proposed adding sinusoidal functions of position to the input embeddings before passing them to the first Transformer layer. The encoding is a deterministic function: for position $pos \in \{0, 1, \ldots, T-1\}$ and dimension index $i \in \{0, 1, \ldots, d_{\text{model}}/2 - 1\}$, the positional encoding is defined as:

$$\text{PE}(pos, 2i) = \sin\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right), \quad \text{PE}(pos, 2i+1) = \cos\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right) \tag{8.3}$$

This produces a $T \times d_{\text{model}}$ matrix of position encodings, one row per position, one column per model dimension. The encoding for position pos is a vector of d_{model} values, alternating between sines and cosines at geometrically spaced frequencies. Dimension 0 uses frequency $1/10000^0 = 1$, oscillating rapidly with period 2π. Dimension $d_{\text{model}} - 2$ uses frequency $1/10000 \approx 0.0001$, oscillating very slowly. Each position therefore has a unique "fingerprint" across the d_{model} dimensions, since the combination of $d_{\text{model}}/2$ different frequencies at a given position uniquely encodes that position (analogously to how a combination of clock hands at different speeds uniquely identifies a time).

The sinusoidal encoding has a mathematically elegant relative-position property. Using the angle addition formulas $\sin(a+b) = \sin a \cos b + \cos a \sin b$ and $\cos(a+b) = \cos a \cos b - \sin a \sin b$, we can express $\text{PE}(pos+k, 2i)$ and $\text{PE}(pos+k, 2i+1)$ as linear functions of $\text{PE}(pos, 2i)$ and $\text{PE}(pos, 2i+1)$. Specifically, for each frequency $\omega_i = 1/10000^{2i/d_{\text{model}}}$, the pair $[\text{PE}(pos+k, 2i), \text{PE}(pos+k, 2i+1)]$ is obtained from $[\text{PE}(pos, 2i), \text{PE}(pos, 2i+1)]$ by applying the rotation matrix $\begin{pmatrix} \cos(k\omega_i) & \sin(k\omega_i) \\ -\sin(k\omega_i) & \cos(k\omega_i) \end{pmatrix}$. Since this rotation matrix depends only on the offset k and not on the absolute position pos, the model can in principle learn to recognize relative offsets through a fixed linear transformation of the position encoding vectors. This is the theoretical motivation for sinusoidal encodings: they admit a position-invariant, relative-position representation.

```
import torch
import math

def sinusoidal_positional_encoding(T, d_model):
    """Compute sinusoidal positional encodings for T positions and d_model dims."""
    PE = torch.zeros(T, d_model)
    pos = torch.arange(0, T, dtype=torch.float).unsqueeze(1)  # (T, 1)
    # Frequencies: 1/10000^(2i/d_model) for i = 0, 1, ..., d_model//2 - 1
    i = torch.arange(0, d_model // 2, dtype=torch.float)
```

```
    freq = torch.exp(-math.log(10000.0) * 2 * i / d_model)     # (d_model//2,)
    # Even dimensions: sin; odd dimensions: cos
    PE[:, 0::2] = torch.sin(pos * freq)
    PE[:, 1::2] = torch.cos(pos * freq)
    return PE

PE = sinusoidal_positional_encoding(T=128, d_model=64)
print(f"PE shape: {PE.shape}")                          # [128, 64]
print(f"Position 0 norm: {PE[0].norm():.4f}")
print(f"Position 127 norm: {PE[127].norm():.4f}")
```

Figure 8.3: Sinusoidal Positional Encoding Heatmap

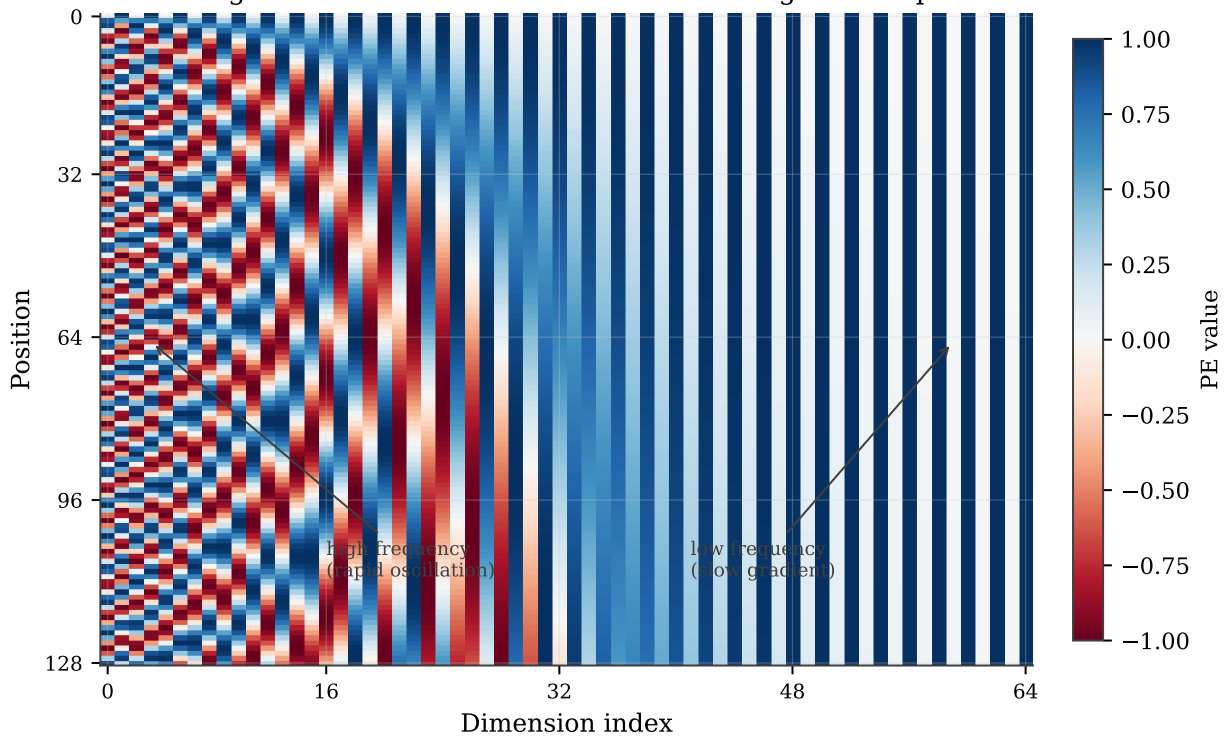

Figure 3: Figure 8.3 – Sinusoidal Positional Encoding Heatmap

8.4.3 Learned Position Embeddings

The simplest approach is often the best starting point: treat position encodings as learnable parameters, just like word embeddings. We define a learnable parameter matrix $\mathbf{P} \in \mathbb{R}^{T_{\max} \times d_{\text{model}}}$, where $T_{\max}$ is the maximum sequence length supported at training time, and add the row $\mathbf{P}[pos]$ to the input embedding at position pos. BERT uses learned position embeddings with $T_{\max} = 512$, and GPT-2 uses them with $T_{\max} = 1024$. Learned embeddings have the advantage of simplicity: they are straightforward to implement, require no special handling, and can adapt to task-specific positional patterns that sinusoidal encodings might not capture. The critical disadvantage is the hard position limit: a model trained with $T_{\max} = 512$ has no embedding for position 513, and naive application to longer sequences would require either truncation or ad-hoc extrapolation of the learned embeddings. The model parameters for positions $T_{\max} + 1$ and beyond were never

trained and have no meaningful value. In practice, BERT's 512-token limit has been a significant operational constraint, motivating numerous extensions (Longformer, BigBird) that either change the attention pattern or switch to a relative positional encoding scheme.

8.4.4 Rotary Position Embeddings (RoPE)

Is it possible to encode relative position information through the attention dot product itself, without modifying the input embeddings?

We confess that RoPE's elegance is easier to appreciate in retrospect than on first encounter — the rotation-matrix formulation can feel opaque until one works through the inner-product identity by hand, and readers who find the formalism challenging may take comfort in knowing that the practical implementation is far simpler than the derivation suggests. Rotary Position Embedding (RoPE), proposed by Su et al. [2021], takes a fundamentally different approach: rather than adding position information to the input before attention, RoPE encodes position by rotating the query and key vectors just before computing the attention dot product. The rotation is applied to pairs of dimensions $(2i, 2i+1)$ using a rotation matrix parameterized by the absolute position m and the dimension index i:

$$f_q(\mathbf{x}_m, m) = \mathbf{R}_\Theta^m \mathbf{W}_q \mathbf{x}_m \tag{8.4}$$

where $\mathbf{R}_\Theta^m$ is a block-diagonal rotation matrix with 2×2 rotation blocks of angle $m \cdot \theta_i$, and $\theta_i = 10000^{-2i/d}$. The key insight of RoPE is that the dot product between a rotated query at position m and a rotated key at position n satisfies:

$$\left(\mathbf{R}_\Theta^m \mathbf{W}_q \mathbf{x}_m\right)^\top \left(\mathbf{R}_\Theta^n \mathbf{W}_k \mathbf{x}_n\right) = \mathbf{x}_m^\top \mathbf{W}_q^\top \mathbf{R}_\Theta^{n-m} \mathbf{W}_k \mathbf{x}_n$$

This is a function of the relative offset $n - m$ rather than the absolute positions m and n, because $(\mathbf{R}_\Theta^m)^\top \mathbf{R}_\Theta^n = \mathbf{R}_\Theta^{n-m}$ by the rotation group property. RoPE therefore naturally encodes relative position through the attention mechanism itself, without requiring a separate relative-position computation. This is in contrast to sinusoidal encodings, which are added to the input and therefore appear in both the query and key representations in an entangled way. RoPE has become the standard positional encoding for modern large language models, including LLaMA, PaLM, and GPT-NeoX, because it supports length extrapolation — the model can be applied to sequences longer than those seen during training — and introduces no additional parameters.

8.4.5 ALiBi and Relative Position Biases

Can we inject positional information without modifying either the input embeddings or the query-key vectors?

ALiBi (Attention with Linear Biases), proposed by Press et al. [2022], takes yet another approach: instead of modifying the input representations or the query-key projections, ALiBi adds a linear bias to the attention scores based on the distance between positions. The modified attention score between positions i and j is:

$$\text{score}(i, j) = \frac{\mathbf{q}_i^\top \mathbf{k}_j}{\sqrt{d_k}} - m_s \cdot |i - j|$$

Figure 8.4: RoPE Visualization

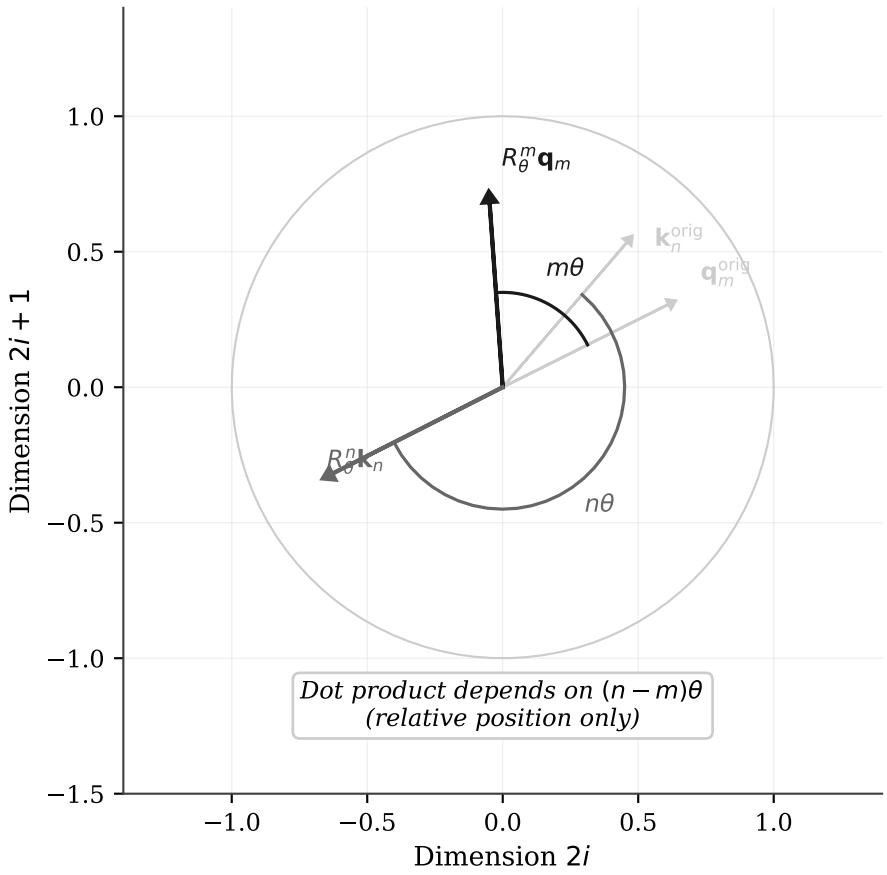

Figure 4: Figure 8.4 – RoPE Visualization

where m_s is a head-specific scalar slope. The slopes are set to a fixed geometric sequence: head 1 uses slope $2^{-8/h}$, head 2 uses slope $2^{-2 \cdot 8/h}$, and so on up to head h which uses slope $2^{-1} = 0.5$. These slopes enforce a recency bias: attending to a token that is k positions away incurs a penalty of $m_s \cdot k$ on the attention score, making it harder for the model to attend to distant tokens. ALiBi requires no positional parameters in the input embeddings (the input is simply the sum of token embeddings), and the slopes are fixed hyperparameters rather than learned weights. This introduces zero additional parameters compared to a model with no positional encoding at all. Press et al. [2022] showed that ALiBi enables strong extrapolation: models trained on sequences of length 1024 can generalize to sequences of length 6144 at inference time, substantially outperforming sinusoidal and learned encoding schemes on long-sequence benchmarks. The recency bias is not a hard constraint — if the content-based attention score $\mathbf{q}_i^\top \mathbf{k}_j / \sqrt{d_k}$ is sufficiently high, the model can overcome the distance penalty and attend to distant positions when needed. ALiBi is used in the BLOOM language model family and several other open-source LLMs.

8.4.6 Comparison and Extrapolation

Given four positional encoding strategies with different tradeoffs, how should one choose among them?

Table 8.1 summarizes the four positional encoding strategies across the key dimensions of interest.

Approach	Absolute/Relative	Learned/Fixed	Extrapolation	Parameters	Representative Models
Sinusoidal	Absolute	Fixed	Limited	0	Original Transformer
Learned	Absolute	Learned	No	$T_{\max} \times d_{\text{model}}$	BERT, GPT-2
RoPE	Relative	Fixed	Good	0	LLaMA, PaLM, GPT-NeoX
ALiBi	Relative bias	Fixed slopes	Strong	0	BLOOM

The progression from sinusoidal to learned to RoPE to ALiBi reflects a gradual movement toward relative position representations and better extrapolation. Sinusoidal encodings were the original solution: deterministic, requiring no parameters, but encoding absolute rather than relative position and offering only limited extrapolation beyond the training length. Learned embeddings are simpler to implement and task-adaptive, but impose a hard maximum sequence length and cannot extrapolate at all. RoPE encodes relative position inherently through the rotation structure and extrapolates well, at the cost of more complex implementation. ALiBi is the most aggressively extrapolatable, relying on a simple linear bias that is independent of any learned representation. Modern LLMs have converged on RoPE as the dominant choice for general-purpose models, with ALiBi remaining competitive for applications where extreme length extrapolation is critical.

8.5 The Full Transformer Block

Having derived each component separately, how do we assemble them into a complete building block that can be stacked arbitrarily deep?

The individual components developed in Sections 8.2 through 8.4 — multi-head attention, positional encodings — are necessary but not sufficient for a Transformer. To build a deep, trainable model, we also need mechanisms that stabilize the signal as it flows through many layers and that add nonlinear transformation capacity beyond what attention provides. These mechanisms are residual connections, layer normalization, and the feed-forward sublayer. This section shows how these components combine into the Transformer block: a self-contained unit that takes a $T \times d_{\text{model}}$ representation as input and produces a $T \times d_{\text{model}}$ representation as output, with shape preservation enabling arbitrary stacking.

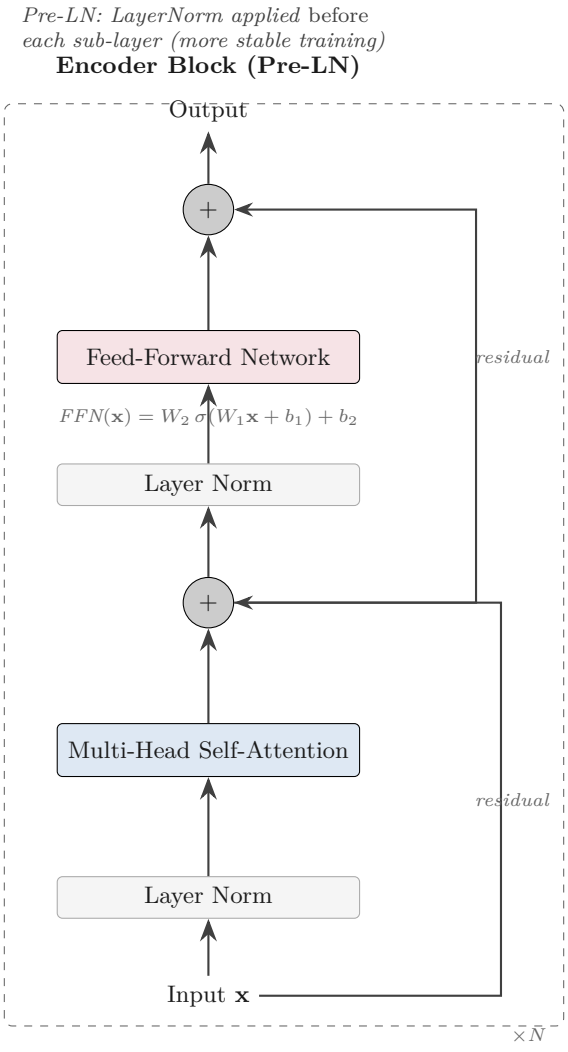

Figure 5: Figure 8.5 – Encoder Block Diagram

8.5.1 The Attention Sublayer

What is the role of the self-attention sublayer within the full Transformer block?

The first sublayer in each Transformer block is the multi-head self-attention layer from Section

8.3, applied in self-attention mode: $\mathbf{Q} = \mathbf{K} = \mathbf{V} = \mathbf{X}$, where $\mathbf{X}$ is the block's input. The output has shape $T \times d_{\text{model}}$, matching the input. This sublayer is the primary mechanism for inter-token communication: each output position has "looked at" every other position (in the encoder) or every previous position (in the causal decoder) and aggregated information from them. The self-attention sublayer is the only place within a single Transformer block where information moves between positions: all other operations (layer normalization, feed-forward) are applied independently and identically to each position. This architectural separation between "position mixing" (attention) and "position-wise processing" (feed-forward) is a fundamental design principle of the Transformer. Students often expect self-attention alone to constitute a complete Transformer block. The attention sublayer provides rich inter-position context but lacks the per-position nonlinear transformation capacity that is needed for the model to learn complex position-specific functions. That capacity is provided by the feed-forward sublayer (Section 8.5.4).

8.5.2 Residual Connections and Their Role

Why do we add the input of each sublayer to its output, and what does this accomplish mathematically?

Residual connections, introduced by He et al. [2016] for deep convolutional image recognition networks, add the input of a sublayer directly to its output: output $= \mathbf{x} + \text{Sublayer}(\mathbf{x})$. This seemingly simple modification has profound consequences for training deep networks. Consider the gradient of the loss $\mathcal{L}$ with respect to the input $\mathbf{x}$ of a sublayer. By the chain rule:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{x}} = \frac{\partial \mathcal{L}}{\partial \text{output}} \cdot \left(\mathbf{I} + \frac{\partial \text{Sublayer}(\mathbf{x})}{\partial \mathbf{x}} \right)$$

The key term is the identity matrix $\mathbf{I}$: it ensures that the gradient always has a component of magnitude 1 propagating backward through the residual connection, regardless of the sublayer's gradient magnitude. Without residual connections, gradients in deep networks are products of many Jacobian matrices, and in the limit of many layers, these products tend toward zero (the vanishing gradient problem) or infinity (the exploding gradient problem). With residual connections, the gradient at any layer receives a direct contribution from the loss without passing through all the subsequent sublayers' Jacobians. This enables stacking of 6, 12, 24, 48, or even 96 Transformer blocks without the gradient vanishing that would prevent training in deep plain networks.

There is a second, complementary benefit of residual connections: they encourage each sublayer to learn a "residual function," the correction to the input rather than a full representation from scratch. It is generally easier for a sublayer to learn a small delta $\Delta(\mathbf{x})$ than to learn the complete mapping $\mathbf{f}(\mathbf{x})$, because the residual can be initialized near zero (representing the identity function) and then refined during training. He et al. [2016] observed that residual networks of 152 layers outperformed plain networks of 34 layers, a reversal of the usual pattern where plain networks degrade with depth — a phenomenon they attributed precisely to optimization difficulty rather than capacity.

8.5.3 Layer Normalization

How does normalizing activations within each forward pass stabilize training, and why does the position of the normalization within the residual block matter?

Layer normalization [Ba et al., 2016] normalizes the activations of a layer by subtracting the mean and dividing by the standard deviation computed across the feature dimension, then applying learned affine parameters:

$$\text{LayerNorm}(\mathbf{x}) = \gamma \odot \frac{\mathbf{x} - \mu}{\sigma + \epsilon} + \beta \tag{8.5}$$

where $\mu = \frac{1}{d_{\text{model}}} \sum_{j=1}^{d_{\text{model}}} x_j$, $\sigma^2 = \frac{1}{d_{\text{model}}} \sum_{j=1}^{d_{\text{model}}} (x_j - \mu)^2$, and $\gamma, \beta \in \mathbb{R}^{d_{\text{model}}}$ are learnable scale and shift parameters. The small constant ϵ (typically 10^{-6}) prevents division by zero. Unlike batch normalization — which normalizes across the batch dimension and is ill-suited to variable-length sequences — layer normalization computes statistics independently for each token, making it sequence-length independent and applicable to both training and inference without reference to batch statistics.

The original Transformer of Vaswani et al. [2017] applied layer normalization after the residual addition (Post-LN): $\text{LayerNorm}(\mathbf{x} + \text{Sublayer}(\mathbf{x}))$. Subsequent work, particularly Xiong et al. [2020], showed that placing the normalization before the sublayer (Pre-LN) produces substantially more stable training, especially for deep models:

$$\mathbf{x} + \text{Sublayer}(\text{LayerNorm}(\mathbf{x})) \quad \text{(Pre-LN)} \tag{8.6}$$

The switch from Post-LN to Pre-LN is one of those quiet engineering decisions that rarely makes headlines but dramatically affects whether a model trains at all — the stability improvement comes from the interaction between residual connections and normalization. In Post-LN, the residual stream accumulates unnormalized updates from every layer; after N layers, the unnormalized residual stream can have very large magnitude, and a single layer normalization applied at the end of each block must handle these large values. In Pre-LN, the normalization is applied before the sublayer, so the sublayer always receives normalized input with controlled magnitude, and the residual stream on the main path remains unnormalized but is not subject to runaway accumulation. Empirically, Pre-LN models converge without the careful learning rate warmup that Post-LN requires, making them more robust to hyperparameter choices. Modern Transformer implementations (GPT-2 and beyond) universally use Pre-LN.

8.5.4 The Feed-Forward Sublayer

The position-wise feed-forward network $\text{FFN}(\mathbf{x}) = \max(0, \mathbf{x}\mathbf{W}_1 + \mathbf{b}_1)\mathbf{W}_2 + \mathbf{b}_2$ is applied identically to each token position — and it is both simpler and more important than it first appears. The word "position-wise" is crucial: the FFN processes each position independently, with no interaction between positions (all inter-position interaction having occurred in the preceding attention sublayer). The FFN consists of two linear transformations with a ReLU nonlinearity between them:

$$\text{FFN}(\mathbf{x}) = \max(0, \mathbf{x}\mathbf{W}_1 + \mathbf{b}_1)\mathbf{W}_2 + \mathbf{b}_2 \tag{8.7}$$

where $\mathbf{W}_1 \in \mathbb{R}^{d_{\text{model}} \times d_{ff}}$, $\mathbf{W}_2 \in \mathbb{R}^{d_{ff} \times d_{\text{model}}}$, and d_{ff} is the inner dimension of the feed-forward network, typically set to $4 \times d_{\text{model}}$ (e.g., $d_{ff} = 2048$ for $d_{\text{model}} = 512$). The FFN first expands the representation from d_{model} to d_{ff} dimensions (the expansion step), applies the ReLU nonlinearity, and then projects back to d_{model} (the compression step). The expansion factor of 4 is a design choice of Vaswani et al. [2017] that has been retained in most subsequent architectures. The FFN serves two complementary purposes. First, it provides per-position nonlinear transformation: attention is a linear operation (a weighted average), and without the FFN, a stack of attention layers would be equivalent to a single layer with deeper attention — the overall mapping would still be linear.

The ReLU nonlinearity enables the model to compute complex per-position functions that pure attention cannot represent. Second, the FFN's large inner dimension provides the majority of the model's parameter count and is believed to be the primary storage site for factual knowledge in large language models.

To see the parameter count: each FFN layer has $d_{\text{model}} \times d_{ff} + d_{ff} \times d_{\text{model}} = 2 \times d_{\text{model}} \times d_{ff}$ weight parameters plus biases. For $d_{\text{model}} = 512$ and $d_{ff} = 2048$, this is $2 \times 512 \times 2048 = 2{,}097{,}152$ parameters — twice as many as the attention sublayer's 1,048,576 parameters. The FFN, far from being an afterthought, accounts for approximately two-thirds of the parameters in each Transformer block. Modern variants of the FFN, including the SwiGLU activation [Shazeer, 2020] used in LLaMA and PaLM, replace the ReLU with a gated linear unit that provides a smoother activation and consistently outperforms the original ReLU on language modeling benchmarks.

8.5.5 Putting It All Together: The Transformer Block

How do self-attention, residual connections, layer normalization, and the feed-forward network combine into a single, stackable building block?

The complete Transformer encoder block (Pre-LN variant) processes input $\mathbf{X} \in \mathbb{R}^{T \times d_{\text{model}}}$ through two sublayers in sequence. In the first sublayer, layer normalization is applied, followed by multi-head self-attention, with the input added back via the residual connection: $\mathbf{X}' = \mathbf{X} + \text{MHA}(\text{LayerNorm}(\mathbf{X}))$. In the second sublayer, layer normalization is applied to $\mathbf{X}'$, followed by the feed-forward network, with another residual addition: $\mathbf{X}'' = \mathbf{X}' + \text{FFN}(\text{LayerNorm}(\mathbf{X}'))$. The output $\mathbf{X}'' \in \mathbb{R}^{T \times d_{\text{model}}}$ has precisely the same shape as the input $\mathbf{X}$, making blocks stackable without any adapters or reshaping operations. This shape-preserving property is the design feature that enables the Transformer's depth: one can stack N identical blocks, each refining the representation from the previous block, without any changes to the block architecture. The original Transformer used $N = 6$ blocks in both the encoder and decoder. Modern LLMs use $N = 32$ (LLaMA-7B), $N = 80$ (GPT-4, estimated), or more.

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class TransformerBlock(nn.Module):
    def __init__(self, d_model, h, d_ff):
        super().__init__()
        self.norm1 = nn.LayerNorm(d_model)
        self.attn = MultiHeadAttention(d_model, h)    # from Code Example 2
        self.norm2 = nn.LayerNorm(d_model)
        self.ff1 = nn.Linear(d_model, d_ff)
        self.ff2 = nn.Linear(d_ff, d_model)

    def forward(self, x, mask=None):
        # Sublayer 1: Pre-LN self-attention with residual
        attn_out, _ = self.attn(self.norm1(x), mask=mask)
        x = x + attn_out
        # Sublayer 2: Pre-LN feed-forward with residual
        ff_out = self.ff2(F.relu(self.ff1(self.norm2(x))))
```

```
        x = x + ff_out
        return x

block = TransformerBlock(d_model=32, h=4, d_ff=128)
x = torch.randn(1, 5, 32)              # (batch, seq_len, d_model)
out = block(x)
print(f"Input shape:  {x.shape}")      # [1, 5, 32]
print(f"Output shape: {out.shape}")    # [1, 5, 32]  -- shape preserved
```

8.5.6 Stacking Blocks: Depth and Capacity

What does each successive Transformer block contribute, and what is the effect of increasing depth?

When N Transformer blocks are stacked, each block builds on the representations produced by the previous block, progressively refining them toward the representations needed for the final prediction task. This hierarchical refinement has been empirically studied through probing experiments that train linear classifiers to predict linguistic properties from the hidden states at each layer. Tenney et al. [2019] found that BERT's lower layers (1–4) encode basic syntactic features such as part-of-speech tags and constituent labels, while higher layers (8–12) encode semantic features such as coreference and semantic role labels. This layered organization mirrors the classical NLP pipeline — morphology, syntax, semantics — emerging spontaneously from the training objective rather than being explicitly encoded in the architecture. The depth-width tradeoff is an important architectural design choice: for a fixed total parameter budget, one can choose a narrow, deep model (many layers, small d_{model}) or a wide, shallow model (few layers, large d_{model}). Deep models tend to learn more hierarchical representations and generalize better on complex tasks, while wide models can capture more information per layer but may not learn the kind of compositional structure that deep stacking enables. Empirically, modern LLMs favor a balanced tradeoff, with depth and width both increasing proportionally as the model scales.

The full Transformer block, with its two sublayers and residual connections, is now complete. In the next section, we show how different configurations of this block — with or without causal masking, with or without cross-attention — give rise to the three major families of Transformer-based models.

8.6 Encoder, Decoder, and Encoder-Decoder Variants

Given the same Transformer building blocks, what determines whether a model is suited for understanding, generation, or structured transformation?

The Transformer block of Section 8.5 is a versatile building block that can be configured in three distinct ways to produce three major model families. The configuration choices — whether to apply the causal mask, and whether to include a cross-attention sublayer — determine the model's ability to model context bidirectionally, generate text autoregressively, and condition on a separate input sequence. Each configuration maps to a distinct pre-training paradigm and a distinct set of downstream tasks, as we will see in detail in Chapter 9.

8.6.1 Encoder-Only Transformers

What happens when we remove the causal mask and allow every position to attend to every other position?

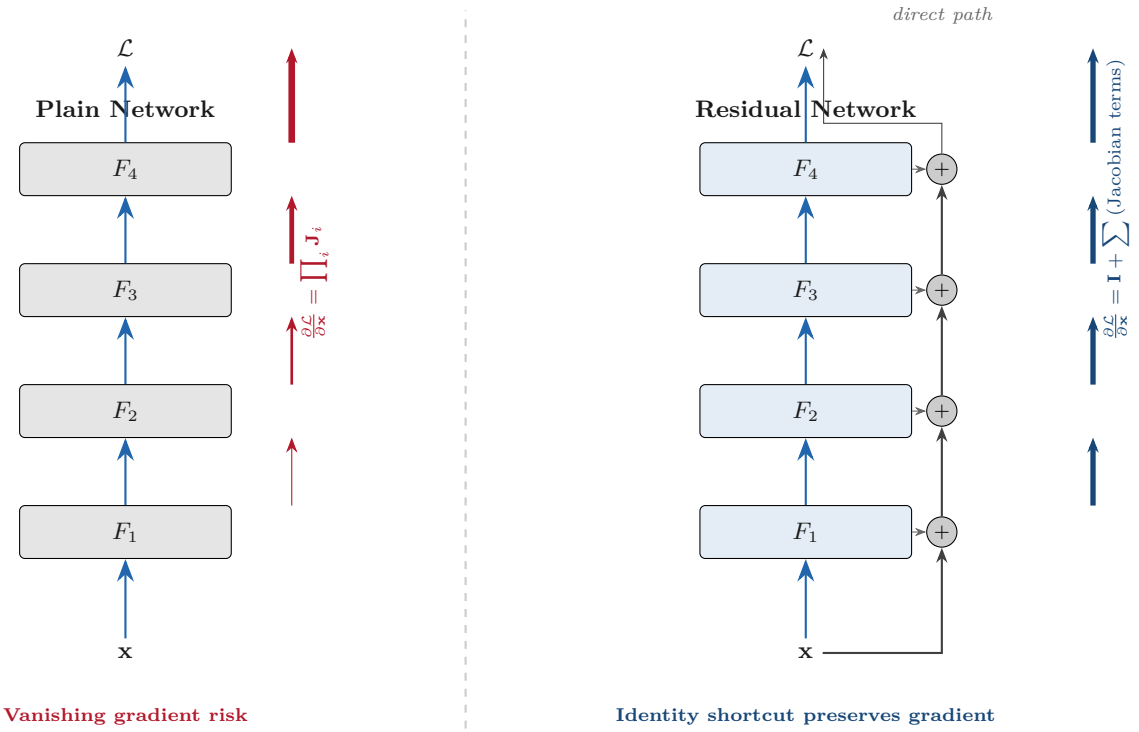

Figure 6: Figure 8.6 – Residual Connections and Gradient Flow

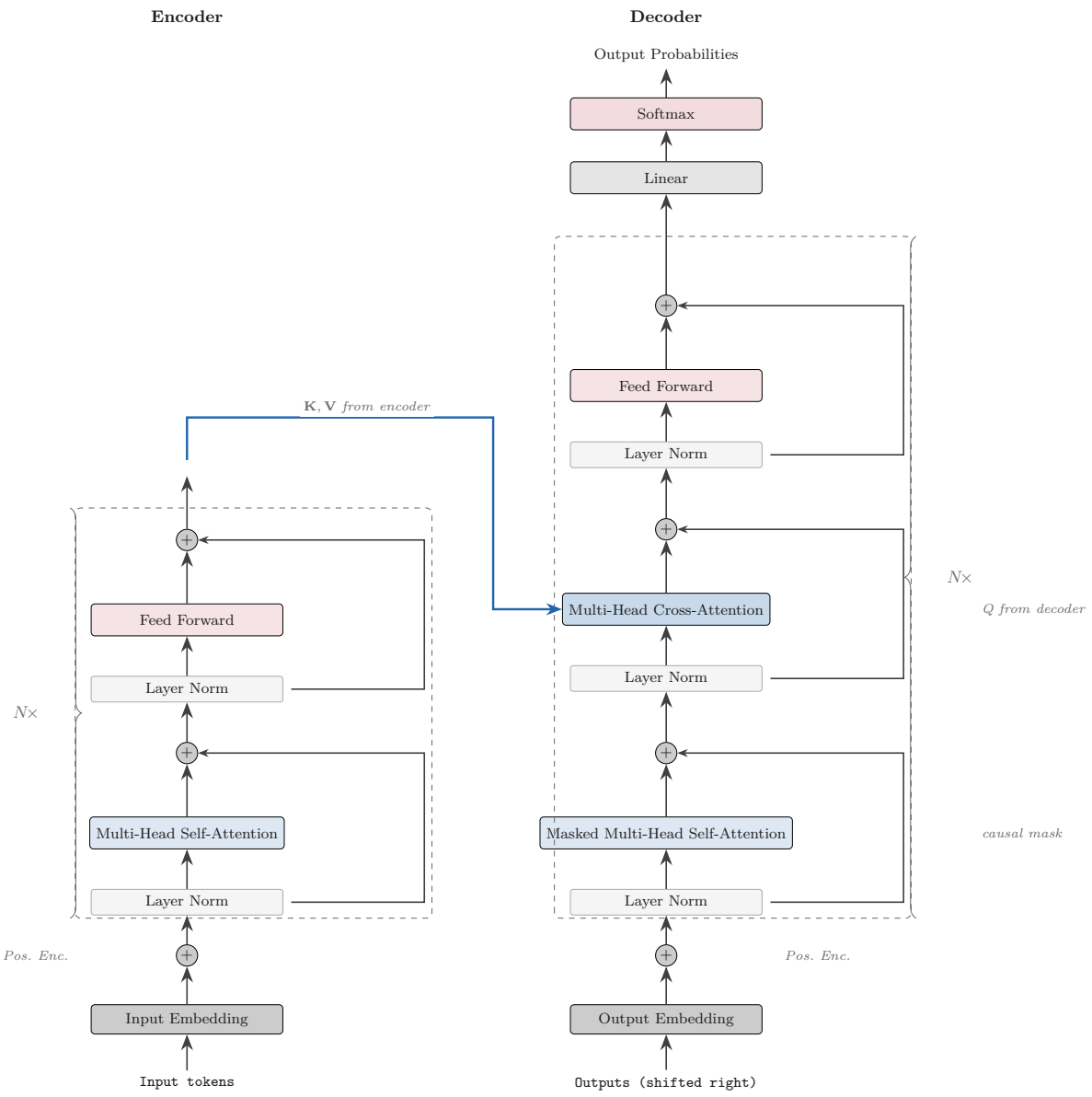

Figure 7: Figure 8.7 – Full Transformer Architecture

An encoder-only Transformer applies N stacked Transformer blocks with fully bidirectional self-attention: every position can attend to every other position in both directions, with no causal mask. The result is a set of contextual token representations where the representation of each token has been influenced by all other tokens in the sequence. This bidirectional context is ideal for tasks that require understanding of the full input — sentiment classification, named entity recognition, question answering, and natural language inference — because these tasks benefit from knowing what comes after a token as much as what came before. BERT [Devlin et al., 2019] exemplifies this architecture, using 12 encoder blocks with $d_{\text{model}} = 768$ and $h = 12$. For classification tasks, BERT appends a special [CLS] token to the input; its representation after the final encoder block is used as the sequence-level representation and passed to a classification head. For token-level tasks such as NER or span extraction, the representation of each token's final-layer state is used directly. A natural but incorrect intuition is that encoder-only models can generate text. They cannot naturally: the bidirectional attention that makes encoders powerful for understanding also prevents them from generating text left-to-right, because each token's representation depends on future tokens that would not yet be generated at inference time.

8.6.2 Decoder-Only Transformers

How does applying the causal mask produce a model that can generate text autoregressively?

A decoder-only Transformer applies N stacked Transformer blocks with causal (masked) self-attention: each position can attend to itself and all preceding positions, but not to any following positions. This enforces the autoregressive factorization: the model's prediction at position t depends only on positions $1, 2, \ldots, t-1$, exactly as required for left-to-right text generation. The GPT family of models exemplifies this architecture: GPT-2 uses 12 to 48 decoder blocks with d_{model} ranging from 768 to 1600, and GPT-3 uses 96 decoder blocks with $d_{\text{model}} = 12288$. Training a decoder-only model requires only the standard autoregressive language modeling objective — predicting the next token at each position — with the causal mask ensuring that no position sees future tokens during training. At inference time, text is generated by sampling from the model's output distribution at each position and appending the sampled token to the context. The decoder-only architecture has become the dominant paradigm for modern large language models because autoregressive generation is a general-purpose capability: through in-context learning (Chapter 13), a decoder-only model can perform classification, question answering, translation, summarization, and virtually any NLP task by reformulating it as a text completion problem.

8.6.3 Encoder-Decoder Transformers

What additional mechanism allows a decoder to condition its generation on a separately encoded input?

The full encoder-decoder Transformer — the original architecture of Vaswani et al. [2017] — combines an encoder stack and a decoder stack connected by cross-attention. The encoder applies N blocks of bidirectional self-attention to the input sequence, producing a set of encoder hidden states $\mathbf{H}^{\text{enc}} \in \mathbb{R}^{T_{\text{src}} \times d_{\text{model}}}$, one per source token. The decoder applies N blocks of a modified architecture that includes three sublayers rather than two: (1) causal self-attention (with the causal mask, attending only to previous decoder positions), (2) cross-attention to the encoder output, and (3) a position-wise FFN. In the cross-attention sublayer, the queries come from the decoder's current state and the keys and values come from the encoder hidden states: $\text{CrossAttention}(\mathbf{Q}_{\text{dec}}, \mathbf{K}_{\text{enc}}, \mathbf{V}_{\text{enc}})$. This allows each decoder position to attend to all encoder positions while remaining causally masked

from future decoder positions. The cross-attention mechanism is the Transformer's implementation of the encoder-decoder attention introduced in Chapter 6: instead of computing attention scores as additive (Bahdanau) or multiplicative (Luong) compatibility between decoder and encoder states, the Transformer uses scaled dot-product attention between projected decoder queries and encoder keys.

T5 [Raffel et al., 2020] uses the encoder-decoder architecture with a text-to-text formulation: both input and output are text strings, and all NLP tasks are converted to this format. Translation becomes "translate English to German: [source text] → [target text]," summarization becomes "summarize: [document] → summary," and classification becomes "classify: [text] → [label]." The encoder-decoder architecture is well-suited for structured input-output tasks where the input and output are clearly distinct sequences of potentially different lengths — translation, summarization, and speech recognition are canonical examples.

8.6.4 The Causal Mask

The causal mask for a sequence of length T is a $T \times T$ lower-triangular binary matrix $\mathbf{M}$, where $\mathbf{M}_{ij} = 1$ if $j \leq i$ (position i can attend to position j) and $\mathbf{M}_{ij} = 0$ otherwise. For $T = 5$:

$$\mathbf{M} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

In the attention computation, masked positions (where $\mathbf{M}_{ij} = 0$) are set to $-\infty$ before the softmax, which maps them to zero attention weight. The causal mask is created once for a given sequence length and reused across all layers and all attention heads in the decoder. It is a fixed, deterministic quantity determined entirely by the sequence length; there are no learned parameters in the mask. The mask enforces the autoregressive factorization: the model's output at position i can depend on positions $1, 2, \ldots, i$ but not on $i+1, \ldots, T$. During training with teacher forcing, all T positions are processed simultaneously, and the mask ensures that this does not constitute information leakage from the future. During inference, the mask is automatically satisfied because future tokens have not yet been generated — but applying it explicitly during inference (as some implementations do) makes no difference to the output.

8.6.5 Choosing an Architecture

Given the three architectural variants, which should be used for a given task or application?

The choice among encoder-only, decoder-only, and encoder-decoder architectures reflects the task structure and the intended pre-training strategy. Before examining these variants, it is worth noting that the architecture's input—the tokens themselves—is not architecture-specific but rather a separate design choice. The choice of tokenization, how we segment text into the tokens that the Transformer processes, has significant implications for model performance and efficiency, as we discuss in detail in Chapter 10. Table 8.2 summarizes the main considerations for architecture selection.

Decoder Block with Causal Mask

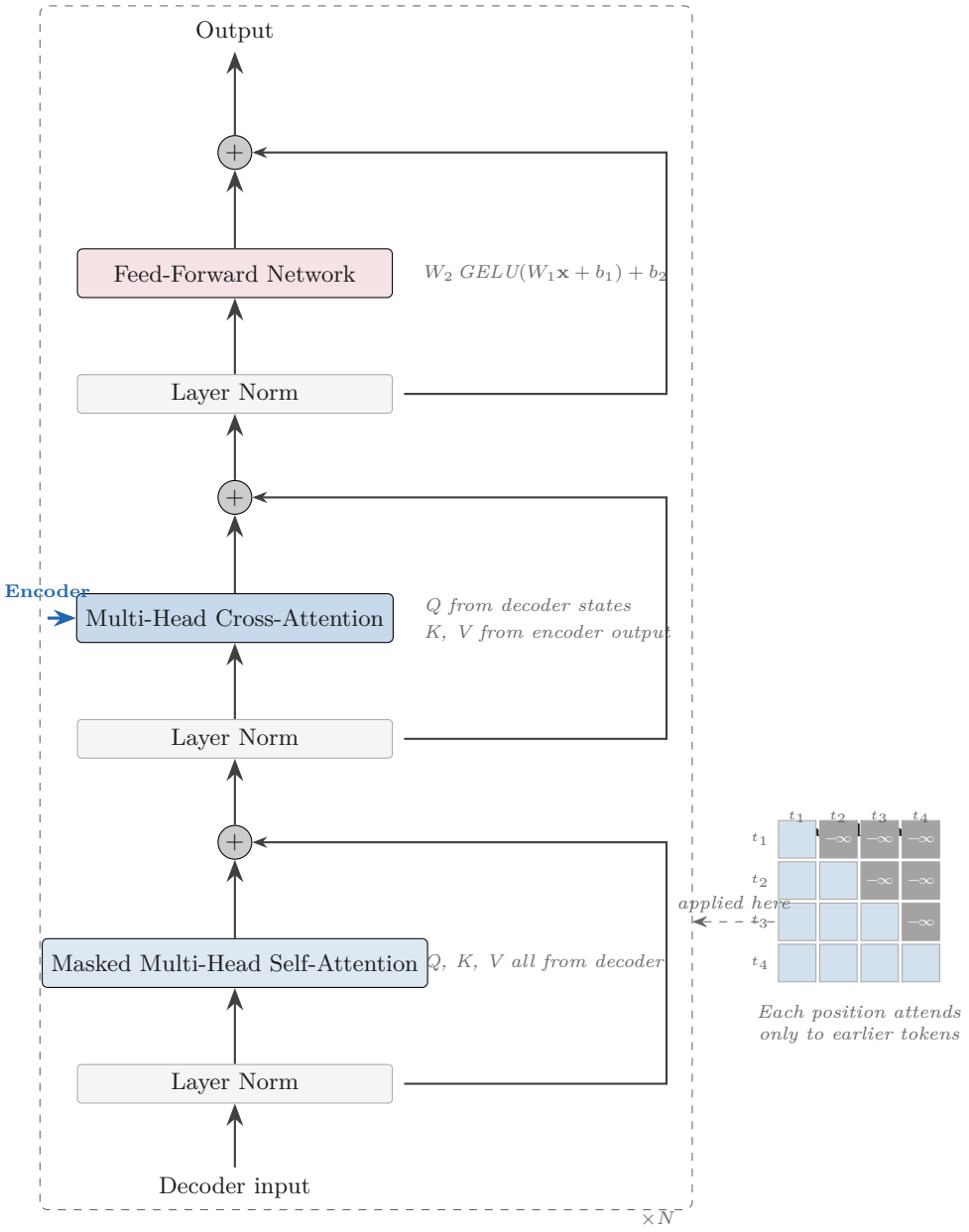

Figure 8: Figure 8.8 – Decoder Block with Masking

Architecture	Directionality	Causal Mask	Cross-Attention	Best For	Pre-training (Ch 9)
Encoder-only	Bidirectional	No	No	Understanding (classification, NER, QA)	Masked LM (BERT)
Decoder-only	Causal	Yes	No	Generation, general-purpose	Causal LM (GPT)
Encoder-decoder	Mixed	Decoder only	Yes	Conditional generation	Span corruption (T5)

The field has undergone a notable convergence: while encoder-only and encoder-decoder models dominated the 2019–2021 period (the era of BERT and T5), the 2022–2025 period has seen decoder-only models become dominant for general-purpose AI systems. The reasons are several. First, autoregressive generation is a maximally general capability: any understanding task can be reformulated as text completion. Second, decoder-only models scale more efficiently on the next-token prediction objective. Third, in-context learning — the ability to perform novel tasks from examples provided in the prompt — is a property of decoder-only models trained autoregressively at scale. The encoder-decoder architecture remains the preferred choice for structured input-output tasks with clear source/target separation, particularly in translation and structured prediction. Chapter 9 will formalize the connections between architectural variants and pre-training objectives, showing how the architecture choice determines which self-supervised objective is most appropriate.

If we had to choose a single architectural lesson from this chapter, it would be this: the Transformer succeeds not because of any one component but because every component — attention, normalization, residuals, feed-forward layers — addresses a specific failure mode that would otherwise prevent learning. The architecture is a careful assembly of targeted solutions, and understanding why each piece is necessary is more instructive than memorizing any particular configuration.

We have now assembled the complete Transformer architecture: scaled dot-product attention, multi-head attention, positional encodings, feed-forward sublayers, residual connections, layer normalization, and the three architectural variants. In the next chapter, we show how these architectural blueprints are filled in by pre-training on large text corpora to produce the foundation models that power modern NLP.

Summary

Chapter 8 has presented the Transformer architecture from first principles. Section 8.1 established the motivation: recurrent networks process sequences sequentially, and attention cannot fix this within the RNN framework, but replacing recurrence with self-attention enables full parallelization. Section 8.2 derived scaled dot-product attention, showing how $\mathbf{Q}$, $\mathbf{K}$, $\mathbf{V}$ matrices are computed by linear projection, how the $1/\sqrt{d_k}$ scaling factor prevents softmax saturation, how the causal mask enforces autoregressive factorization, and how the full computation flows from a $T \times d_{\text{model}}$ input

to a $T \times d_v$ output. Section 8.3 introduced multi-head attention: h parallel attention functions in lower-dimensional subspaces, concatenated and projected to produce a rich, multi-relational contextual representation with $4d_{\text{model}}^2$ parameters per layer. Section 8.4 addressed permutation equivariance, presenting four positional encoding strategies — sinusoidal, learned, RoPE, and ALiBi — and their tradeoffs in extrapolation, parameter efficiency, and relative-position encoding. Section 8.5 assembled the full Transformer block: Pre-LN layer normalization, multi-head self-attention, residual connection, Pre-LN layer normalization, feed-forward network, and residual connection, all shape-preserving to enable arbitrary stacking. Section 8.6 mapped the three architectural variants — encoder-only, decoder-only, and encoder-decoder — to their respective task families and pre-training objectives, previewing Chapter 9.

Looking Ahead

We have built the Transformer from the ground up, deriving each component from first principles and assembling them into a complete, stackable building block. But the architecture alone is not the full story. How do we train a Transformer on raw text to produce useful representations? What objective should we optimize? In the next chapter, we explore the pre-training paradigms — masked language modeling, causal language modeling, and text-to-text learning — that transform the Transformer architecture into the foundation models that power modern NLP.

The Transformer architecture provides a natural transition to Chapter 9, where we explore how this architecture enables large-scale pre-training paradigms—BERT, GPT, and T5—that have reshaped the field.

Exercises

Exercise 8.1 (Theory, Basic)

Without the $1/\sqrt{d_k}$ scaling factor, the dot product $\mathbf{q}^\top \mathbf{k}$ has variance that grows with d_k. Prove this formally. Assume that the components q_i and k_i are independent random variables with mean 0 and variance 1 for $i = 1, \ldots, d_k$. Compute $\mathbb{E}[\mathbf{q}^\top \mathbf{k}]$ and $\text{Var}(\mathbf{q}^\top \mathbf{k})$. Then show that dividing by $\sqrt{d_k}$ normalizes the variance to 1, and explain in one paragraph why variance d_k leads to softmax saturation while variance 1 does not.

Hint: Write $\mathbf{q}^\top \mathbf{k} = \sum_{i=1}^{d_k} q_i k_i$. Each term $q_i k_i$ has mean $\mathbb{E}[q_i]\mathbb{E}[k_i] = 0$ and variance $\text{Var}(q_i k_i) = \text{Var}(q_i)\text{Var}(k_i) = 1$ for independent zero-mean unit-variance variables. Since the terms are independent, the variance of the sum equals the sum of variances.

Exercise 8.2 (Theory, Intermediate)

Compute the total parameter count of a Transformer with $d_{\text{model}} = 512$, $h = 8$, $d_{ff} = 2048$, $N = 6$ layers, and vocabulary size $|V| = 32000$. Assume weight tying between input embeddings and the output projection. Break the count into (a) the embedding layer, (b) attention parameters per layer, (c) FFN parameters per layer, and (d) layer normalization parameters per layer. Report the grand total.

Hint: Attention per layer: $4 \times d_{\text{model}}^2$ (three input projections plus output projection). FFN per layer: $2 \times d_{\text{model}} \times d_{ff}$. Layer norm per sublayer: $2 \times d_{\text{model}}$ (scale γ and shift β). Embedding:

$|V| \times d_{\text{model}}$. With weight tying, the output layer shares this embedding matrix.

Exercise 8.3 (Theory, Intermediate)

Show that the sinusoidal positional encoding allows the model to represent relative position via a fixed linear transformation. For a single frequency ω, show that the vector $[\text{PE}(pos+k, 2i), \text{PE}(pos+k, 2i+1)]$ can be expressed as a 2×2 matrix multiplication applied to $[\text{PE}(pos, 2i), \text{PE}(pos, 2i+1)]$. Write down the explicit matrix and verify that it is a rotation matrix (orthogonal, determinant $+1$) that depends only on k and not on pos.

Hint: Use the angle addition formulas $\sin(\alpha+\beta) = \sin\alpha\cos\beta + \cos\alpha\sin\beta$ and $\cos(\alpha+\beta) = \cos\alpha\cos\beta - \sin\alpha\sin\beta$ with $\alpha = pos \cdot \omega$ and $\beta = k \cdot \omega$.

Exercise 8.4 (Theory, Advanced)

Analyze the computational complexity of multi-head self-attention as a function of sequence length T and model dimension d_{model}. Show that the time complexity is $\mathcal{O}(T^2 d_{\text{model}})$ and the space complexity for the attention matrix is $\mathcal{O}(T^2 h)$. For $T = 4096$ and $d_{\text{model}} = 4096$ with $h = 32$ heads in float16 (2 bytes per value), estimate the memory required to store all h attention matrices in gigabytes. Determine the crossover point at which $T^2 > d_{\text{model}} \cdot T$ (i.e., when the quadratic term dominates the linear term) and interpret this in terms of typical sequence lengths and model sizes.

Hint: The $\mathbf{QK}^\top$ computation requires $\mathcal{O}(T^2 d_k)$ FLOPs per head. The attention matrix has T^2 entries per head and h heads. The quadratic term dominates when $T > d_k$. For $d_k = d_{\text{model}}/h$, the crossover is at $T = d_{\text{model}}/h$.

Exercise 8.5 (Programming, Basic)

Implement scaled dot-product attention from scratch in PyTorch, without using `nn.MultiheadAttention`. Your function should accept query, key, and value tensors of shape `(batch, seq_len, d_k)` and an optional boolean mask tensor of shape `(seq_len, seq_len)`, and return the attention output of shape `(batch, seq_len, d_v)` and the attention weight matrix. Test on a random input with `batch=2`, `seq_len=8`, `d_k=d_v=16`. Verify that each row of the attention weight matrix sums to 1 (use `assert torch.allclose(weights.sum(-1), torch.ones_like(weights.sum(-1)))`). Also test with a causal mask and verify that the upper-triangular weights are exactly zero.

Hint: Use `torch.matmul` for $\mathbf{QK}^\top$, divide by `math.sqrt(d_k)`, apply `F.softmax(dim=-1)`, and multiply by $\mathbf{V}$. For the causal mask, create a lower-triangular matrix with `torch.tril(torch.ones(T, T))` and apply `masked_fill(mask == 0, float('-inf'))` before softmax.

Exercise 8.6 (Programming, Intermediate)

Implement multi-head attention with $h = 4$ heads and $d_{\text{model}} = 32$. Apply it to a sentence encoded as word embeddings — use a lookup table initialized with random embeddings for the vocabulary {"The", "cat", "sat", "on", "the", "comfortable", "mat"}. Visualize the attention patterns for each of the four heads using a heatmap grid (one heatmap per head), with the sentence tokens on both axes. Describe in two to three sentences which heads appear to focus on local patterns (near-diagonal attention) versus global patterns (broadly distributed attention).

Hint: After training a forward pass (no training needed, just a forward pass of randomly initialized attention), use `matplotlib.pyplot.imshow` to display the 7×7 attention weight matrices. For a 2×2 grid of subplots, use `fig, axes = plt.subplots(2, 2)`.

Exercise 8.7 (Programming, Intermediate)

Compute sinusoidal positional encodings for $T = 256$ positions and $d_{\text{model}} = 128$. Produce three visualizations: (1) a heatmap of the full 256×128 PE matrix; (2) a plot of the cosine similarity between PE(0) and PE(k) for $k = 0, 1, \ldots, 128$; and (3) a bar chart showing how similarity decreases with distance. Verify empirically that the cosine similarity between PE(i) and PE(j) depends primarily on $|i - j|$ by computing the average similarity for all pairs with the same absolute offset.

Hint: Cosine similarity between PE(i) and PE(j) is $\text{PE}(i) \cdot \text{PE}(j) / (\|\text{PE}(i)\| \|\text{PE}(j)\|)$. The similarity should decrease roughly monotonically with $|i - j|$ for small distances. Use `torch.nn.functional.cosine_similarity` with `dim=-1`.

Exercise 8.8 (Programming, Intermediate)

Build a complete Transformer encoder block using the Pre-LN configuration: `LayerNorm -> MultiHeadAttention -> residual; LayerNorm -> FFN -> residual`. Use $d_{\text{model}} = 64$, $h = 4$, and $d_{ff} = 256$. Stack four such blocks. Pass a random input tensor of shape `(2, 12, 64)` (batch 2, sequence length 12) through the stack. Verify that (1) the output shape matches the input shape, and (2) gradients flow to all parameters by calling `.backward()` on the output's sum and checking that `all(p.grad is not None for p in model.parameters())`.

Hint: Stack blocks in an `nn.ModuleList`. Use your `MultiHeadAttention` from Exercise 8.6. The shape-preservation property of each block ensures that the output of block ℓ can be fed directly to block $\ell + 1$ without any reshaping.

Exercise 8.9 (Programming, Advanced)

Train a small Transformer language model on a toy corpus. Use 4 decoder blocks with $d_{\text{model}} = 128$, $h = 4$, $d_{ff} = 512$, and learned position embeddings ($T_{\max} = 256$). Train on a 50,000-word sample from WikiText-2 (available via the `datasets` library) using the causal language modeling objective and the AdamW optimizer with learning rate 3×10^{-4}. Report training perplexity after 5 epochs. Generate a sample 50-token continuation of the prompt "The history of machine learning" using greedy decoding. Compare training time and final perplexity with an LSTM language model of comparable parameter count trained on the same corpus.

Hint: Use a causal mask created with `torch.tril(torch.ones(T, T))`. The Transformer should achieve lower perplexity and train faster (per epoch on a GPU) than the LSTM because the self-attention computation is fully parallelizable across the sequence length dimension.

Exercise 8.10 (Programming, Advanced)

Implement RoPE from scratch for a two-head attention module with $d_{\text{model}} = 16$ (so $d_k = 8$ per head). Apply RoPE rotations to the query and key vectors for a sequence of 32 positions using rotation angles $\theta_i = 10000^{-2i/d_k}$ for dimension pair i. Verify the relative position property empirically: compute the attention score between positions $(m, n) = (5, 12)$ and compare it to the score between $(m', n') = (10, 17)$ — both pairs have the same relative offset $n - m = n' - m' = 7$. Show that these scores are equal (or approximately equal, up to floating-point precision) by printing both values and their absolute difference.

Hint: RoPE applies a block-diagonal rotation matrix $\mathbf{R}_\Theta^m$ to the query at position m and $\mathbf{R}_\Theta^n$ to the key at position n. The dot product satisfies $(\mathbf{R}^m \mathbf{q})^\top (\mathbf{R}^n \mathbf{k}) = \mathbf{q}^\top \mathbf{R}^{n-m} \mathbf{k}$, which depends only on $n - m$.

References

Ba, J. L., Kiros, J. R., and Hinton, G. E. (2016). Layer Normalization. *arXiv:1607.06450*.

Clark, K., Khandelwal, U., Levy, O., and Manning, C. D. (2019). What Does BERT Look At? An Analysis of BERT's Attention. In *Proceedings of the ACL Workshop on BlackboxNLP*, pp. 276–286.

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778.

Press, O., Smith, N. A., and Lewis, M. (2022). Train Short, Test Long: Attention with Linear Biases Enables Input Length Extrapolation. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

Shazeer, N. (2020). GLU Variants Improve Transformer. *arXiv:2002.05202*.

Su, J., Lu, Y., Pan, S., Murtadha, A., Wen, B., and Liu, Y. (2021). RoFormer: Enhanced Transformer with Rotary Position Embedding. *arXiv:2104.09864*.

Tenney, I., Das, D., and Pavlick, E. (2019). BERT Rediscovers the Classical NLP Pipeline. In *Proceedings of the Association for Computational Linguistics (ACL)*, pp. 4593–4601.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). Attention Is All You Need. In *Advances in Neural Information Processing Systems (NeurIPS)*, pp. 5998–6008.

Voita, E., Talbot, D., Moiseev, F., Sennrich, R., and Titov, I. (2019). Analyzing Multi-Head Self-Attention: Specialized Heads Do the Heavy Lifting, the Rest Can Be Pruned. In *Proceedings of the Association for Computational Linguistics (ACL)*, pp. 5797–5808.

Xiong, R., Yang, Y., He, J., Zheng, K., Zheng, S., Xing, C., Zhang, H., Lan, Y., Wang, L., and Liu, T. (2020). On Layer Normalization in the Transformer Architecture. In *Proceedings of the International Conference on Machine Learning (ICML)*, pp. 10524–10533.