Predicting the Next Word

From Shannon to ChatGPT

Test Compilation - Chapter 6

Contents

Chapter 6

Transformers: Parallel Attention for Next-Word Prediction

In this chapter, we advance next-word prediction by:

- Introducing attention mechanisms for direct token access
- Eliminating the sequential bottleneck of RNNs
- Learning to focus on relevant context positions
- Enabling parallel processing of entire sequences

6.1 From Sequential Bottleneck to Parallel Attention

The recurrent neural networks examined in Chapter **??** process sequences one token at a time, compressing all previous context into a fixed-dimensional hidden state $\mathbf{h}$. This sequential processing introduces two fundamental limitations that constrain model capacity and training efficiency. First, it creates a computational bottleneck: we cannot predict $w[t]$ until we have computed $\mathbf{h}[t-1]$, which requires computing $\mathbf{h}[t-2]$, and so on back to the start of the sequence. This dependency chain has $O(T)$ sequential depth for a sequence of length T, preventing parallelization across positions during training and inference. Second, it creates an information bottleneck: the fixed-size hidden state must summarize arbitrarily long histories, forcing the model to discard or compress information as sequences grow longer. The information capacity of $\mathbf{h}$ is bounded by its dimension d_{model}, regardless of sequence length. When predicting the missing word in "The software engineer who had been working remotely from the mountain cabin for several months finally submitted the code that would _____," an RNN must compress the entire 22-word context into a single vector before making its prediction. This compression inevitably loses fine-grained information about specific earlier words that might be crucial for prediction, particularly for long-range syntactic or semantic dependencies. The question naturally arises: can we design an architecture that accesses context directly, without sequential processing or fixed-size compression?

The transformer architecture answers this question by replacing recurrence with *attention*. Instead of processing tokens sequentially and compressing context into a hidden state, transformers allow each position to directly attend to all previous positions in parallel. When predicting position 23 in our running example, the model can simultaneously look back at positions 1 through 22, computing a weighted combination of their representations with computational complexity $O(T^2 \cdot d_{\text{model}})$ but only $O(1)$ sequential depth. The weights in this combination are not fixed but learned: the model learns which previous positions are most relevant for predicting each next word through gradient-based optimization. This attention mechanism eliminates both bottlenecks simultaneously. The computational bottleneck vanishes because all positions can be processed in parallel rather than sequentially, enabling efficient utilization of GPU tensor cores designed for matrix oper-

ations. The information bottleneck vanishes because we maintain separate representations for each position rather than compressing everything into a single vector, preserving $O(T \cdot d_{\text{model}})$ bits of information. The price we pay is increased memory usage: instead of a single hidden state, we now maintain representations for all positions, requiring quadratic memory in sequence length for attention weight storage. However, modern hardware architectures with high-bandwidth memory excel at this parallel computation pattern, making the trade-off favorable for sequences up to several thousand tokens.

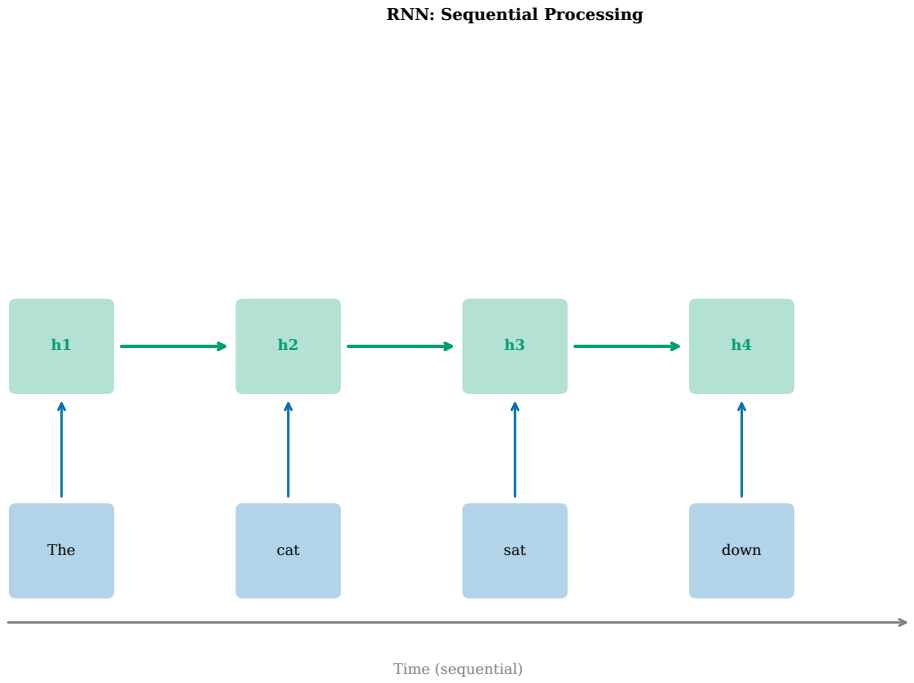


Figure 6.1: Sequential processing in RNNs versus parallel processing in transformers. The RNN (left) computes hidden states sequentially, creating a computational bottleneck where each step depends on the previous. The transformer (right) processes all positions simultaneously, with attention connections allowing direct communication between any pair of positions.

Figure 6.1 contrasts these two processing paradigms. The RNN's sequential chain means that information from position 1 must pass through 21 intermediate steps to influence the prediction at position 23, with each step applying nonlinear transformations that can distort or lose the original signal. Each step risks degrading or losing that information through gradient vanishing (where gradients shrink exponentially with distance, scaling as λ^{21} for eigenvalue $\lambda < 1$), nonlinear transformations that saturate or compress, or competition with new incoming information that overwrites earlier representations. The transformer's parallel structure allows position 23 to directly access position 1 through a learned attention weight, bypassing all intermediate positions with a path length of exactly one. This direct access proves particularly valuable for long-range dependencies, where critical context might appear many positions earlier than the prediction target. The attention mechanism computes these weights dynamically based on the content at each position through query-key dot products, not just their distance or fixed positional biases. Unlike fixed positional biases that always attend more to nearby words regardless of semantic relevance, learned attention can focus on a distant but relevant token while ignoring many nearby but irrelevant ones, adapting to the specific requirements of each prediction. This content-based routing of information is the key innovation that enables transformers to handle complex linguistic structures including nested clauses, long-distance agreement, and coreference resolution.

The information bottleneck illustrated in Figure 6.2 becomes increasingly severe as sequences lengthen, following information-theoretic constraints on representation capacity. An RNN with a 512-dimensional hidden state must compress a 50-token sequence (potentially 50 different embeddings, each of dimension 512) into that same 512-dimensional vector, achieving a compression ratio of 50:1 that necessarily discards information. Some information loss is inevitable: the hidden state cannot represent all 25,600 numbers (50 times 512)

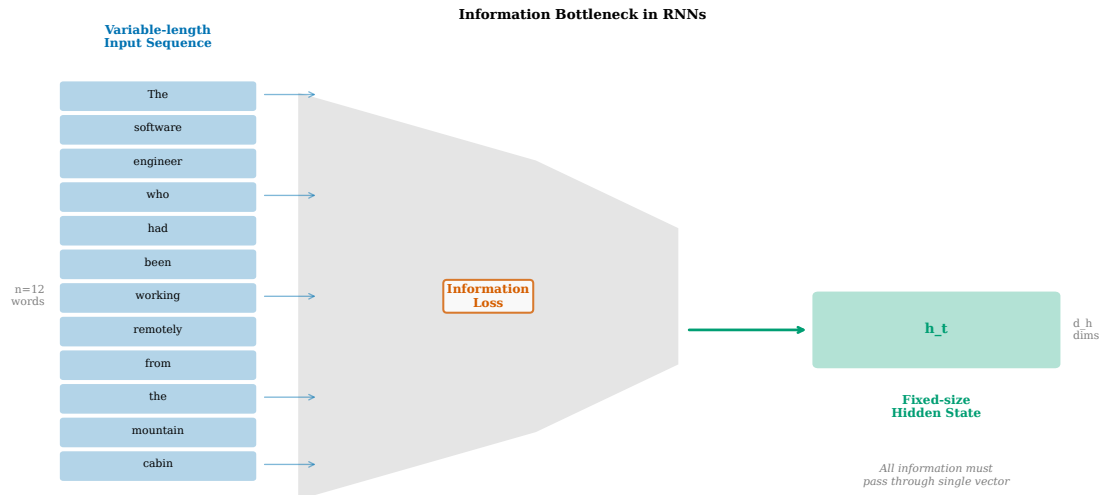


Figure 6.2: The information bottleneck in RNNs. As sequence length increases (x-axis), the amount of information that must pass through the fixed-size hidden state $\mathbf{h}$ grows, but the state's capacity remains constant. Transformers avoid this bottleneck by maintaining separate representations for all positions.

without lossy compression, and the mutual information $I(\mathbf{h}[T]; w[1:T])$ is bounded by the entropy of $\mathbf{h}[T]$. Transformers sidestep this constraint by keeping all 50 embeddings separate and using attention to access them selectively, maintaining the full 25,600-dimensional joint representation. When predicting the next word, the model need not have compressed "software engineer" and "mountain cabin" into a single representation that trades off between them. Instead, it can attend to the embedding of "engineer" directly when determining subject-verb agreement, and attend to "cabin" when considering location-related predictions, with independent attention weights for each aspect. The memory cost is higher, storing all 50 position representations requiring $O(T \cdot d_{\text{model}})$ space, but modern GPUs with tensor cores excel at parallel operations on large matrices, making this trade-off favorable in practice for sequences up to context window limits. The information capacity scales linearly with sequence length rather than being fixed, enabling transformers to preserve fine-grained distinctions that RNNs must sacrifice.

Figure 6.3 provides an intuitive view of the attention mechanism as a soft, differentiable memory lookup operation. We can think of attention as a differentiable version of a key-value store or dictionary lookup, where hard selection is replaced by soft weighting. In a traditional dictionary, we have exact keys and look up associated values with $O(1)$ lookup but no gradient signal. In attention, we have a query (the question we are asking, such as "what word comes next?"), a set of keys (representations of what each previous position contains or offers), and a set of values (the actual information to retrieve from each position for downstream computation). We compute how well the query matches each key using dot product similarity $\mathbf{q}^\top \mathbf{k}$, producing similarity scores that measure alignment in the learned embedding space. These scores are normalized into a probability distribution using the softmax function, giving us attention weights α that sum to one and are non-negative. Finally, we compute a weighted average of the values using these weights, producing an output that lies in the convex hull of the value vectors. The crucial insight is that all of this is differentiable end-to-end, so the model can learn what queries to ask, what keys to provide, and what values to return through gradient descent on the next-word prediction loss. The attention weights adapt based on content similarity, not just position, enabling dynamic context-dependent information routing.

Our running example in Figure 6.4 demonstrates why direct access matters for capturing distributed linguistic constraints. The sentence "The software engineer who had been working remotely from the mountain cabin for several months finally submitted the code that would _____" contains multiple relevant signals distributed across 22 positions that interact to constrain the prediction. Position 3 ("engineer") constrains the subject and thus influences verb choices through subject-verb agreement, requiring singular third-person forms. Position

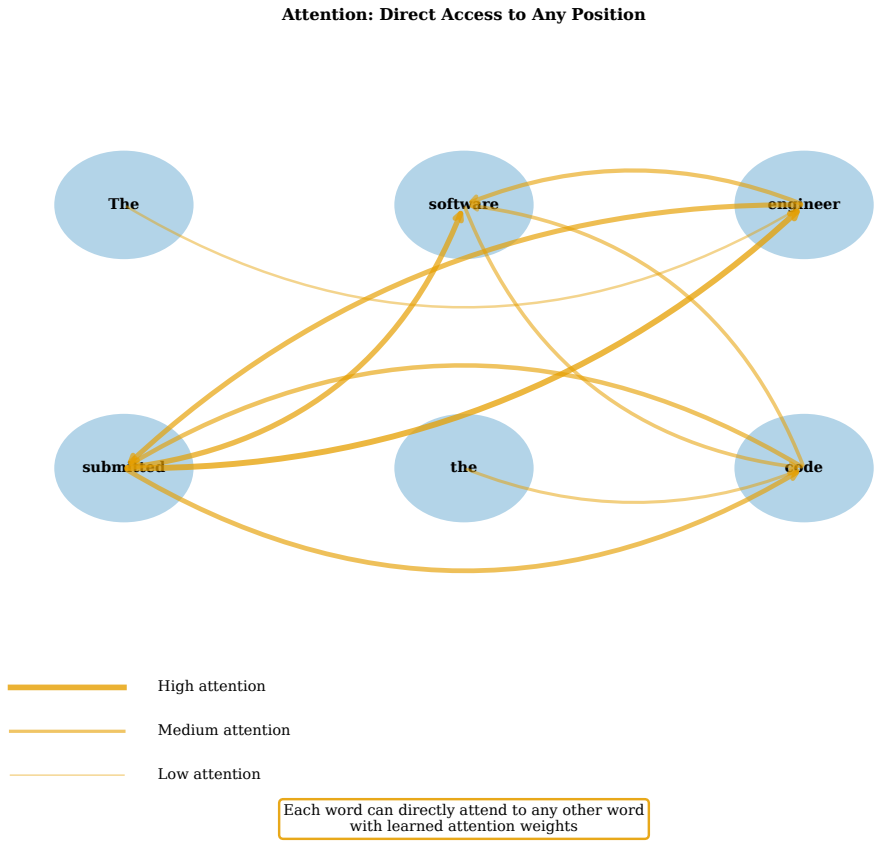


Figure 6.3: Intuitive view of attention as a soft lookup mechanism. Given a query (what we're trying to predict), we compute similarity scores with all keys (previous positions), convert these to weights via softmax, and return a weighted combination of values. The weights are shown as connection thickness.

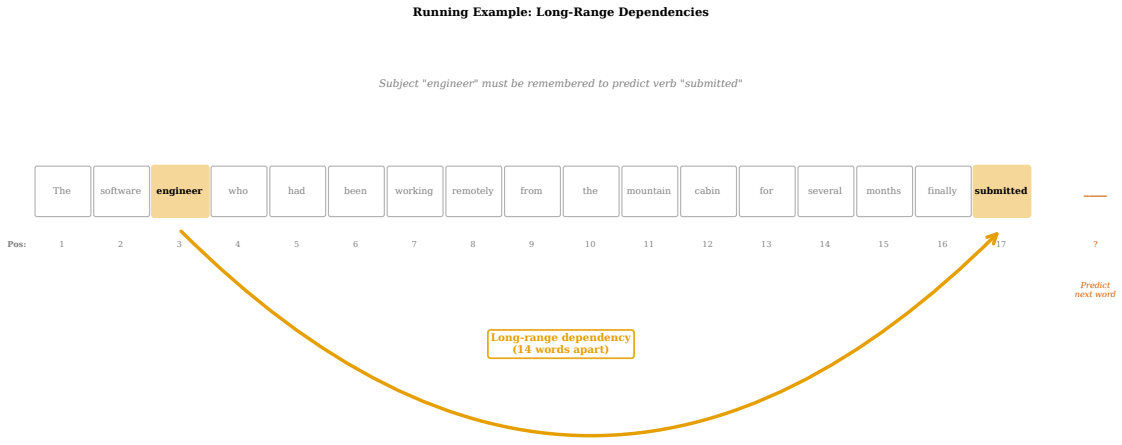


Figure 6.4: Our running example sentence with 22 context words. When predicting position 23, the model can attend to any previous position. Relevant words for subject-verb agreement ("engineer", "submitted") and semantic constraints ("code", "software") may be scattered throughout the context.

2 ("software") and position 21 ("code") establish a programming context through semantic coherence, making completions like "compile," "execute," or "solve" more plausible than unrelated verbs by shifting probability mass toward the technical domain. Position 19 ("submitted") indicates a completed action in past tense, potentially influencing tense or aspect in the continuation and establishing temporal sequencing. An RNN must somehow encode all these signals into its hidden state at position 22, balancing their relative importance through learned gating mechanisms that face the credit assignment problem across many timesteps. A transformer can learn to attend strongly to positions 2, 3, 19, and 21 when computing the representation for position 23, with explicit attention weights $\alpha[23, j]$ quantifying each position's contribution. The attention weights reveal which context the model considers relevant, providing interpretability alongside performance and enabling analysis of model decision-making through visualization of the attention matrix.

6.2 The Attention Mechanism

The attention mechanism transforms the informal intuition of "looking at relevant context" into a precise mathematical operation with well-defined gradients for learning. Given a sequence of token embeddings $\mathbf{e}[1]$, $\mathbf{e}[2]$, $\ldots$, $\mathbf{e}[n]$, where each $\mathbf{e}[i]$ has dimension d_{model}, we first project each embedding into three different spaces: queries $\mathbf{Q}$, keys $\mathbf{K}$, and values $\mathbf{V}$. These projections are learned linear transformations parameterized by weight matrices $W^Q, W^K, W^V \in \mathbb{R}^{d_{\text{model}} \times d_{\text{model}}}$, contributing $3 \cdot d_{\text{model}}^2$ parameters per attention layer. For position i, we compute the query vector $\mathbf{q}[i] = \mathbf{e}[i]W^Q$, the key vector $\mathbf{k}[i] = \mathbf{e}[i]W^K$, and the value vector $\mathbf{v}[i] = \mathbf{e}[i]W^V$ through matrix multiplication. The query represents what position i is asking for or searching for in the context, the key represents what position i contains or offers to other positions, and the value represents the actual information to retrieve from position i for downstream computation. This separation into three distinct roles allows the model to decouple matching (query-key interaction that determines attention weights) from information flow (value retrieval that determines the output content), providing flexibility in how attention is computed and applied. A single embedding can have a key that matches many queries while providing a value optimized for a different purpose, enabling sophisticated information routing patterns.

To compute the attention output for position i, we measure how well $\mathbf{q}[i]$ matches each key $\mathbf{k}[j]$ for $j \leq i$ (in the causal setting where future positions are masked). The match is quantified by the dot product $\mathbf{q}[i]^\top \mathbf{k}[j]$, which is large when the two vectors are aligned (pointing in similar directions) and small or negative when they are orthogonal or opposed. Geometrically, the dot product equals $\|\mathbf{q}[i]\| \|\mathbf{k}[j]\| \cos\theta_{ij}$, measuring both alignment angle and vector magnitudes. We collect all these dot products into a vector of scores, scale them by $1/\sqrt{d_{\text{model}}}$ to control their magnitude and variance, and apply softmax to convert scores into a probability distribution. Mathematically, the attention weights $\alpha[ij]$ are given by

$$\alpha[ij] = \frac{\exp\left(\frac{\mathbf{q}[i]^\top \mathbf{k}[j]}{\sqrt{d_{\text{model}}}}\right)}{\sum_{k=1}^{i} \exp\left(\frac{\mathbf{q}[i]^\top \mathbf{k}[k]}{\sqrt{d_{\text{model}}}}\right)}.$$

The softmax ensures that $\sum_{j=1}^{i} \alpha[ij] = 1$ and $\alpha[ij] \geq 0$, making the weights a valid probability distribution over previous positions that can be interpreted as the model's belief about relevance. The scaling factor $1/\sqrt{d_{\text{model}}}$ prevents the dot products from growing too large in high dimensions (where random vectors have expected dot product variance proportional to d_{model}), which would cause the softmax to saturate and produce near-one-hot distributions with negligible gradients that block learning.

The attention output for position i, denoted $\text{Attention}(\mathbf{q}[i], \mathbf{K}, \mathbf{V})$, is the weighted sum of value vectors using the attention weights:

$$\text{Attention}(\mathbf{q}[i], \mathbf{K}, \mathbf{V}) = \sum_{j=1}^{i} \alpha[ij]\, \mathbf{v}[j].$$

This weighted combination integrates information from all previous positions, with the weights determining each position's contribution to the output representation. If the model learns that position j is highly relevant for predicting at position i, then $\alpha[ij]$ will be close to 1 and the attention output will approximate $\mathbf{v}[j]$, effectively copying that position's information. If relevance is distributed across multiple positions, the output will be a

more balanced mixture lying in the convex hull of the value vectors with coefficients given by the attention weights. The attention mechanism is fully differentiable with respect to all parameters: gradients flow through the softmax via the Jacobian $\partial \text{softmax}/\partial z$, through the dot products via chain rule, and back to the query, key, and value weight matrices W^Q, W^K, W^V. During training with cross-entropy loss on next-word prediction, the model learns to construct queries that ask the right questions about context, keys that advertise useful information about each position, and values that provide that information in a form useful for predicting the next token.

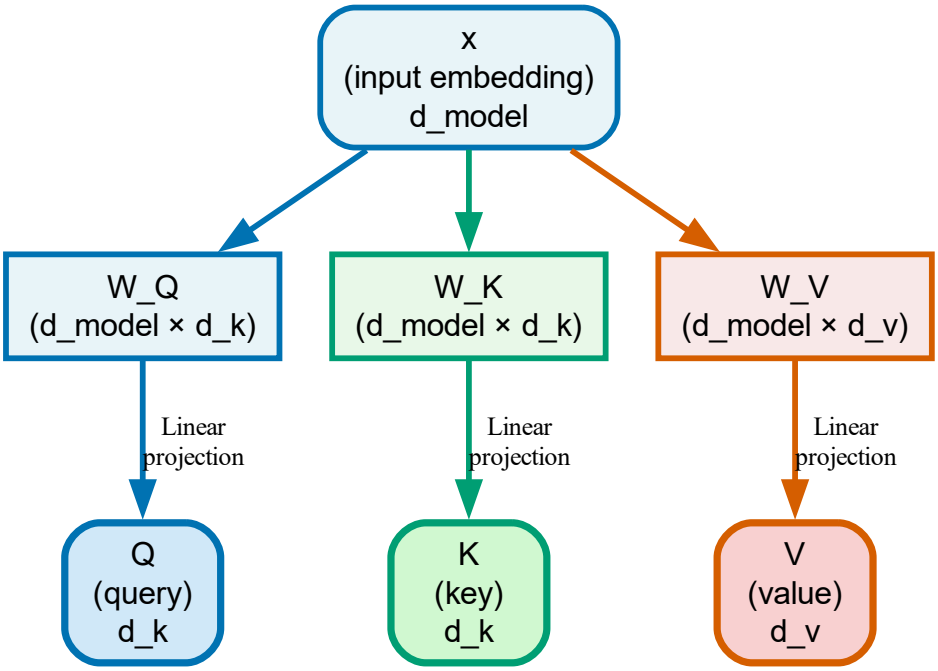


Figure 6.5: Projecting token embeddings into query, key, and value spaces. Each embedding $\mathbf{e}[i]$ is multiplied by three learned weight matrices to produce $\mathbf{q}[i]$, $\mathbf{k}[i]$, and $\mathbf{v}[i]$. These projections decouple the roles of matching (Q-K) and information retrieval (V).

Figure 6.5 shows the linear projections that transform embeddings into queries, keys, and values through learned affine transformations. The same embedding $\mathbf{e}[i]$ generates all three vectors, but the three weight matrices W^Q, W^K, W^V are learned independently during training through backpropagation, giving the model flexibility to extract different information for different purposes. This allows the model to represent different aspects of each token depending on its role in the attention mechanism, effectively learning task-specific feature extractors. For example, the query projection might emphasize syntactic properties (part of speech, grammatical role) when the model is searching for grammatical dependencies like subject-verb agreement, while the value projection might emphasize semantic properties (word meaning, topic membership) when the model is retrieving meaning for next-word prediction. The dimension d_{model} is typically the same for embeddings, queries, keys, and values in standard transformers, preserving model capacity and allowing residual connections, though some variants use lower-dimensional key and value projections $d_{kv} < d_{\text{model}}$ to reduce the quadratic memory cost of storing attention weights. The learned projections enable the model to discover through gradient descent which features are most useful for matching relevance (Q-K interaction) versus which are most useful for information retrieval (V contribution to output).

The dot product similarity measure in Figure 6.6 is computationally efficient and geometrically in-

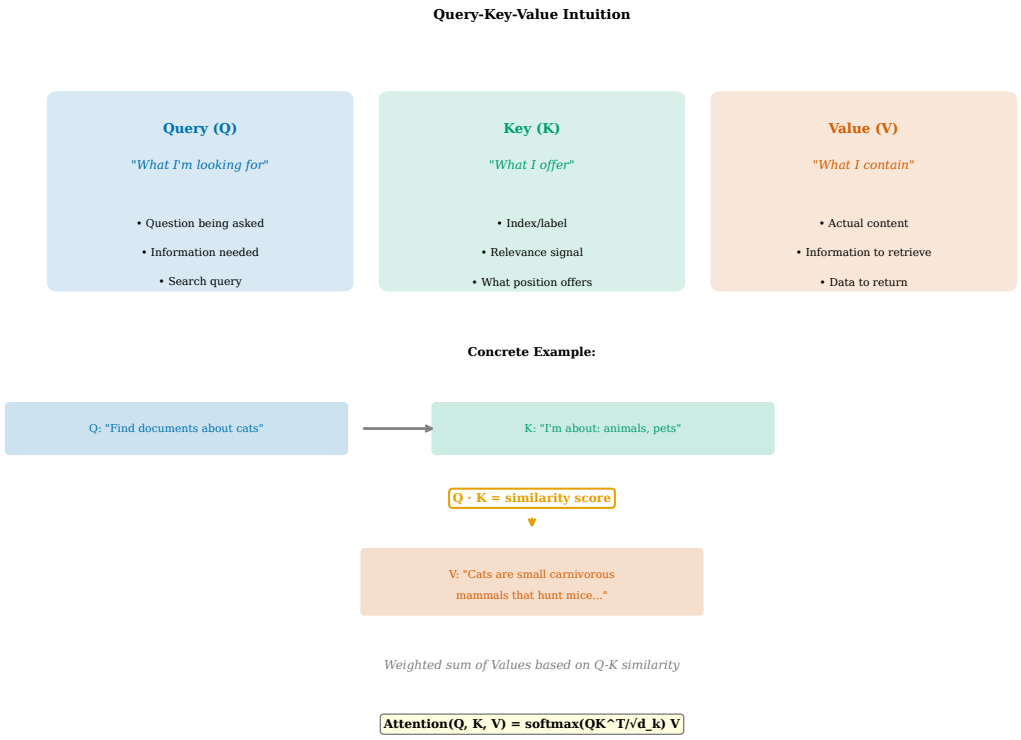


Figure 6.6: Computing attention scores via dot products. For each position i, we compute the dot product of $\mathbf{q}[i]$ with all keys $\mathbf{k}[j]$ for $j \leq i$. Higher dot products indicate stronger alignment between the query and key, suggesting that position j is relevant for position i.

terpretable, making it ideal for high-dimensional learned representations. The dot product $\mathbf{q}[i]^\top \mathbf{k}[j] = \|\mathbf{q}[i]\| \|\mathbf{k}[j]\| \cos\theta$, where θ is the angle between the two vectors in d_{model}-dimensional space. If queries and keys are normalized to unit length, the dot product directly measures the cosine similarity, which is 1 when vectors are parallel (perfectly aligned, maximum similarity) and 0 when they are orthogonal (independent, no similarity). In practice, transformers do not explicitly normalize queries and keys before the dot product, allowing the model to learn both direction (via angle, capturing semantic alignment) and magnitude (via vector length, capturing confidence or salience) as signals for relevance. The scaling by $1/\sqrt{d_{\text{model}}}$ compensates for the fact that in high dimensions, random vectors tend to have large dot products simply due to the accumulation of many small terms across dimensions: for i.i.d. entries with variance σ^2, the dot product has variance $d_{\text{model}}\sigma^4$, which grows with dimension. Matrix multiplication hardware on modern GPUs and TPUs makes dot product computation extremely fast with $O(d_{\text{model}})$ operations per query-key pair, enabling attention over thousands of positions efficiently in parallel through batched matrix-matrix multiplication.

Figure 6.7 illustrates the softmax normalization that converts unbounded real-valued scores into a probability distribution suitable for weighted averaging. The softmax function is defined as $\text{softmax}(z)_j = \exp(z_j)/\sum_k \exp(z_k)$, a generalization of the logistic sigmoid to multiple outputs. The exponential ensures all outputs are positive regardless of input sign, and the normalization ensures they sum to 1, forming a valid probability distribution over the i positions being attended to. The softmax is a smooth, differentiable approximation to the argmax: when one score is much larger than the others by more than a few units, the softmax output is close to a one-hot vector concentrating all weight on the maximum, with the Jacobian having near-zero off-diagonal entries. The scaling factor $1/\sqrt{d_{\text{model}}}$ acts as an inverse temperature in the softmax, controlling the peakedness of the resulting distribution. A smaller temperature (larger scaling) makes the distribution sharper and more peaked, concentrating attention on a few positions with near-hard selection, while a larger temperature (smaller scaling) makes it more uniform and diffuse, spreading attention broadly across all positions. The choice $1/\sqrt{d_{\text{model}}}$ was introduced in the original transformer paper by Vaswani et al. based on empirical performance and the statistical properties of high-dimensional dot products, ensuring variance-normalized scores

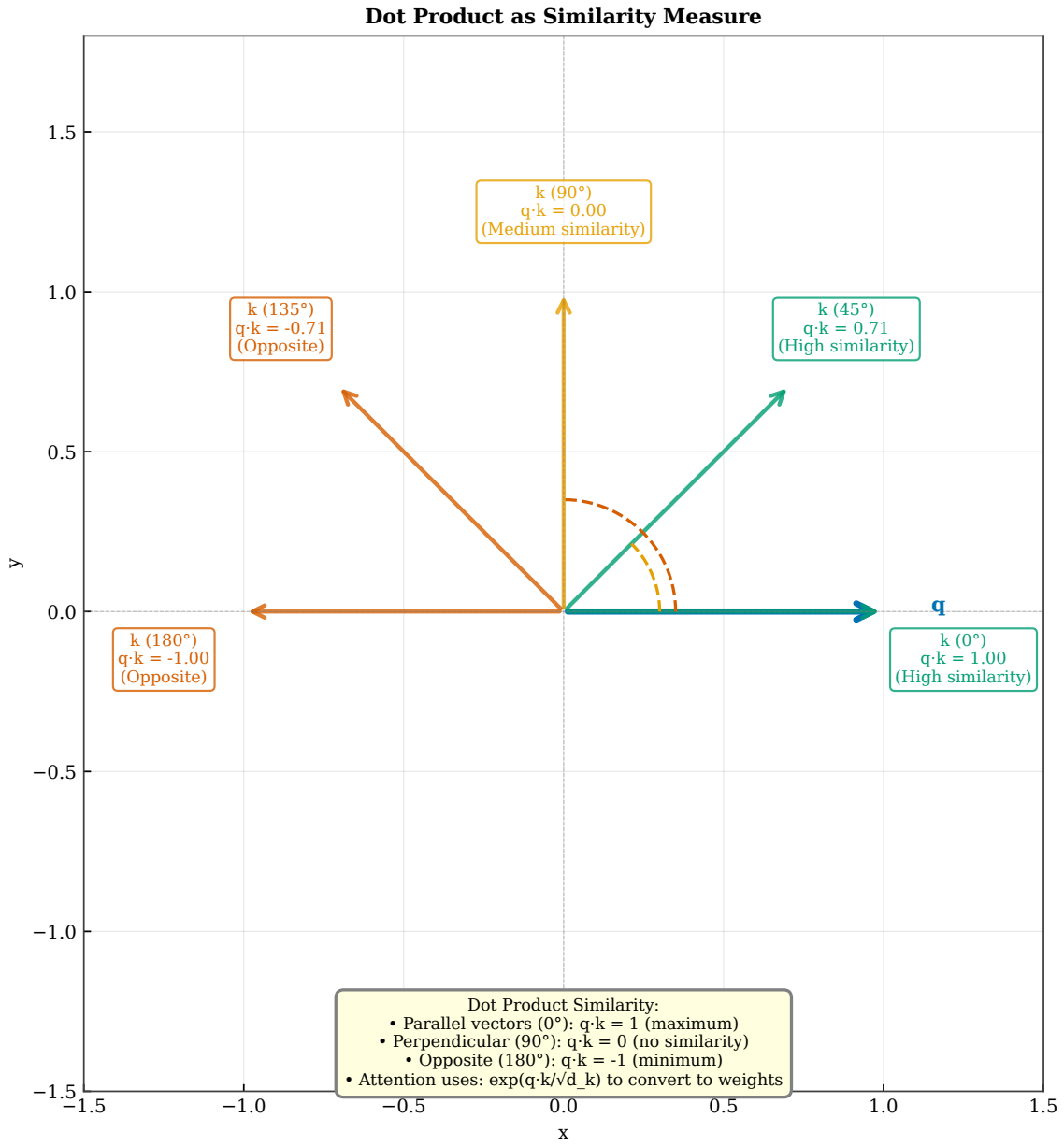


Figure 6.7: Converting attention scores to weights via softmax. The raw dot-product scores (which can be any real number) are exponentiated and normalized to sum to 1, producing a valid probability distribution over previous positions. Higher scores correspond to higher weights after normalization.

enter the softmax.

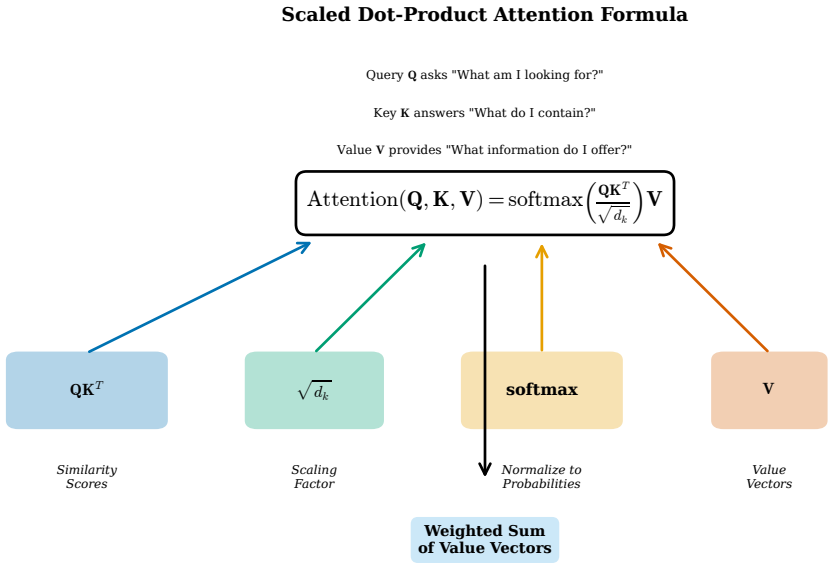


Figure 6.8: Computing the attention output as a weighted sum of values. Each value vector $\mathbf{v}[j]$ is scaled by its attention weight $\alpha[ij]$ and summed to produce the final output. Positions with higher weights contribute more to the output representation.

The weighted sum in Figure 6.8 integrates information from all attended positions into a single output vector through linear combination with learned coefficients. This output has the same dimension d_{model} as the input embeddings, allowing it to replace or augment the original embedding in subsequent layers through residual connections. In a transformer decoder, the attention output is typically combined with the original embedding via a residual connection $\mathbf{h} + \text{Attention}(\mathbf{h})$ and then passed through a feed-forward network for non-linear transformation. The weighted sum is a convex combination when all weights are nonnegative and sum to 1, which the softmax guarantees by construction through the exponential and normalization operations. Geometrically, the output lies in the convex hull of the value vectors (the smallest convex set containing all $\mathbf{v}[j]$), and the attention weights determine where in that hull it falls as barycentric coordinates. If one position receives weight close to 1, the output approximates that position's value vector, effectively copying that representation. If weights are distributed uniformly as $1/i$ for each of i positions, the output blends multiple perspectives smoothly into an average. The differentiability of the weighted sum ensures gradients flow back to all attended positions proportionally to their attention weights, enabling credit assignment during training.

Figure 6.9 visualizes the full attention weight matrix for a sequence, where entry $\alpha[ij]$ represents the attention weight from query position i to key position j. Reading across row i shows which previous positions are most important for position i, forming a probability distribution that sums to 1. Some patterns commonly emerge from training on natural language: diagonal attention (each position attends primarily to itself and nearby neighbors, capturing local syntactic context and n-gram patterns), vertical lines (all positions attend to a particular salient token, such as the subject of a sentence or a discourse marker, indicating globally relevant information), and horizontal bands (a position attends uniformly to a range of previous positions, performing a smoothing or averaging operation). These patterns are learned from data through gradient descent and vary across different attention heads and layers, as we will see in Section 6.4, enabling specialization for different linguistic functions. The lower-triangular structure enforces causality: position i cannot attend to position $j > i$ because those entries are masked to $-\infty$ before softmax, preventing the model from peeking into the future dur-

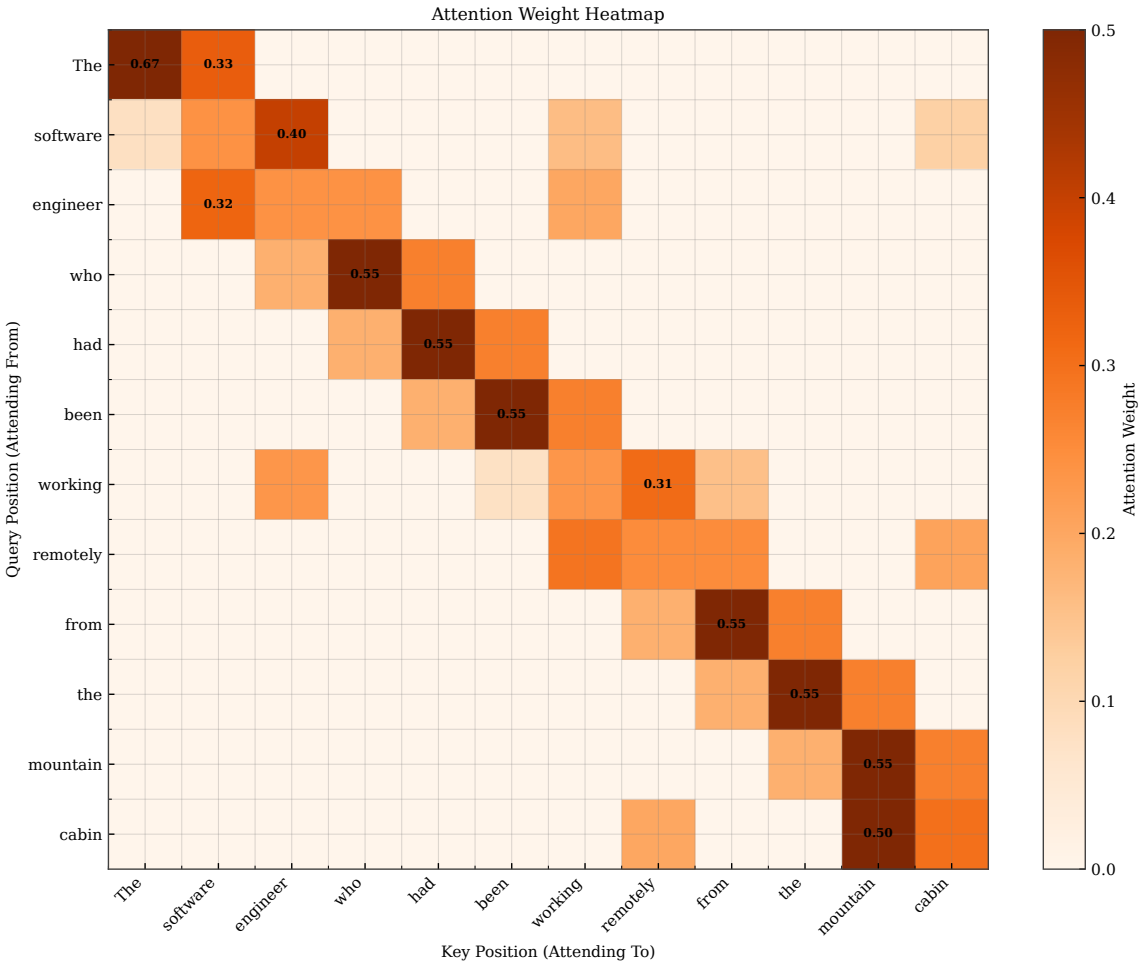


Figure 6.9: Attention weight matrix α for an entire sequence. Rows correspond to query positions (where we predict), columns correspond to key positions (where we look). Each entry $\alpha[ij]$ shows how much position i attends to position j. The matrix is lower-triangular due to causal masking (discussed in the next section).

ing training or generation. Analyzing these matrices helps researchers understand which linguistic phenomena the model has learned to track and enables interpretability studies of transformer behavior, revealing learned patterns through visualization.

Applying attention to our running example in Figure 6.10, we see that the learned weights concentrate on content words that provide semantic and syntactic constraints relevant for next-word prediction. The high weight on "code" makes sense from an information-theoretic perspective: the phrase "the code that would _____" strongly suggests a verb describing what code does, such as "run", "compile", or "solve", significantly reducing the entropy of the next-word distribution. The attention to "submitted" captures the past tense and completed aspect, which might influence whether the continuation uses a modal verb or a different tense, establishing temporal coherence. The attention to "engineer" and "software" reinforces the technical domain through semantic field activation, narrowing the distribution over possible next words to programming-related vocabulary. Function words like "who", "had", "been", "from", and "the" receive lower weights because they carry less mutual information with the next word, though they do contribute to grammatical structure and are handled by other attention heads specializing in syntax. This learned selectivity is a key advantage of attention: the model automatically discovers through gradient descent which context positions matter most for minimizing prediction loss, without requiring manual feature engineering or explicit linguistic rules.

The three-dimensional surface in Figure 6.11 provides another perspective on the attention weight matrix by mapping $\alpha[ij]$ to height at coordinates (i, j). Height represents attention strength, making peaks visually salient where the model attends strongly and valleys showing low attention where positions are deemed irrele-

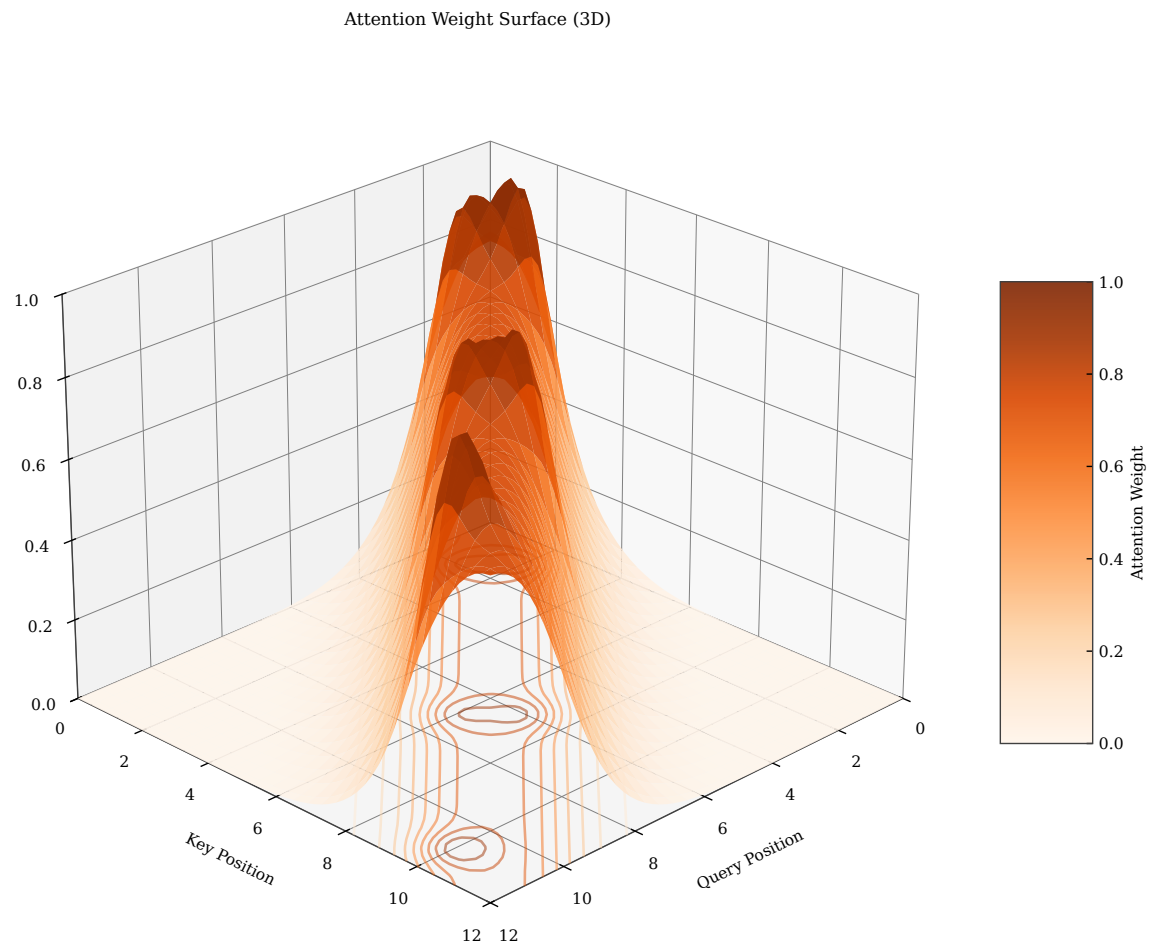


Figure 6.10: Attention weights for position 23 (predicting the next word) in our running example. The model attends most strongly to "code" (position 21), "submitted" (position 19), "engineer" (position 3), and "software" (position 2), which are semantically and syntactically relevant for predicting a programming-related verb.

vant. This view can reveal global patterns such as whether attention is concentrated (many tall, narrow peaks corresponding to selective focus with near-one-hot distributions) or diffuse (a relatively flat surface indicating broad context aggregation with near-uniform weights), and whether certain positions consistently receive high attention across many query positions (ridges running parallel to the query axis, indicating globally important tokens). The lower-triangular constraint appears as a sharp cliff at the diagonal: the surface is nonzero only for $j \leq i$, dropping to exactly zero where causal masking prevents attending to future positions. The three-dimensional visualization makes the sparsity or density of attention patterns immediately apparent through surface topology, helping researchers diagnose whether models are learning useful selectivity with interpretable patterns or attending too uniformly to all context without differentiation. This visual analysis can guide architectural choices such as the number of attention heads and debugging of attention mechanisms when models underperform.

Figure 6.12 demonstrates why the scaling factor $1/\sqrt{d_{\text{model}}}$ is critical for stable training and gradient flow through deep transformer networks. In high dimensions, the variance of a dot product between two random d_{model}-dimensional vectors grows linearly with d_{model}: if each component has variance σ^2, the dot product has variance $d_{\text{model}}\sigma^4$. Without scaling, dot products can reach values of magnitude 10 or more for $d_{\text{model}} = 512$, pushing the softmax into saturation where $\exp(z_i)/\sum_k \exp(z_k) \approx 1$ for the largest score and approximately 0 for all others, producing a near-one-hot distribution. The gradients of softmax in this saturated regime are tiny because the Jacobian entries approach zero, effectively blocking learning and preventing the model from discovering subtle attention patterns that distribute weight across multiple positions. Dividing by $\sqrt{d_{\text{model}}}$

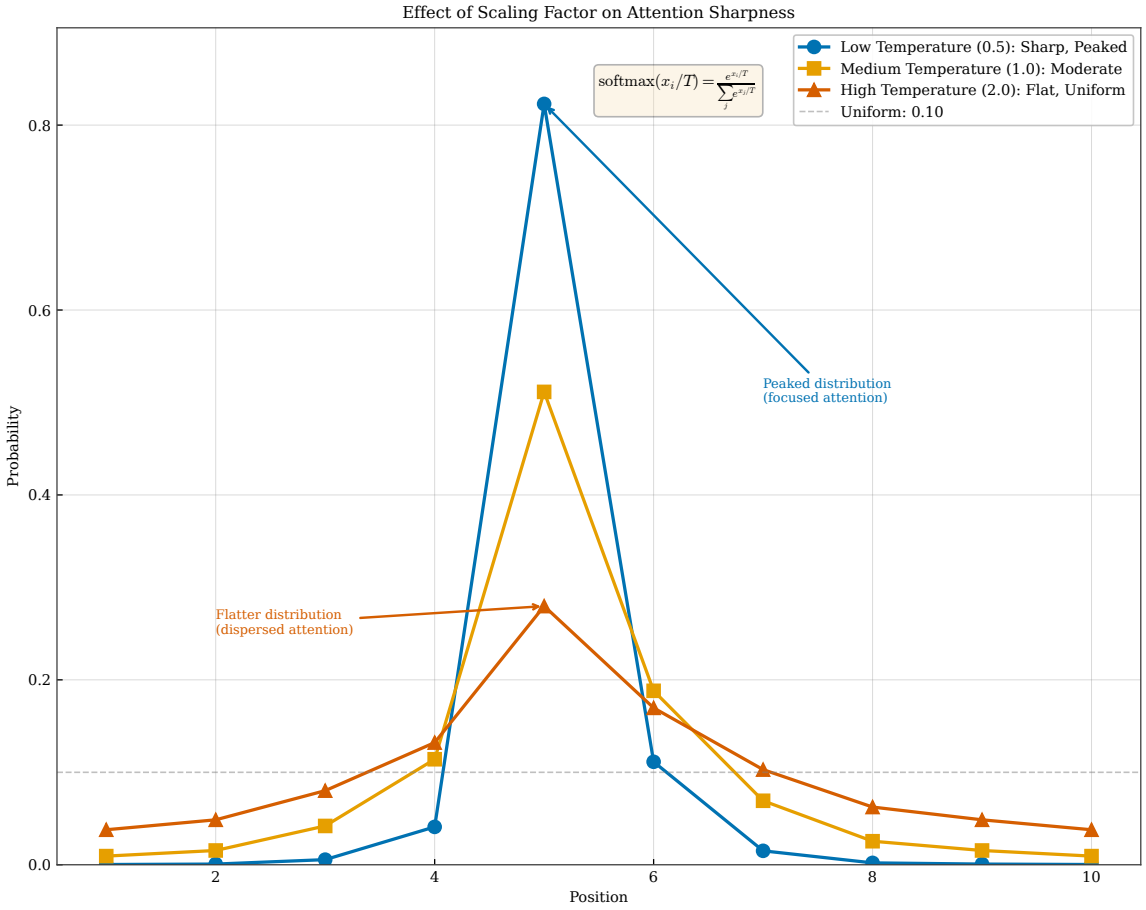


Figure 6.11: Three-dimensional visualization of attention weights as a surface. The height at position (i, j) represents $\alpha[ij]$. Peaks indicate strong attention, while valleys indicate low attention. The surface shows how attention evolves across the sequence.

keeps the variance of scaled dot products near 1 regardless of dimension, ensuring that the softmax operates in its sensitive regime where gradients are substantial and the model can learn to adjust attention weights smoothly. This scaling is a small but essential detail that enables transformers to train effectively at large model dimensions $d_{\text{model}} \geq 512$ and deep architectures with many layers, maintaining gradient magnitude throughout the network.

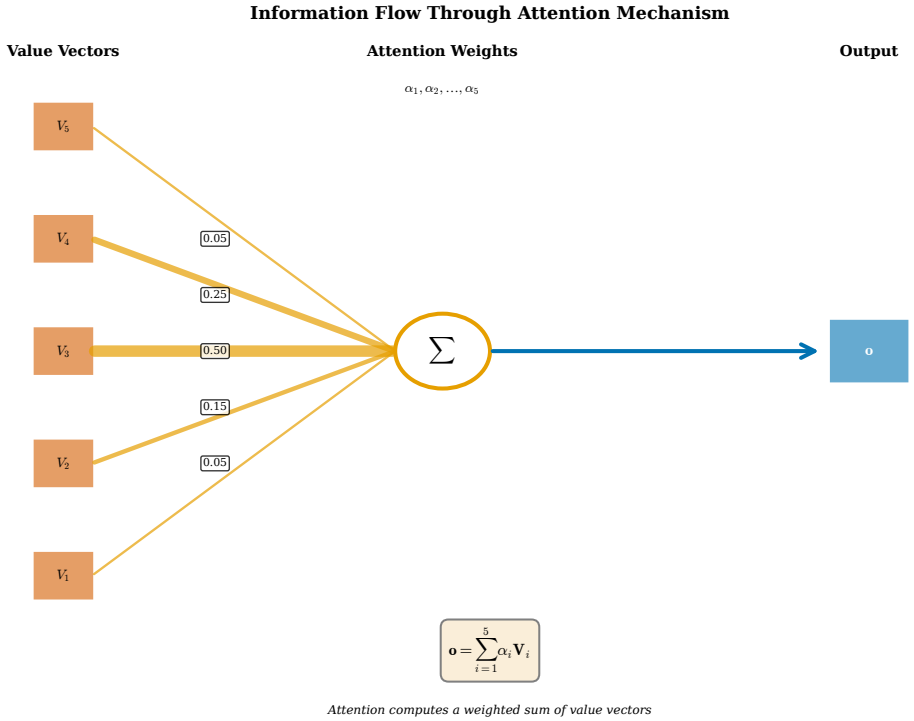


Figure 6.12: Effect of the scaling factor $1/\sqrt{d_{\text{model}}}$ on attention distributions. Without scaling (top), large dot products cause the softmax to produce near-one-hot distributions with negligible gradients. With scaling (bottom), the distribution is smoother, allowing gradients to flow to multiple positions.

6.3 Causal Masking for Autoregressive Generation

Language models are trained to predict the next word given all previous words, and they generate text one word at a time in an autoregressive fashion following the chain rule of probability: we predict $w[1]$, then use $w[1]$ to predict $w[2]$, then use $w[1]$ and $w[2]$ to predict $w[3]$, and so on until reaching the end of sequence. This autoregressive property is fundamental to the probability factorization $P(w[1:T]) = \prod_{t=1}^{T} P(w[t] \,|\, w[1:t-1])$, which decomposes joint probability into a product of conditionals. During training, however, we have access to the entire sequence $w[1:T]$ for computing the loss function. If we naively allowed the attention mechanism to attend to all positions including future ones, the model could simply copy $w[t]$ from position t when predicting it at position $t-1$, achieving perfect training loss of zero cross-entropy without learning anything useful about next-word prediction from context patterns. To prevent this cheating or information leakage, we apply *causal masking*: when computing attention at position i, we forbid attending to any position $j > i$ by setting those attention scores to negative infinity before the softmax. This ensures that the model sees only the context available during generation, making training consistent with inference and preventing distribution shift between the two phases.

Causal masking is implemented by setting the attention scores for forbidden positions to $-\infty$ before applying the softmax, a technique that leverages the properties of the exponential function. Mathematically, the masked attention weights are

$$\alpha[ij] = \begin{cases} \dfrac{\exp\left(\frac{\mathbf{q}[i]^\top \mathbf{k}[j]}{\sqrt{d_{\text{model}}}}\right)}{\sum_{k=1}^{i} \exp\left(\frac{\mathbf{q}[i]^\top \mathbf{k}[k]}{\sqrt{d_{\text{model}}}}\right)} & \text{if } j \le i, \\ 0 & \text{if } j > i. \end{cases}$$

The $-\infty$ scores become 0 after the exponential in softmax because $\exp(-\infty) = 0$ in the limit, and they do not contribute to the denominator because they are excluded from the sum by this zero value. This results in a lower-triangular attention matrix where $\alpha[ij] = 0$ for all $j > i$, with exactly $T(T+1)/2$ nonzero entries for sequence

length T. The remaining weights for $j \leq i$ still sum to 1 after normalization because the softmax denominator only includes the non-masked positions, so the attention output remains a valid weighted combination of available context positions. During the backward pass, gradients for masked positions are exactly zero because the forward pass contribution was zero, so the model receives no learning signal to attend to future positions, enforcing the causal constraint throughout training without any additional loss terms or regularization.

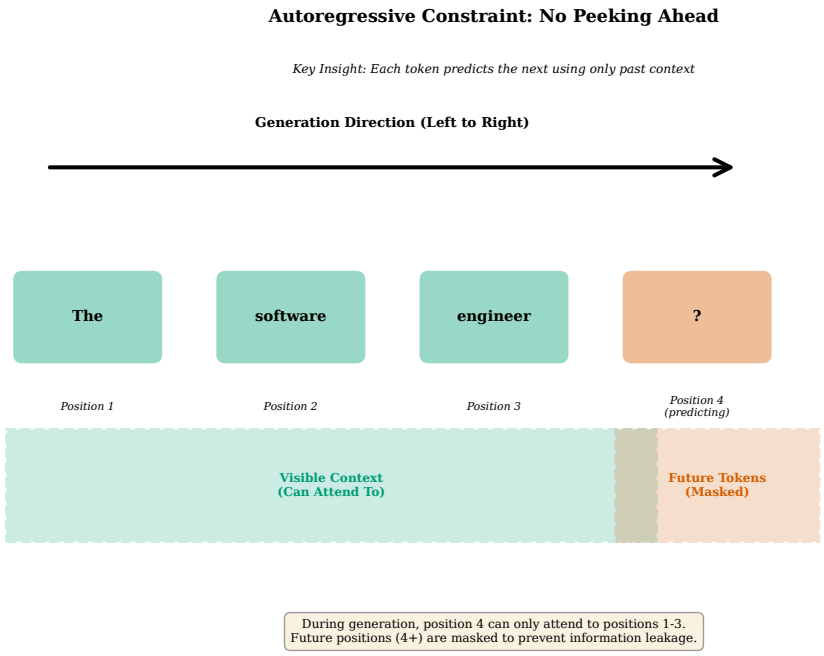


Figure 6.13: Causal mask structure for a sequence of length 8. White cells indicate allowed attention (score is computed normally), black cells indicate forbidden attention (score set to $-\infty$). The lower-triangular pattern ensures that each position can only attend to itself and earlier positions.

Figure 6.13 shows the binary mask applied before softmax, where white indicates allowed attention and black indicates forbidden positions. The mask is position-based, not content-based: it depends only on whether $j \leq i$, not on the actual tokens at those positions or their embeddings. This makes the mask completely data-independent and efficient to implement with $O(T^2)$ boolean values that can be computed once and reused. In practice, frameworks like PyTorch and JAX provide built-in functions for generating causal masks and applying them during attention computation through broadcasting and masking operations. The mask is often cached and reused across all layers and all attention heads, since the causal constraint is universal across the model and does not vary between heads or layers. The binary structure means the mask can be precomputed once at the start of training and stored efficiently as a boolean or binary tensor using only one bit per entry. Modern deep learning frameworks optimize masked operations heavily using fused kernels, making the computational overhead of masking negligible compared to the $O(T^2 \cdot d_{\text{model}})$ attention computation itself. The mask pattern remains identical across all training examples and batches, enabling aggressive optimization, memory reuse, and caching strategies that amortize mask creation cost.

Figure 6.14 shows the resulting attention weights after applying the causal mask, with the characteristic lower-triangular structure clearly visible. The zero entries in the upper triangle are strict: no information flows from future to past positions, guaranteeing the autoregressive property required by the probability factorization $P(w[1:T]) = \prod_{t=1}^{T} P(w[t] \,|\, w[1:t-1])$. Within the lower triangle, the attention mechanism is free to distribute weight however it deems useful based on learned relevance from training data, subject only to the softmax normalization constraint. Some positions might attend primarily to the immediately preceding token (diagonal attention, capturing bigram-like local dependencies), while others might attend to tokens much earlier in the

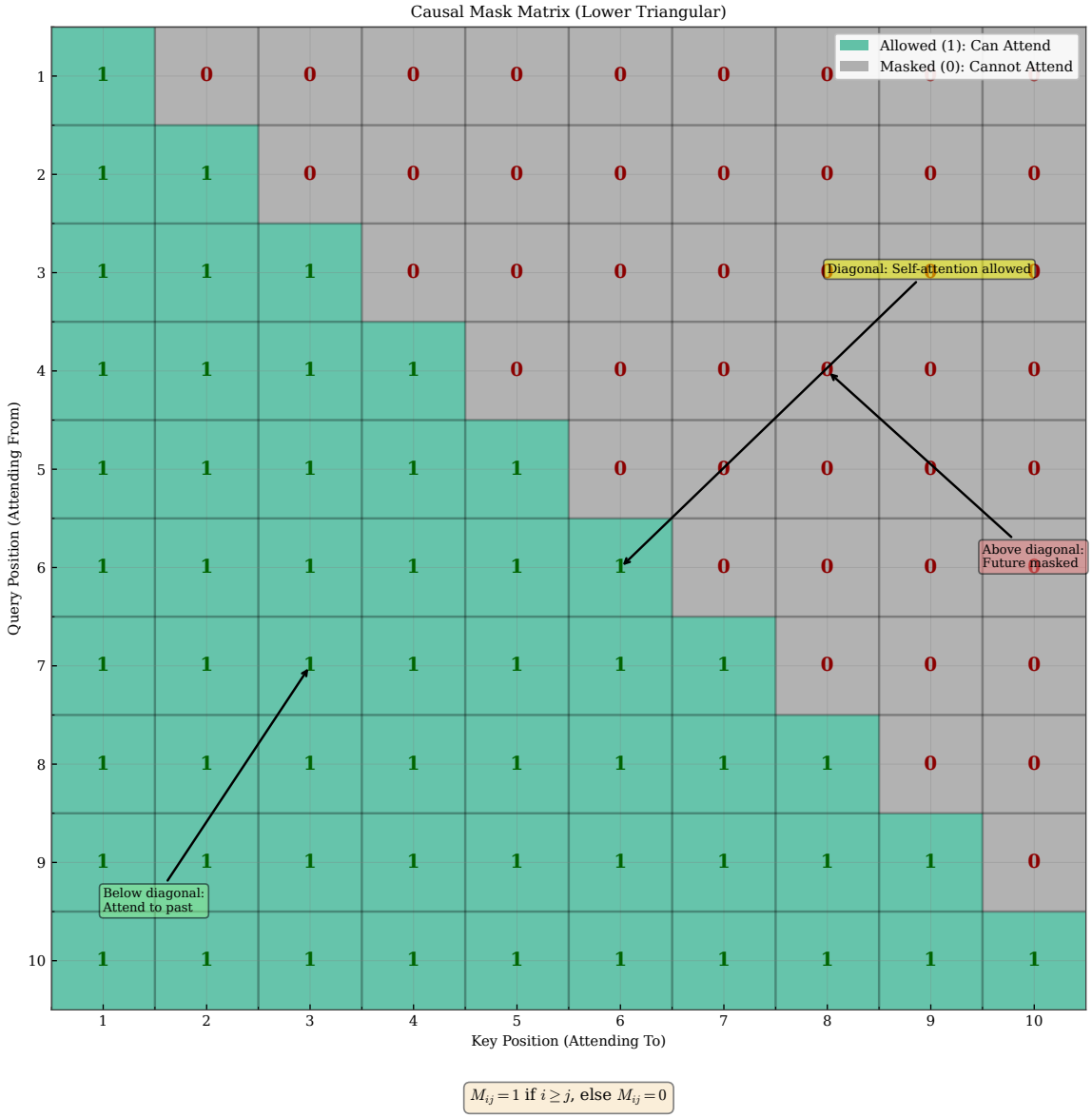


Figure 6.14: Attention weight matrix with causal masking applied. Compare to Figure 6.9: the upper triangle is now strictly zero. Each row i represents a valid probability distribution over positions 1 through i.

sequence, skipping over irrelevant intermediate positions to capture long-range dependencies. The causal constraint does not dictate *how* to use the available context or which positions should receive high weight, only that unavailable future context is strictly excluded from the computation. This allows the model to learn arbitrarily complex dependencies within the causal window, such as attending to the beginning of a sentence when predicting the end for discourse coherence, without violating the temporal ordering required for generation.

In our running example shown in Figure 6.15, causal masking means that when predicting the word following "would" at position 23, the model can attend to all 22 preceding words but cannot peek at words that might come later in the sentence, even though they exist in the training data. During training on a dataset of complete sentences, the target word at position 23 exists in the training data and is used to compute the cross-entropy loss, but the model is prevented from seeing it when making the prediction through the attention mechanism. This ensures that the training objective matches the generation scenario exactly: in both cases, the model must predict $w[23]$ using only $w[1:22]$ as context, with no access to future information. If we were to remove causal masking, the model could attend to $w[23]$ when predicting $w[23]$, trivially achieving zero loss by learning an identity mapping or near-perfect copying rather than learning to model language from context. The consistency

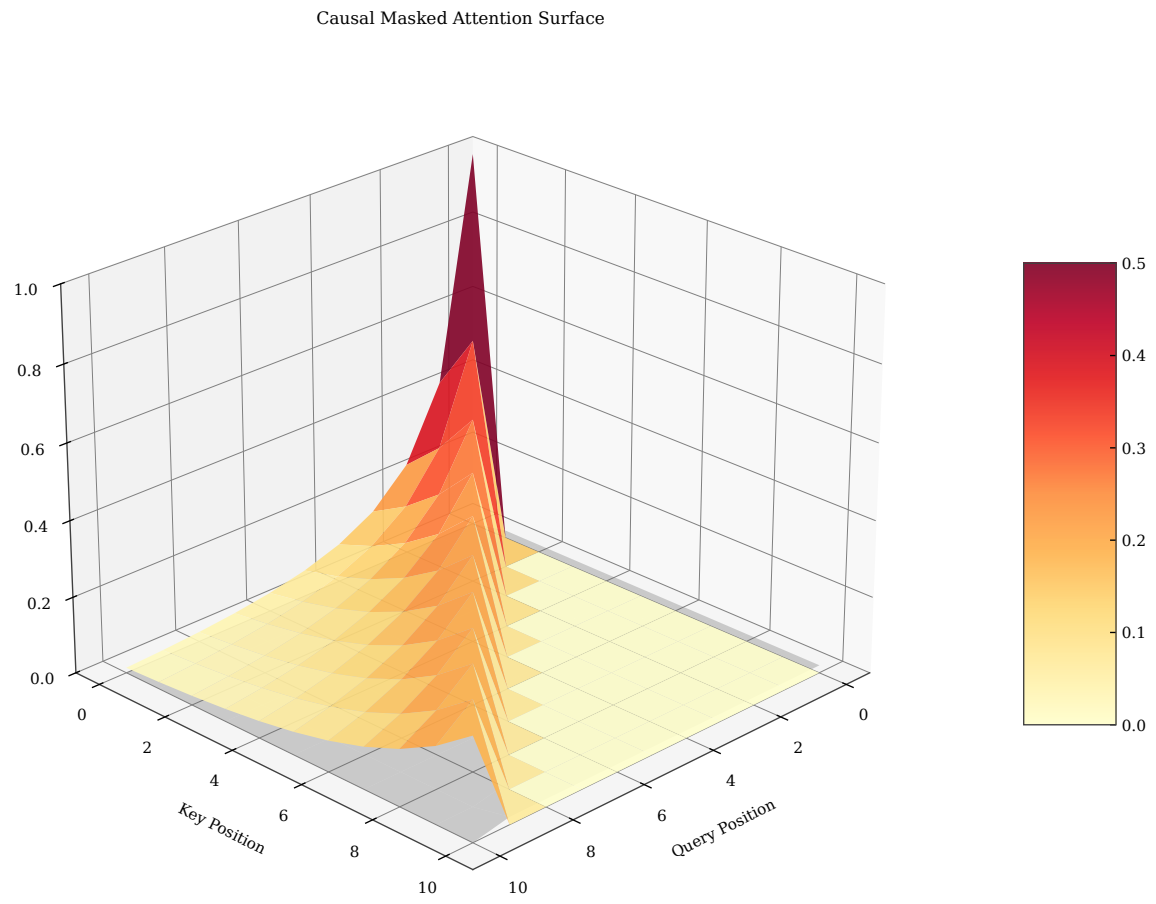


Figure 6.15: Applying causal masking to our running example. When predicting position 23, the model can attend to positions 1 through 22 (shown in color) but not to any future positions (grayed out). This matches the generation setting where future tokens are unknown.

between training and inference contexts is crucial for model performance and prevents distribution shift at test time, where future tokens are genuinely unknown and cannot be accessed by any mechanism.

The three-dimensional visualization in Figure 6.16 emphasizes the abrupt boundary imposed by causal masking, showing the attention surface with height proportional to weight. The surface rises from zero as we move back in time (decreasing j for fixed i), reaches peaks at positions the model finds most relevant for prediction, then drops to zero again for positions with low attention weight that contribute little to the output. The future region (where $j > i$) is an empty void with exactly zero height, representing information that does not exist yet in the autoregressive generation process and cannot contribute to predictions. This visualization makes clear that each position's context is strictly limited to what came before, preserving the temporal asymmetry inherent in language generation where cause precedes effect. The stepped pyramid structure emerges naturally from the position-by-position accumulation of context: position 1 has no prior context so its attention row is trivial, position 2 can attend only to position 1, position 3 to positions 1 and 2, and so forth until position T can attend to all $T - 1$ preceding positions. This incremental expansion of the attention window with row i having exactly i possible nonzero entries ensures consistent behavior across all sequence positions and prevents information leakage from future to past.

Figure 6.17 shows how causal masking operates during generation, maintaining consistency with training throughout the autoregressive process. At each step, the model processes the entire sequence constructed so far through the full transformer stack, but each position attends only to positions at or before itself due to the causal mask. When generating the first token, position 1 attends only to itself (the initial embedding or start token that seeds the generation). When generating the second token, position 2 can attend to positions

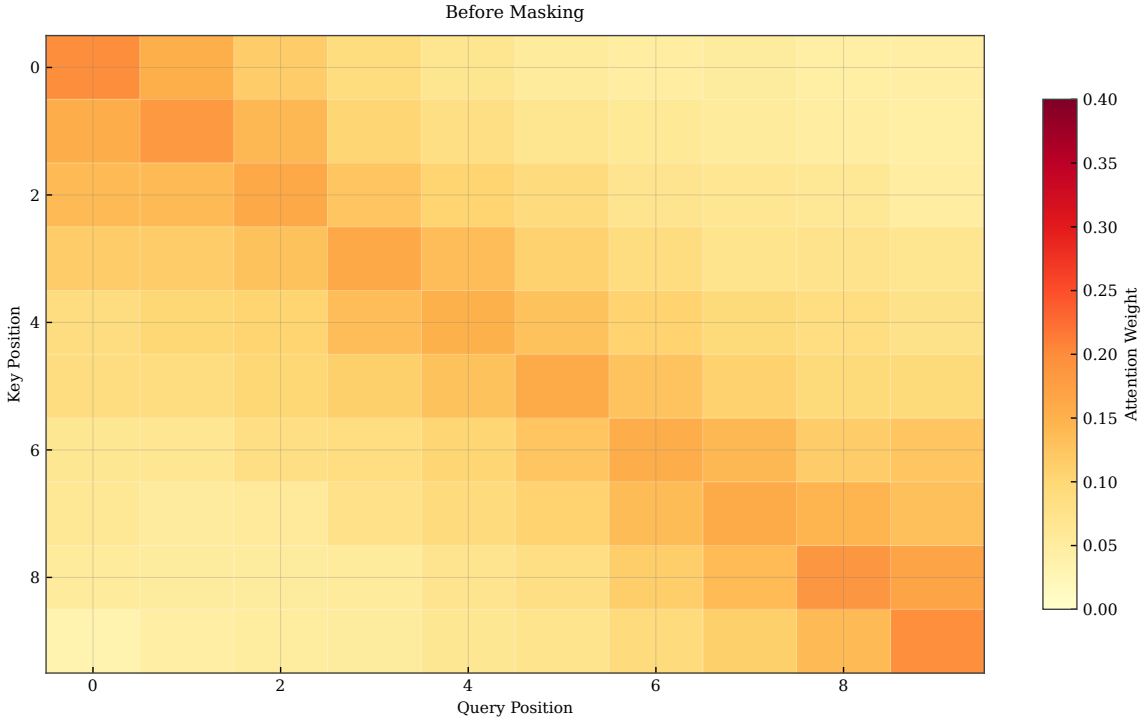


Figure 6.16: Three-dimensional view of causal masked attention. The surface is nonzero only in the lower triangle, forming a stepped pyramid shape. The sharp drop to zero at the diagonal boundary enforces the autoregressive constraint.

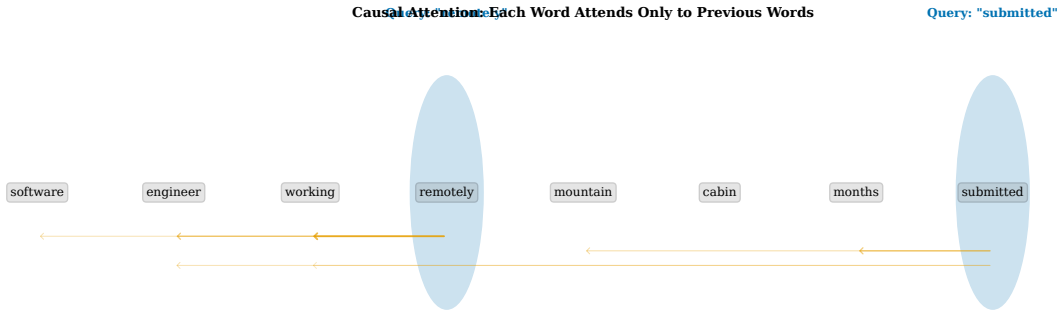


Figure 6.17: Autoregressive generation with causal attention. At step 1, the model attends only to the initial prompt. At step 2, it can attend to the prompt plus the first generated token. At step 3, the context includes all previously generated tokens. Each step uses causal masking to ensure consistency.

1 and 2, incorporating the first generated token into context. This continues iteratively until we reach the desired sequence length or generate an end-of-sequence token indicating completion. The causal masking at generation time is identical to the masking used during training on the same sequence positions, ensuring that the model never encounters distribution shift between training and inference contexts. This consistency is a major advantage of transformers over some alternative architectures where training and generation procedures differ significantly due to exposure bias or other mismatches. The model sees exactly the same attention patterns during generation that it learned to use during training, eliminating the train-test mismatch problem that can degrade performance in sequence models.

6.4 Multi-Head Attention

A single attention mechanism computes one set of weights α, producing one weighted combination of values for each position based on a single learned notion of relevance. This single view of relevance might be insufficient to capture the diverse relationships that exist in language, which operates on multiple levels simultaneously. Some words relate through syntax (subject-verb agreement, pronoun-antecedent reference requiring grammatical tracking), others through semantics (thematic similarity, co-occurrence patterns indicating topical coherence), and still others through discourse structure (topic flow, narrative progression maintaining document-level coherence). Multi-head attention addresses this limitation by running multiple attention mechanisms in parallel, each with its own learned query, key, and value projections that can specialize for different purposes. These different *heads* can specialize in different types of relationships through independent learned parameters, providing a richer and more nuanced representation than a single attention mechanism that must compromise between objectives. The outputs of all heads are concatenated along the feature dimension and linearly transformed to produce the final multi-head attention output, combining diverse perspectives into one unified representation of dimension d_{model} that captures multiple aspects of linguistic structure simultaneously without forcing a single attention pattern to handle everything.

Formally, multi-head attention with n_{heads} heads operates as follows. We partition the model dimension d_{model} into n_{heads} equal parts, so each head operates in dimension $d_{\text{head}} = d_{\text{model}}/n_{\text{heads}}$. For head h, we have query, key, and value weight matrices $W_h^Q, W_h^K, W_h^V \in \mathbb{R}^{d_{\text{model}} \times d_{\text{head}}}$ and compute

$$\text{head}_h = \text{Attention}(\mathbf{e}W_h^Q, \mathbf{e}W_h^K, \mathbf{e}W_h^V),$$

where $\mathbf{e}$ is the matrix of all embeddings in the sequence (rows are positions, columns are dimensions), and the attention function computes scaled dot-product attention as defined in Section 6.2. Each head produces an output of dimension d_{head} for each position. We concatenate the outputs of all n_{heads} heads to obtain a d_{model}-dimensional vector per position, then apply a final linear projection $W^O \in \mathbb{R}^{d_{\text{model}} \times d_{\text{model}}}$ to produce the multi-head attention output:

$$\text{MHA}(\mathbf{e}) = \text{Concat}(\text{head}_1, \text{head}_2, \ldots, \text{head}_{n_{\text{heads}}})W^O.$$

The output has dimension d_{model}, matching the input dimension and allowing the multi-head attention module to be integrated into a residual stream. The total number of parameters is roughly the same as a single-head attention with full dimension, because each head uses smaller projection matrices.

Figure 6.18 illustrates the parallel structure of multi-head attention with n_{heads} heads operating independently on shared inputs. All heads operate simultaneously on the same input embeddings, but they use different projection matrices W_h^Q, W_h^K, W_h^V, so they compute different attention weights and produce different outputs specialized for their learned roles. The parallelism enables efficient computation on modern GPUs and TPUs, which excel at executing many independent operations concurrently through batched matrix multiplication. The concatenation combines these diverse perspectives into a single rich representation of dimension $n_{\text{heads}} \cdot d_{\text{head}} = d_{\text{model}}$. The final projection $W^O \in \mathbb{R}^{d_{\text{model}} \times d_{\text{model}}}$ allows the model to learn how to blend the head outputs: it might weight some heads more heavily than others, or learn to combine specific patterns from different heads into composite features useful for prediction. This learned blending is crucial because not all heads contribute equally to all prediction tasks at all positions. Some heads may be more important for syntactic

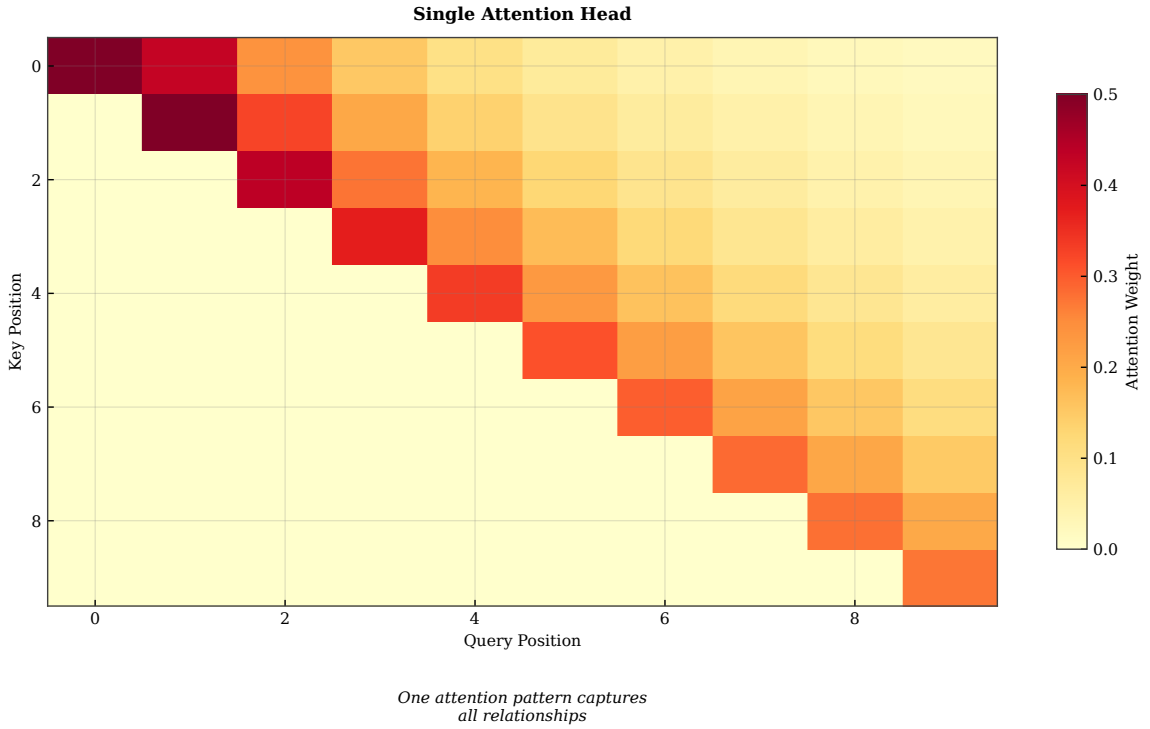


Figure 6.18: Multi-head attention architecture with $n_{\text{heads}} = 4$. The input embeddings are projected into queries, keys, and values for each head independently. Each head computes attention in its own d_{head}-dimensional subspace, producing a head output. All head outputs are concatenated and projected by W^O to produce the final output.

predictions requiring grammatical agreement while others matter more for semantic predictions requiring topical coherence. The model learns through backpropagation on the next-word prediction loss how to optimally weight each head's contribution for accurate prediction.

Figure 6.19 shows attention patterns from four different heads in a trained model, revealing emergent functional specialization. These patterns are learned from data through gradient descent, not manually specified or hard-coded by the architecture designer. Head 1 exhibits strong diagonal attention with weight concentrated on positions $j = i$ and $j = i - 1$, suggesting it captures local context or positional smoothing similar to bigram models. Head 2 shows a vertical line at the position of the subject noun, suggesting it has learned to track syntactic dependencies relevant for subject-verb agreement or pronoun resolution across arbitrary distances. Head 3 attends to multiple content words that are semantically related by topic or theme, suggesting a role in thematic coherence or topic modeling that maintains domain consistency. Head 4 shows a relatively flat distribution approaching $1/i$ for each query position i, suggesting it aggregates information broadly rather than focusing sharply on specific positions, computing something like an average representation. These diverse patterns illustrate how multi-head attention enables the model to simultaneously represent multiple views of the same sequence, each capturing different aspects of linguistic structure without interference. The specialization emerges automatically during training as the model discovers through backpropagation that different patterns are useful for minimizing prediction loss across diverse contexts.

The concatenation step in Figure 6.20 is a simple operation that stacks the head outputs side by side into a single long vector of dimension d_{model} by concatenating n_{heads} vectors of dimension d_{head}. If head 1 captures local context in dimensions 1 through d_{head} and head 2 captures syntactic dependencies in dimensions $d_{\text{head}} + 1$ through $2 \cdot d_{\text{head}}$, the concatenated vector contains both types of information in orthogonal subspaces that do not interfere. The final projection W^O can then learn to route this information appropriately: it might use the local context subspace for predicting function words and the syntactic dependencies subspace for predicting content words, for example, by having different rows of W^O attend to different head outputs. The projection also allows

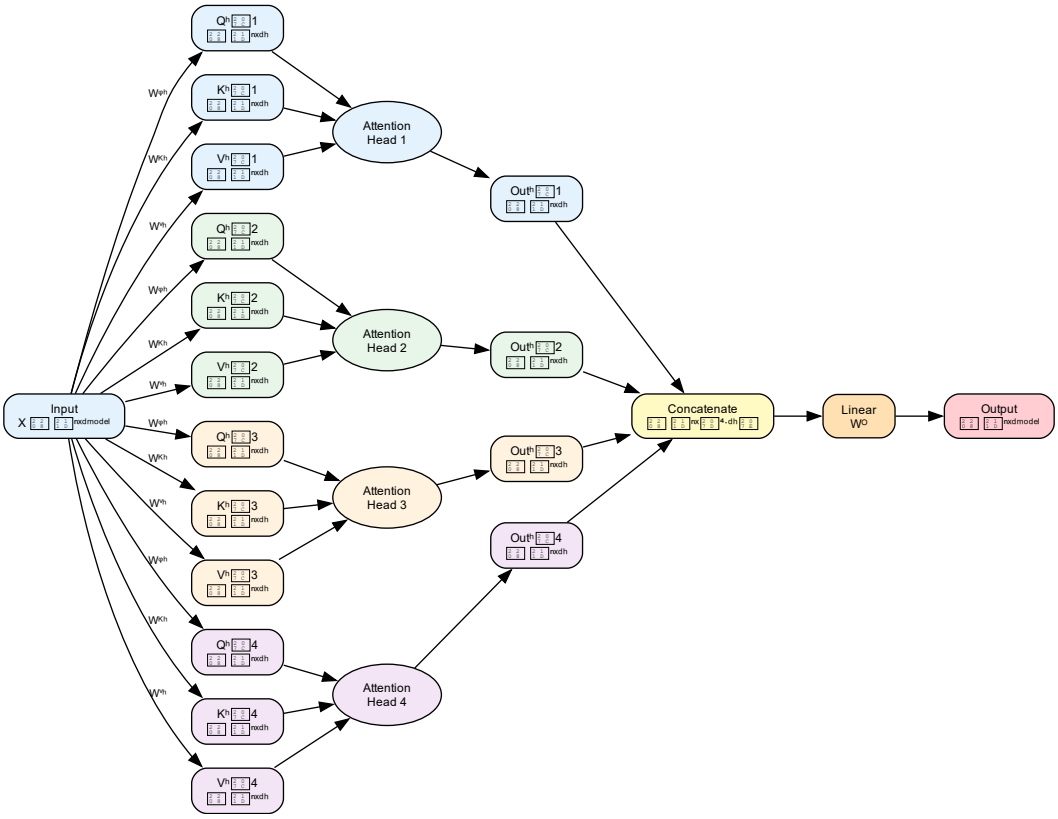
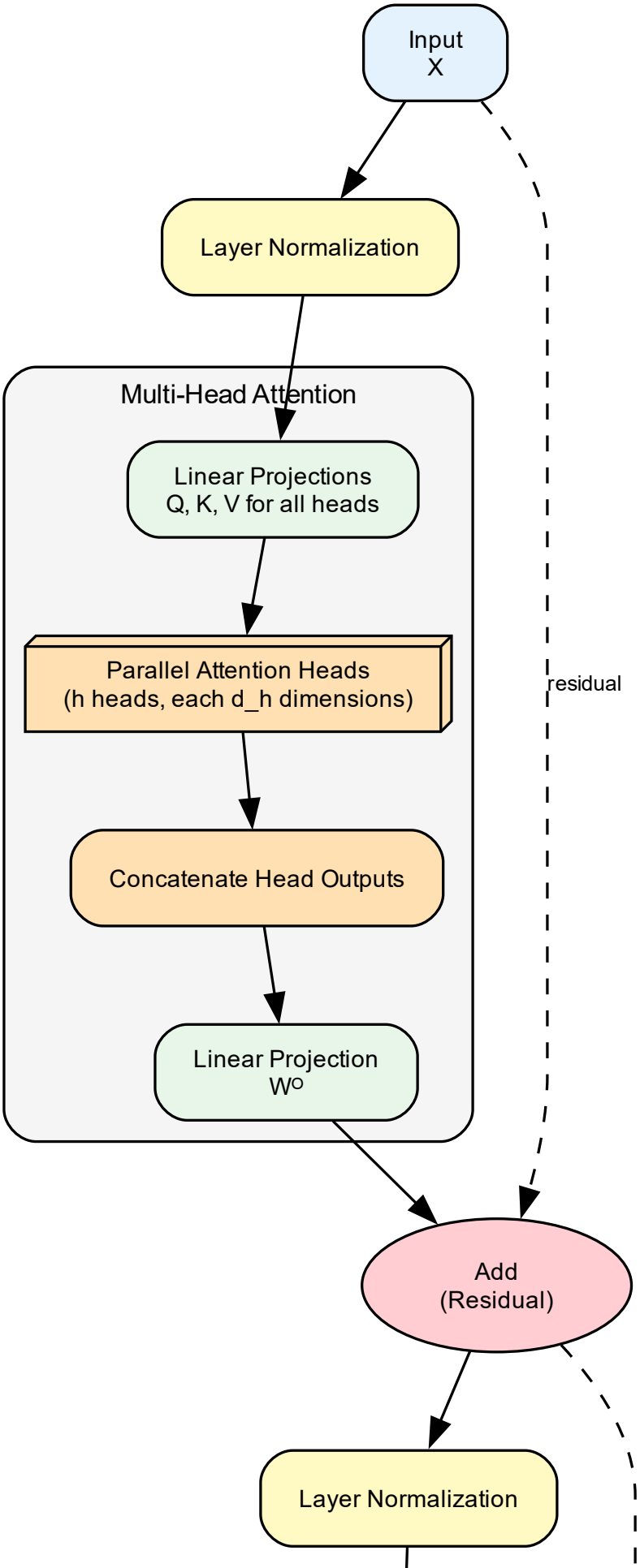


Figure 6.19: Example of head specialization in a trained transformer. Head 1 focuses on adjacent tokens (local context), Head 2 attends to the subject of the sentence (long-range syntactic dependency), Head 3 attends to semantically similar words (thematic coherence), and Head 4 shows a diffuse pattern (aggregating broad context).

the model to detect and exploit interactions between heads through cross-terms in the linear transformation, such as using head 1's output to modulate head 2's contribution when both are relevant. By learning the output projection through backpropagation on the prediction loss, the model discovers which combinations of head outputs are most useful for next-word prediction, adapting the blending to the specific patterns present in the training data distribution. This flexibility enables the multi-head mechanism to function as an ensemble of complementary attention patterns that collectively provide richer information than any single pattern alone.

Applying multi-head attention to our running example in Figure 6.21, we see that different heads prioritize different aspects of context with specialized attention patterns. One head might focus on the verb "submitted" to capture tense and aspect for temporal consistency, while another head focuses on the noun "code" to capture the object that the next verb will act upon for argument structure. A third head might attend to "engineer" and "software" to maintain topical coherence within the technical domain, biasing the vocabulary distribution toward programming terminology. By combining these perspectives through concatenation and the output projection W^O, the model builds a representation that captures multiple constraints simultaneously, leading to more accurate and contextually appropriate predictions than any single attention pattern could achieve. The multi-head mechanism prevents the model from committing to a single interpretation of relevance, instead maintaining multiple hypotheses about what context matters that are resolved jointly through the output projection based on learned importance weights. This ensemble approach to context aggregation provides robustness: if one head fails to capture a relevant dependency due to its learned query-key structure, other heads can compensate by attending differently. The diversity of attention patterns across heads reduces reliance on any single learned feature and provides redundancy against failure modes.


Input
X
Layer Normalization
Multi-Head Attention
Linear Projections
Q, K, V for all heads
Parallel Attention Heads
(h heads, each d_h dimensions)
Concatenate Head Outputs
Linear Projection
W^O
residual
Add
(Residual)
Layer Normalization

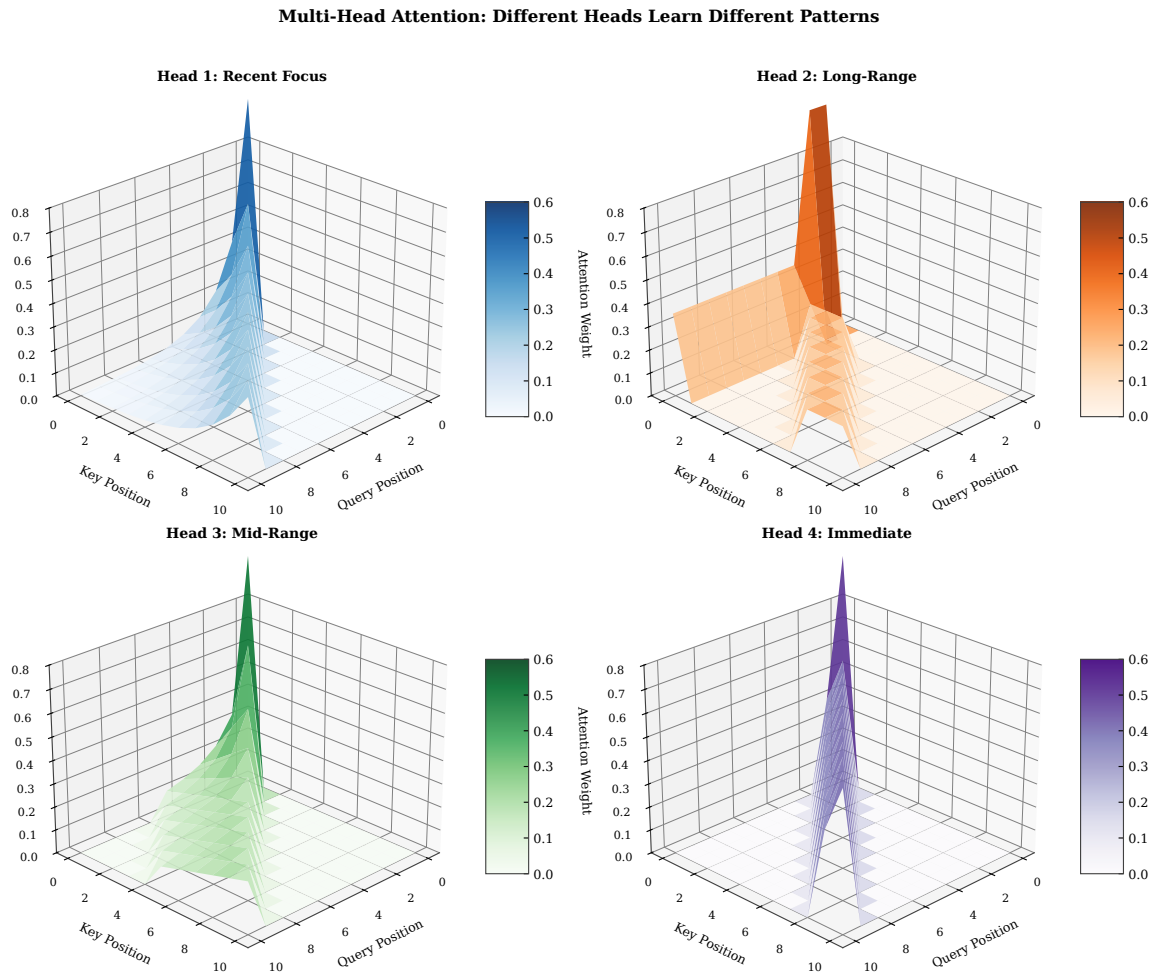


Figure 6.21: Multi-head attention applied to our running example. Different heads attend to different aspects of the context. Head 1 focuses on "code" and "submitted" (semantic and temporal cues), Head 2 focuses on "engineer" and "software" (domain and subject), demonstrating complementary views of relevance.

The three-dimensional multi-head visualization in Figure 6.22 juxtaposes the attention weight surfaces from different heads in a single view, enabling comparison of their learned behaviors. The variation in surface topology is striking: some heads produce tall, narrow peaks indicating very selective attention with near-one-hot distributions focusing on specific positions, while others produce low, broad plateaus indicating distributed attention that aggregates information uniformly. Some heads show smooth gradients suggesting soft relevance decay with distance, while others exhibit sharp discontinuities at specific positions indicating hard boundaries learned from syntactic structure. This heterogeneity suggests that the heads are not redundant but complementary, each contributing a distinct type of information to the final representation that would be lost if all heads behaved identically. Empirical studies have found that removing individual heads often degrades performance on specific tasks, but removing random sets of heads degrades it much more severely, indicating that the model relies on the diversity of the ensemble rather than the strength of any single head for robust performance. The complementarity emerges naturally during training as heads learn to specialize in different aspects of context through gradient descent. This automatic specialization arises from the gradient-based optimization process without explicit architectural constraints forcing differentiation, as heads learn to fill different functional niches.

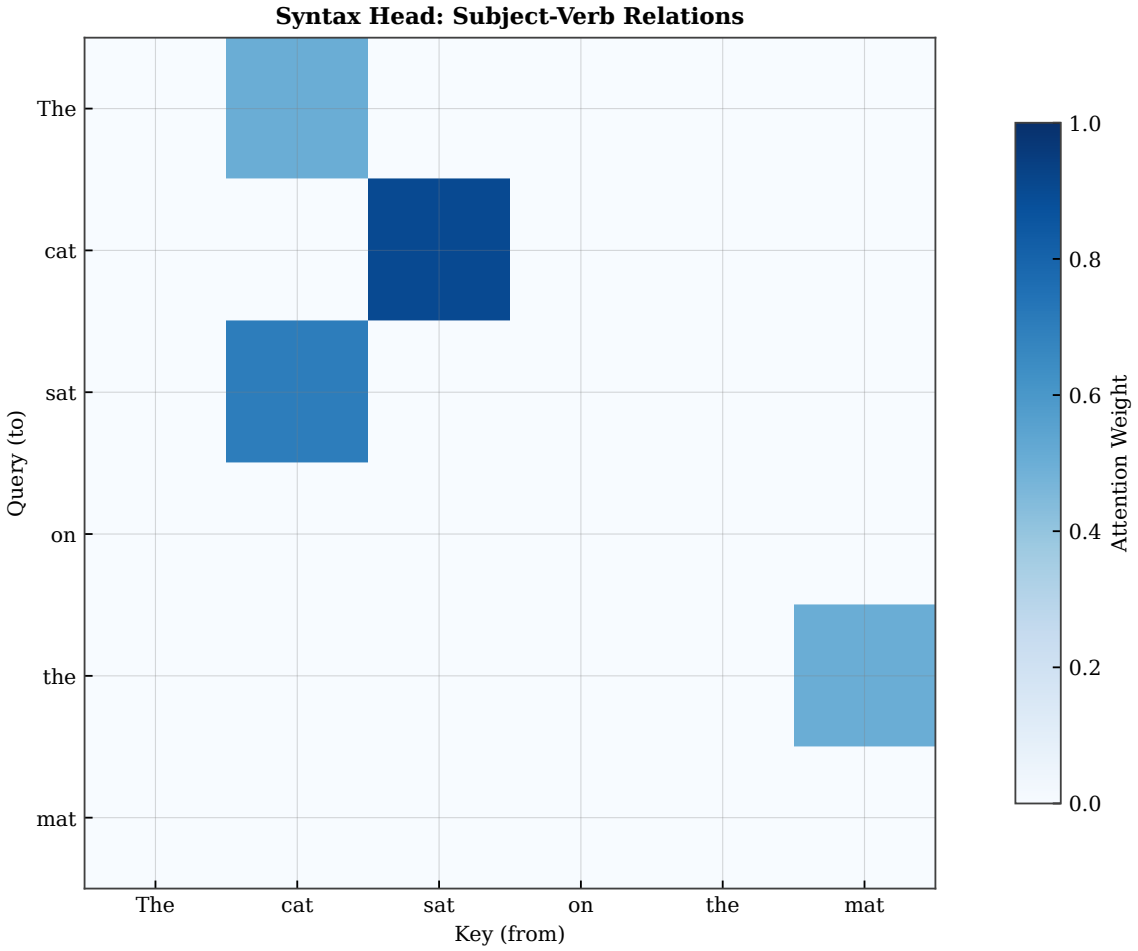


Figure 6.22: Three-dimensional visualization of attention patterns across four heads. Each surface represents the attention weights $\alpha[ij]$ for one head. The diversity in surface shapes reflects the different roles each head has learned, from sharp peaks (focused attention) to broad plateaus (diffuse attention).

6.5 Positional Encoding

The attention mechanism as described so far is permutation-invariant: if we reorder the input sequence, the attention weights and outputs will simply be reordered correspondingly, but the operation itself does not distinguish the order or use position information. Mathematically, the dot product $\mathbf{q}[i]^\top \mathbf{k}[j]$ depends only on the content of positions i and j through their embeddings, not on their absolute positions in the sequence or their relative distance $|i-j|$. For language modeling, however, word order is critical for determining meaning and grammatical structure. The sentence "The dog bit the man" has a very different meaning from "The man bit the dog," even though both contain the same words and the same bag-of-words representation would be identical. Without position information, the model cannot distinguish subject from object or agent from patient. To make transformers sensitive to position, we add *positional encodings* to the token embeddings before attention computation. These encodings inject information about each token's position in the sequence, allowing the model to differentiate tokens based on where they appear in the sequence order and to learn position-dependent patterns such as typical subject positions or verb placements.

The original transformer architecture introduced sinusoidal positional encodings, which use sine and cosine functions of different frequencies to encode position in a deterministic, parameter-free manner. For position t and dimension d, the positional encoding is defined as

$$\mathrm{PE}(t,2d) = \sin\left(\frac{t}{10000^{2d/d_{\text{model}}}}\right), \quad \mathrm{PE}(t,2d+1) = \cos\left(\frac{t}{10000^{2d/d_{\text{model}}}}\right).$$

This formula generates a d_{model}-dimensional vector for each position t with values bounded in $[-1, 1]$. The even dimensions use sine, the odd dimensions use cosine, and the frequency decreases exponentially with dimension index (higher dimensions oscillate more slowly, with wavelengths growing from 2π to $10000 \cdot 2\pi$). The positional encoding PE(t) is added element-wise to the token embedding $\mathbf{e}[t]$ before feeding the combined representation $\mathbf{e}[t] + \text{PE}(t)$ into the first transformer layer. Because the frequencies are different across dimensions forming a geometric sequence, each position receives a unique encoding like a binary counter with smooth interpolation, and the model can learn to extract positional information from these patterns through the attention mechanism. The sinusoidal choice enables the model to generalize to sequence lengths not seen during training because the formula can be evaluated at any integer position t.

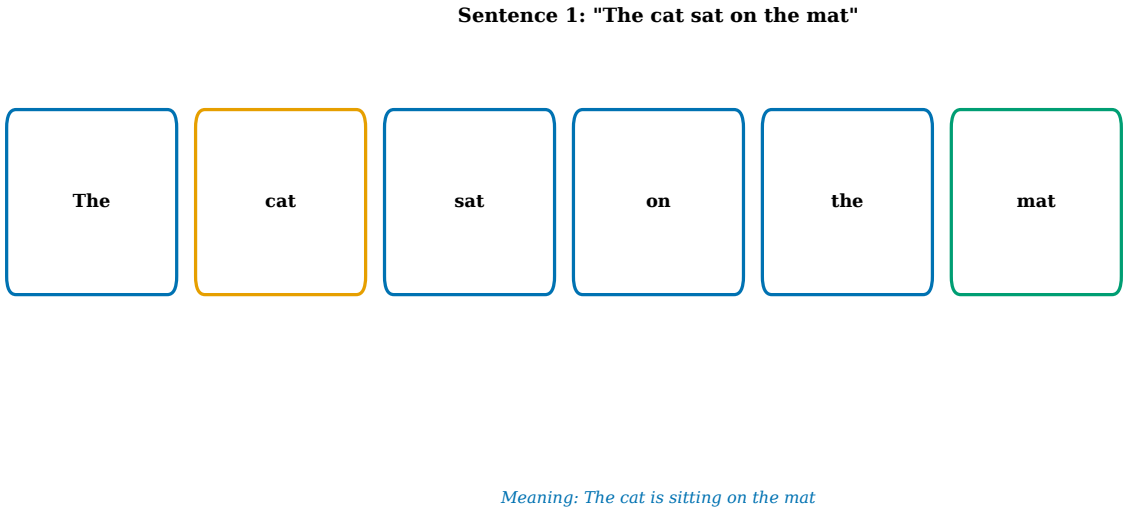


Figure 6.23: Sinusoidal positional encodings for positions 0 to 50 across 128 dimensions. Each row represents a position, each column a dimension. The color intensity indicates the value of the encoding (ranging from -1 to 1). Low dimensions oscillate rapidly, high dimensions oscillate slowly, creating a unique pattern for each position.

Figure 6.23 visualizes the sinusoidal encodings as a heatmap where color intensity represents the encoding value ranging from -1 (dark) to $+1$ (light). The horizontal axis represents dimensions from 0 to d_{model}, the vertical axis represents positions from 0 to the sequence length. The characteristic stripes emerge from the sinusoidal structure: low dimensions (left) show rapid oscillation with short wavelengths, creating many narrow vertical stripes that change quickly with position, while high dimensions (right) show slow oscillation with long wavelengths, creating a few wide stripes that vary smoothly. Each row (position) has a distinct pattern, ensuring that no two positions have identical encodings because the multi-frequency representation provides a unique signature. The sinusoidal structure also has a useful mathematical property: the encoding at position $t + k$ can be expressed as a linear function of the encoding at position t using rotation matrices, which might help the model learn to attend to relative positions through learned linear transformations. This property means that the model can potentially learn translation-invariant patterns in the positional information, recognizing dependencies based on distance k rather than absolute position t. The mathematical structure provides an inductive bias toward relative positioning while still encoding absolute position information.

The addition of positional encodings in Figure 6.24 is a simple but critical step that combines content and position information into a unified representation. The token embedding $\mathbf{e}[t]$ encodes what the word is (its semantic and syntactic properties learned from co-occurrence patterns), while the positional encoding PE(t) encodes where it is in the sequence (its absolute position from the start). By adding them element-wise as $\mathbf{e}[t] +$

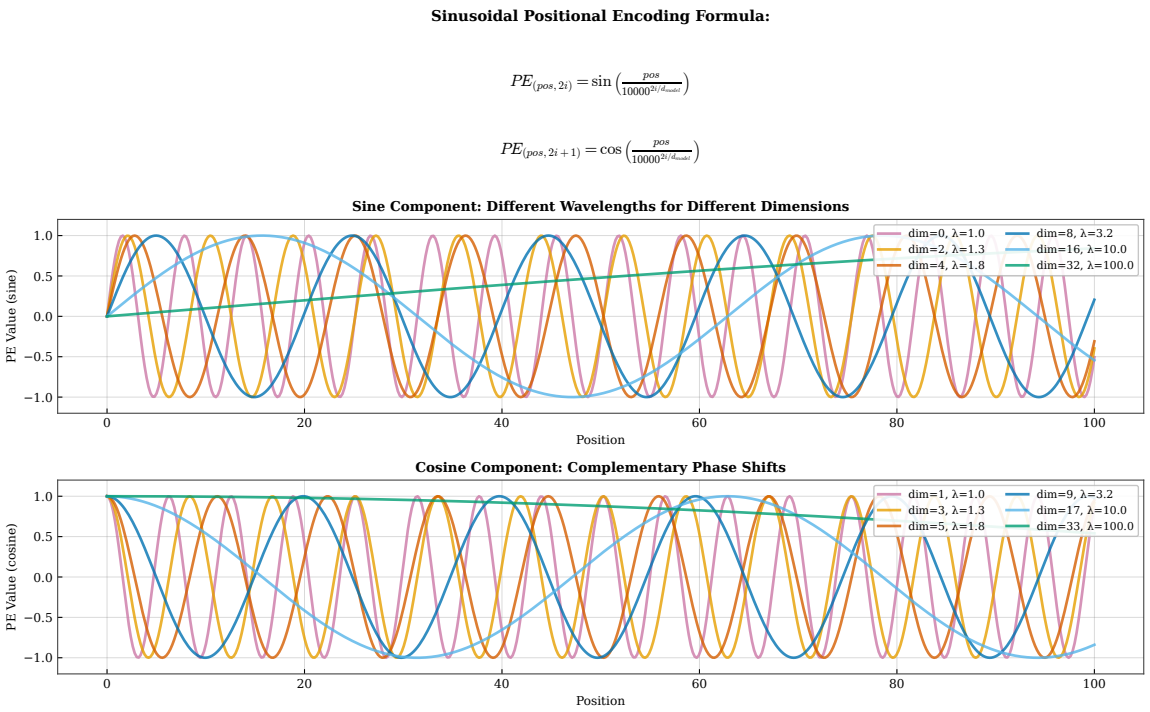


Figure 6.24: Adding positional encodings to token embeddings. Each token embedding $\mathbf{e}[t]$ (representing word identity) is combined element-wise with the positional encoding PE(t) (representing position in the sequence). The resulting vector contains both content and position information.

PE(t), we create a combined representation that the model can use to make position-dependent predictions, with the two sources of information superimposed in the same vector space. For example, the subject of a sentence typically appears early, so a model might learn to attend more to early positions when predicting verbs that need to agree with that subject by detecting the positional signature in the combined representation. The addition means that the model must learn to disentangle content from position through its learned projections, but this appears to be learnable in practice: trained transformers successfully extract both types of information from the combined representation. The element-wise addition preserves the dimension d_{model}, allowing the combined vector to flow through the transformer layers without dimension changes. The model learns through backpropagation on next-word prediction loss which combinations of content and position information are most predictive for the task.

The three-dimensional surface in Figure 6.25 emphasizes the wave-like structure of sinusoidal encodings when rendered as a height field over the position-dimension plane. The ripples run diagonally because the wavelength depends on dimension: low dimensions have short wavelengths (many peaks and troughs across the position axis, oscillating every few positions), while high dimensions have long wavelengths (smooth variation across positions, barely changing over the typical sequence length). This diversity in frequencies ensures that each position's encoding is unique even at long sequence lengths, as the multi-frequency signature is distinct for each integer position. The sinusoidal encodings are deterministic and data-independent: they are the same for every sequence and every training example, which makes them a form of prior knowledge about position rather than a learned representation. This determinism has both advantages (no parameters to learn, no risk of overfitting positional patterns, can generalize to arbitrary sequence lengths) and disadvantages (cannot adapt to task-specific positional biases that might favor certain positions). The choice of base frequency 10000 was empirically determined in the original transformer paper by Vaswani et al. through experimentation on machine translation tasks with typical sequence lengths around 100-200 tokens.

Figure 6.26 compares sinusoidal encodings with learned positional embeddings, two fundamentally different approaches to injecting position information. In the learned approach, we treat the positional encodings as

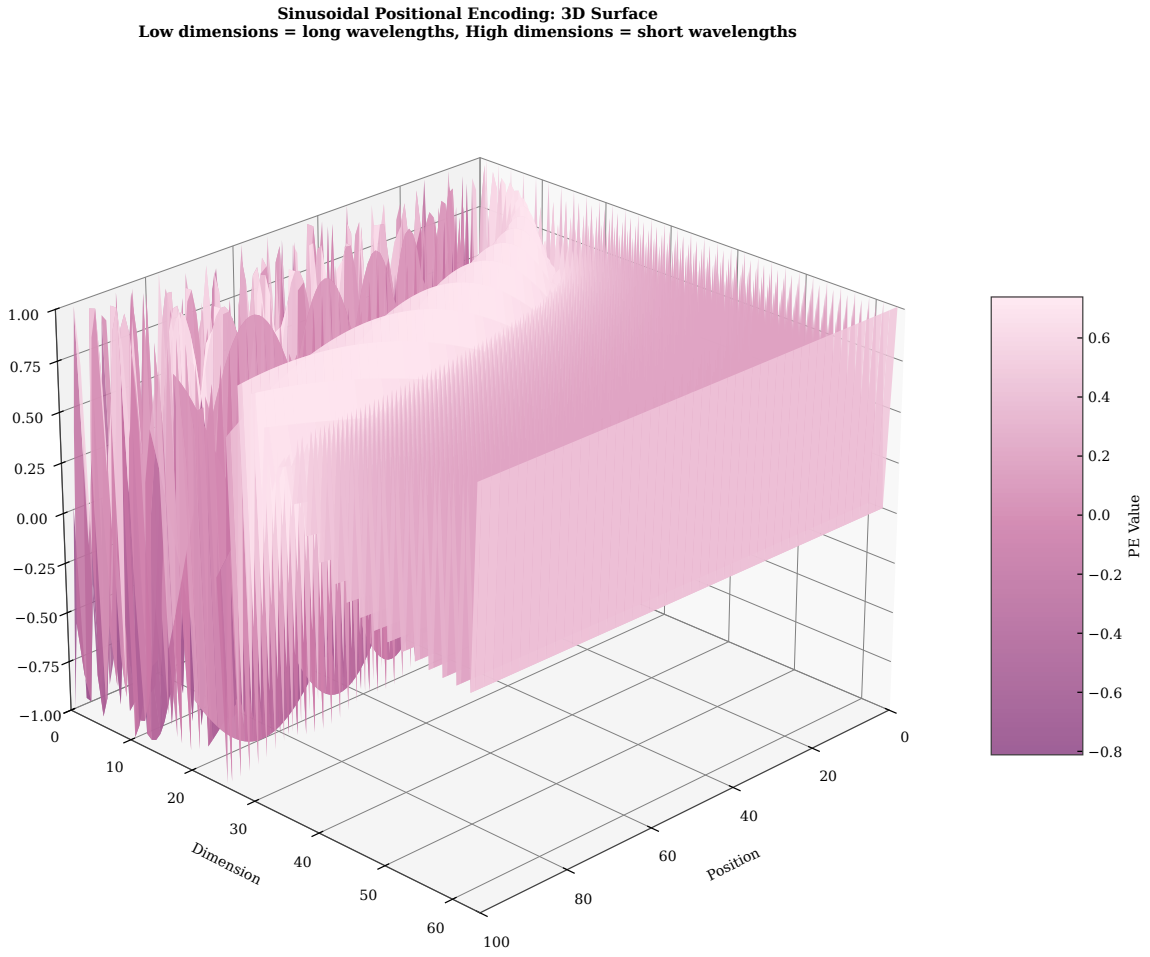


Figure 6.25: Three-dimensional view of sinusoidal positional encodings. The surface shows how the encoding value varies across position (x-axis) and dimension (y-axis). The rippling pattern reflects the sinusoidal structure, with wavelength decreasing as dimension increases.

parameters: we initialize a d_{model}-dimensional vector for each position (up to some maximum sequence length $T_{\max}$) and update these vectors during training via backpropagation, giving $T_{\max} \cdot d_{\text{model}}$ additional parameters. Learned encodings can adapt to the specific patterns in the training data such as typical sentence lengths or paragraph boundaries, but they do not inherently generalize to longer sequences than seen during training. If the model is trained on sequences of length 512 and we try to run it on a sequence of length 1024, the learned encodings for positions 513 onward do not exist and cannot be constructed without additional techniques. Sinusoidal encodings, by contrast, can be computed for any position via the closed-form formula, allowing perfect generalization to arbitrary lengths without extrapolation artifacts. Empirically, both approaches yield comparable performance (similar perplexity) on tasks where sequence lengths at training and test time are similar. Some modern models use learned encodings during training but interpolate or extrapolate them at test time to handle longer sequences through position interpolation techniques. The choice between learned and sinusoidal depends on the expected length distribution at inference time and whether length generalization is required.

Rotary Position Embedding (RoPE), shown in Figure 6.27, is an alternative positional encoding scheme that has become popular in recent large language models including LLaMA, PaLM, and GPT-NeoX. Instead of adding positional information to the embeddings, RoPE applies a rotation to the query and key vectors before computing attention scores, treating pairs of dimensions as 2D vectors in the complex plane. The rotation angle is proportional to the position times a frequency θ_d, so the dot product $\mathbf{q}[i]^\top \mathbf{k}[j]$ becomes a function of the relative position $i - j$ rather than the absolute positions i and j through the trigonometric identity $\cos(\theta_i - \theta_j)$.

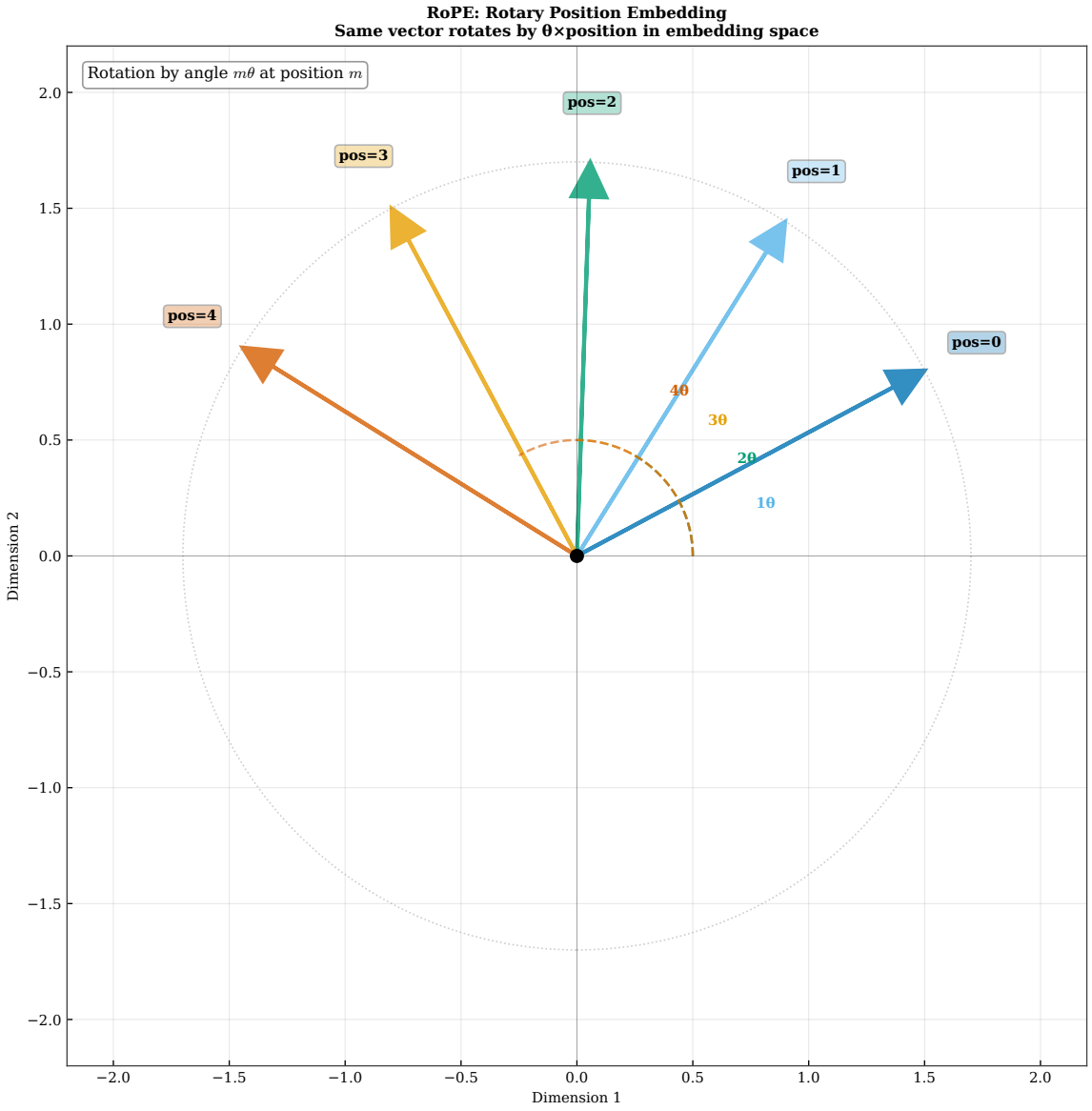


Figure 6.26: Comparison of sinusoidal (left) and learned (right) positional encodings. Learned encodings are treated as parameters and optimized via gradient descent. Both approaches yield similar performance in practice, but sinusoidal encodings generalize better to sequence lengths not seen during training.

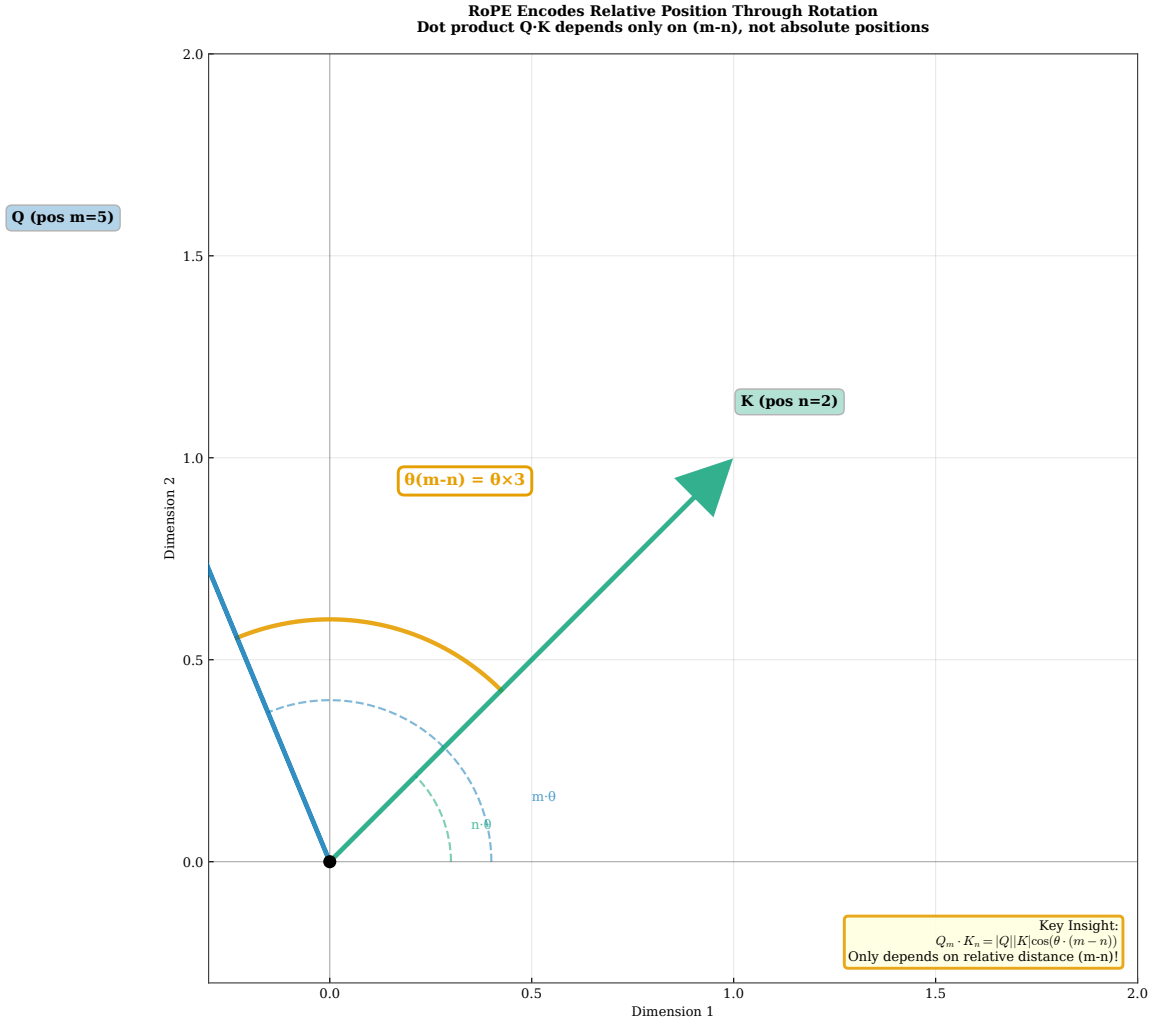


Figure 6.27: Rotary Position Embedding (RoPE) applies position information via rotation in the complex plane. Instead of adding positional encodings to embeddings, RoPE rotates the query and key vectors by an angle proportional to their positions. The relative rotation between positions i and j depends only on their distance $i - j$, naturally encoding relative position.

This relative encoding is appealing because many linguistic phenomena depend on distance rather than absolute position in the sequence. For example, subject-verb agreement typically involves the nearest subject, not the subject at a specific absolute position, so encoding relative distance is more natural. RoPE has been shown empirically to improve performance on tasks requiring long-range dependencies and to generalize better to longer sequences than sinusoidal or learned encodings, making it the preferred choice in many state-of-the-art models. The rotation-based approach provides strong inductive biases for relative positional relationships while maintaining the ability to distinguish absolute positions through multi-frequency rotations.

Figure 6.28 summarizes empirical results comparing positional encoding methods on language modeling perplexity across different sequence lengths. Within the training sequence length (left region), all methods perform similarly with comparable perplexity scores, indicating that the model can learn to use positional information effectively regardless of the encoding scheme when operating within distribution. The differences emerge when we extrapolate to longer sequences (right region) beyond what was seen during training. Sinusoidal encodings maintain reasonable performance because the formula naturally extends to any length via the closed-form computation without requiring new parameters. Learned encodings fail catastrophically because no embeddings exist for the new positions beyond the training maximum $T_{\max}$, causing undefined behavior or requiring extrapolation heuristics. RoPE often outperforms sinusoidal encodings in the extrapolation regime

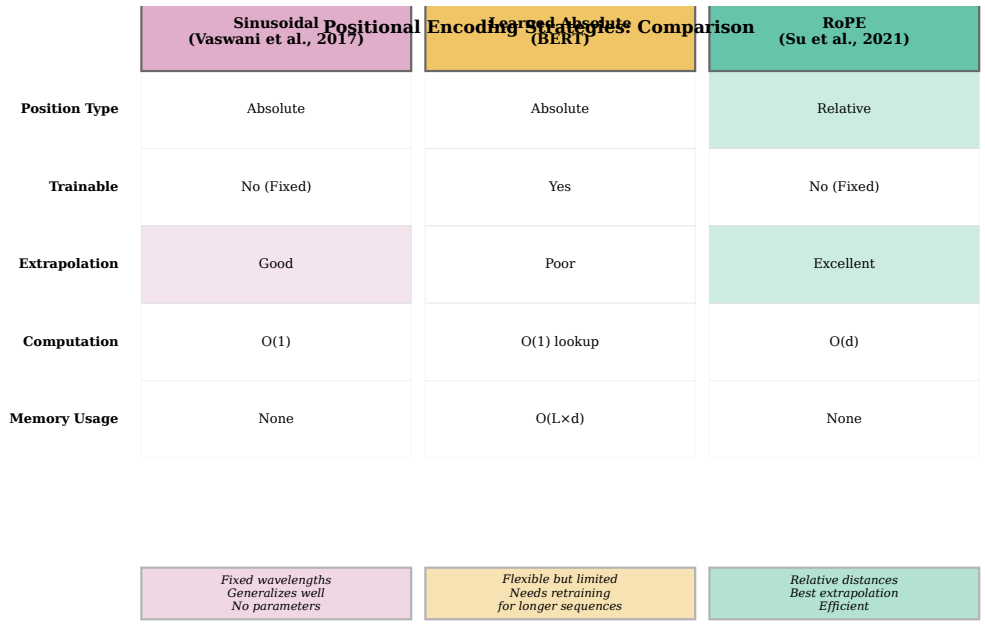


Figure 6.28: Performance comparison of different positional encoding schemes on a language modeling task. Sinusoidal, learned, and RoPE achieve similar perplexity on in-distribution sequence lengths, but RoPE generalizes better when evaluated on longer sequences than seen during training (extrapolation region on the right).

because its relative encoding better matches the inductive bias of language: dependencies depend on distance $i - j$, not absolute position i or j. Modern large language models (such as GPT-NeoX, LLaMA, and PaLM) frequently use RoPE for this reason, combined with techniques like position interpolation for even longer contexts. The choice of positional encoding can significantly impact a model's ability to handle long-context tasks and generalize to sequences longer than those seen during training, making it a critical architectural decision.

6.6 Context Representation in Transformers

The transformer's approach to context representation differs fundamentally from the recurrent models examined in Chapter **??**, trading sequential processing for parallel access. An RNN compresses all previous context into a single fixed-dimensional hidden state $\mathbf{h}[t]$, which is updated sequentially as each new token arrives through the recurrence relation $\mathbf{h}[t] = f(\mathbf{h}[t-1], \mathbf{e}[t])$. This compressed representation is efficient in memory (only one vector of dimension d_{model} per position) but lossy in information (the fixed dimension cannot represent unbounded history without lossy compression, limiting the mutual information between hidden state and history). A transformer, by contrast, maintains separate representations for all positions in the sequence and uses attention to compute weighted combinations of these representations based on learned relevance. The context representation is therefore distributed across many vectors rather than compressed into one, with each position maintaining its own d_{model}-dimensional representation. When predicting $w[t]$, the transformer has direct access to the embeddings $\mathbf{e}[1], \mathbf{e}[2], \ldots, \mathbf{e}[t-1]$, augmented with positional information and transformed by layers of attention and feed-forward networks that refine the representations. This distributed approach trades memory for expressiveness, enabling richer representations at the cost of higher memory consumption proportional to sequence length.

How This Chapter Represents Context:
Unlike RNNs that compress all history into a fixed-size hidden state, Transformers represent context as weighted combinations of all previous positions. Each position maintains its own representation, and attention mechanisms dynamically select which positions are relevant. This distributed representation avoids the information bottleneck, allowing the model to preserve fine-grained details about specific earlier words. The causal masking ensures that context respects temporal order, while multi-head attention enables multiple simultaneous views of relevance. Positional encodings inject sequence order into an otherwise permutation-invariant mechanism. The result is a rich, adaptive context representation that scales naturally to long sequences and captures diverse dependencies through learned attention patterns.

A decoder-only transformer language model consists of a stack of n_{layers} identical transformer blocks, each refining the representations from the previous layer. Each block contains two main components: a multi-head self-attention layer and a position-wise feed-forward network, both wrapped with residual connections and layer normalization for training stability. The input to the first block is the sum of token embeddings and positional encodings $\mathbf{e}[t] + \text{PE}(t)$. Each subsequent block receives as input the output of the previous block, building increasingly abstract representations. At the top of the stack, a final linear layer and softmax produce a probability distribution over the vocabulary $\mathscr{V}$ for next-word prediction. Formally, denoting the input to layer ℓ as $\mathbf{h}^{(\ell)}$, the layer computes

$$\mathbf{h}^{(\ell+1)} = \text{LayerNorm}\left(\mathbf{h}^{(\ell)} + \text{FFN}\left(\text{LayerNorm}\left(\mathbf{h}^{(\ell)} + \text{MHA}(\mathbf{h}^{(\ell)})\right)\right)\right),$$

where MHA is multi-head self-attention with causal masking, FFN is a position-wise feed-forward network (typically two linear layers with a nonlinearity like GELU in between, with hidden dimension d_{ff} typically $4 \cdot d_{\text{model}}$), and LayerNorm is layer normalization. The residual connections (the addition of $\mathbf{h}^{(\ell)}$) allow gradients to flow directly through the network via skip connections, mitigating vanishing gradient problems that plagued deep networks before this innovation. The layer normalization stabilizes training by normalizing activations to have zero mean and unit variance across the feature dimension.

Figure 6.29 shows the internal structure of one transformer block, with data flowing from bottom to top through the component stack. The multi-head attention layer allows each position to aggregate information from previous positions through learned attention weights, computing contextual representations. The feed-forward network then applies a non-linear transformation independently to each position with shared weights across positions, allowing the model to compute complex functions of the aggregated context through the two-layer MLP structure with hidden dimension d_{ff}. The residual connections ensure that information from lower layers can bypass the attention and feed-forward transformations if needed via additive skip connections, providing a gradient highway and making very deep networks trainable without vanishing gradients. The layer normalization prevents activations from growing or shrinking uncontrollably as they pass through many layers, stabilizing training dynamics by keeping activations well-scaled throughout the forward pass. Each component plays a crucial role: attention for context aggregation across positions, feed-forward for non-linear feature transformation, residuals for gradient flow enabling depth, and normalization for numerical stability during training. These components work together synergistically to enable deep, trainable architectures for language modeling with dozens or hundreds of layers.

The stacking of layers in Figure 6.30 creates a hierarchical representation that progressively abstracts from surface forms to deeper linguistic structure. Lower layers (near the input) tend to learn surface-level patterns such as part-of-speech tagging, syntactic parsing, and local collocations that depend primarily on adjacent words. Higher layers (near the output) tend to learn more abstract patterns such as semantic roles, discourse structure, and long-range dependencies that require integrating information from distant positions. This hierarchical organization emerges automatically from training on next-word prediction: to predict the next word accurately, the model must extract increasingly abstract features from the input sequence through the compositional structure of language. Empirical studies using probing classifiers have confirmed that different layers encode different types of linguistic information, with the highest layers most predictive of the final next-word distribution and lower layers better for syntactic tasks. The depth of the network allows the model to build representations of increasing sophistication, similar to how convolutional networks learn hierarchical visual

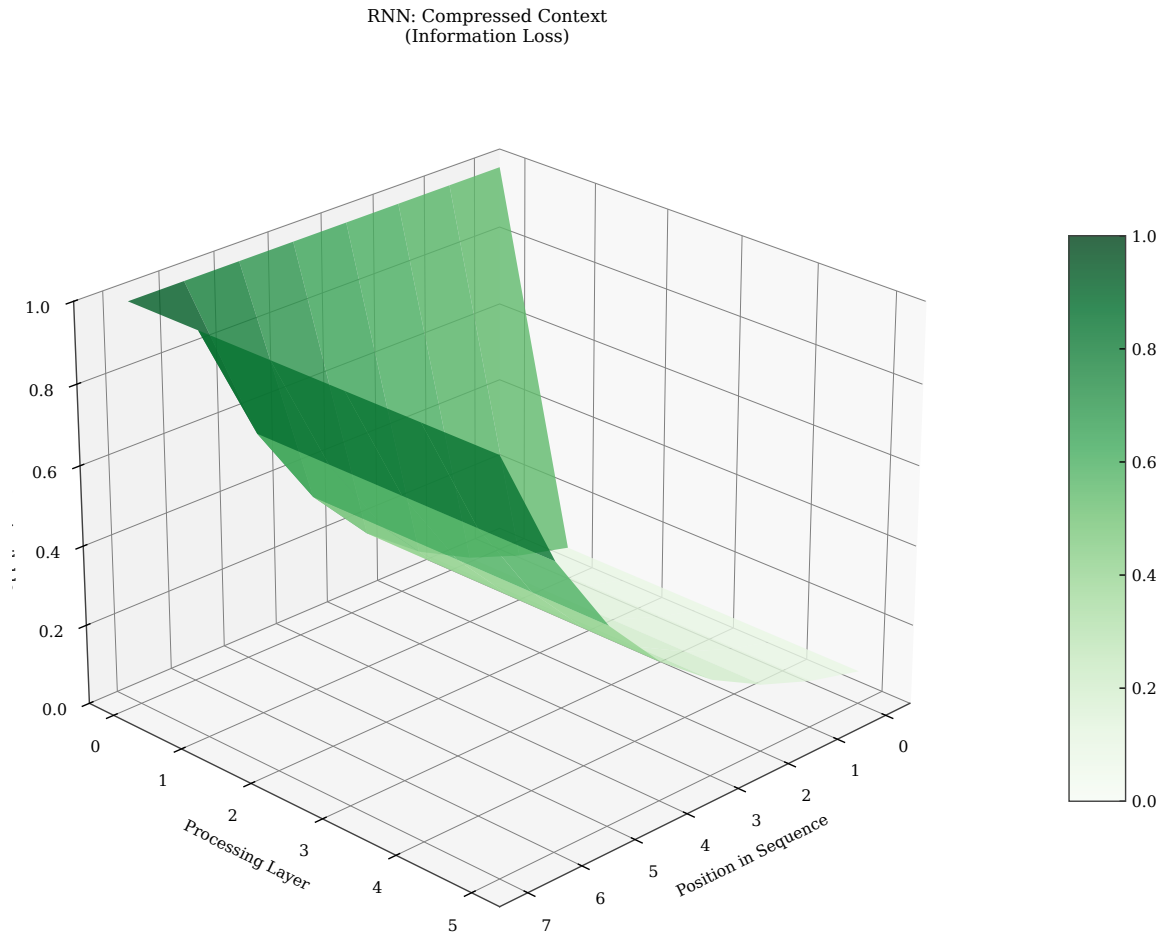
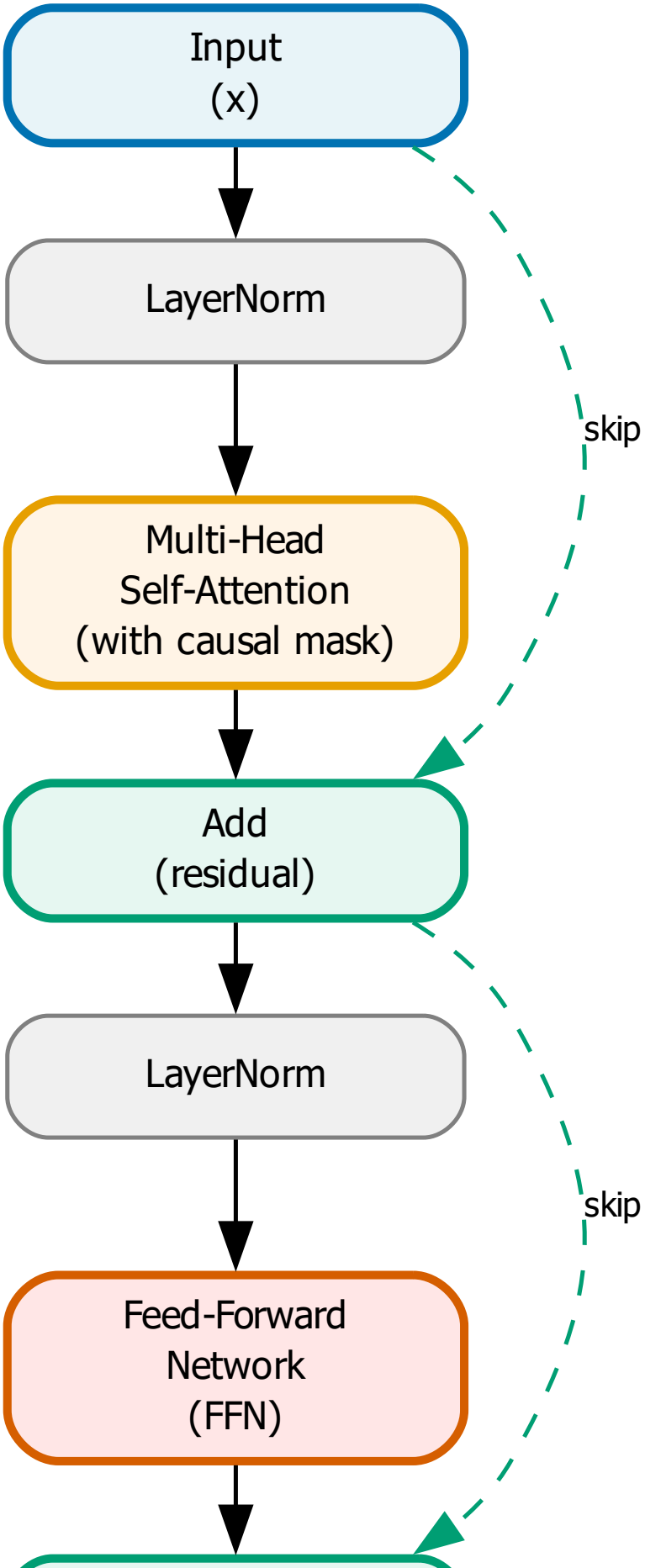


Figure 6.29: A single transformer decoder block. The input passes through multi-head self-attention (with causal masking), residual connection, and layer normalization, then through a position-wise feed-forward network, residual connection, and layer normalization. The output is passed to the next block.

features from edges to textures to objects. Deep transformers with many layers (e.g., 12, 24, 96, or more) can capture very complex linguistic patterns through this hierarchical processing, with each layer refining the representations further.

Figure 6.31 visually contrasts the context representations of RNNs and transformers using 3D surfaces to show how information is stored. The RNN's hidden state is a single trajectory through representation space, updated sequentially with each new token. All information about positions 1 through $t-1$ must be encoded in $\mathbf{h}[t-1]$, a single point in d_{model}-dimensional space with information capacity bounded by d_{model}. The transformer's representation is a grid: position t has a d_{model}-dimensional representation, position $t-1$ has a separate d_{model}-dimensional representation, and so on for all positions independently. This grid preserves more information with total capacity $T \cdot d_{\text{model}}$ but requires $O(T \cdot d_{\text{model}})$ memory instead of $O(d_{\text{model}})$. The trade-off is memory for expressiveness: transformers can remember fine-grained details about specific positions that RNNs must discard or compress when new tokens arrive. Modern hardware architectures with large memory capacity (tens to hundreds of gigabytes) make this trade-off increasingly favorable for longer sequences. The parallel structure also enables much faster training on GPUs compared to sequential RNN processing because all positions can be computed simultaneously. This architectural difference fundamentally determines what patterns each model type can learn effectively and how they scale.

Applying a transformer to our running example in Figure 6.32, we see that the final representation at position 23 has been enriched by multiple rounds of attention across all 22 preceding words through the layer


Input
(x)
LayerNorm
skip
Multi-Head
Self-Attention
(with causal mask)
Add
(residual)
LayerNorm
skip
Feed-Forward
Network
(FFN)

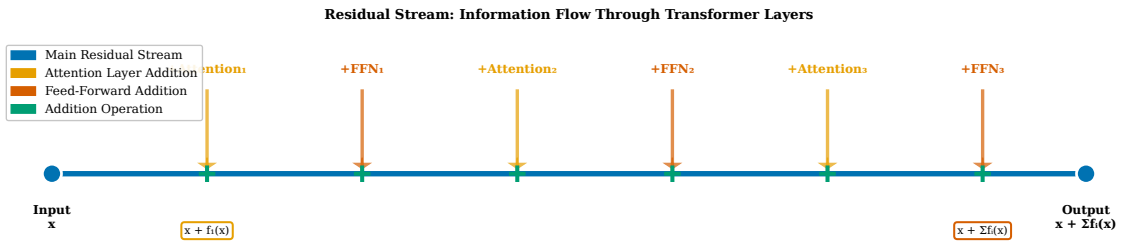


Figure 6.31: Three-dimensional comparison of context representations. The RNN (left surface) compresses all context into a single hidden state trajectory that evolves over time. The transformer (right surface) maintains a separate representation for each position, creating a grid of context vectors. The transformer's surface is richer but requires more memory.

stack. The first layer might have learned to attend to adjacent words, capturing local syntax and collocations like "the code" and "would" that form immediate bigram context. The second layer might have learned to attend to the subject and main verb, capturing sentence structure and grammatical dependencies that span across intervening clauses. The third layer might have learned to attend to semantically related content words like "software," "engineer," and "submitted," capturing topical coherence and thematic consistency within the programming domain. By the final layer, the representation at position 23 encodes a rich summary of all relevant context, distributed across multiple attention heads and integrated through feed-forward transformations into a unified d_{model}-dimensional vector. This representation is then fed into the output layer, which computes $P(w[23] \mid w[1:22])$ as a softmax distribution over $\mathcal{V}$ by applying a final linear projection and normalization. The transformer has successfully predicted the next word by directly accessing and weighting all available context through learned attention patterns, enabling accurate generation of contextually appropriate completions like "compile," "execute," or "solve."

We can now predict better because:

- Direct access to any position enables long-range dependencies
- Attention weights learn which context is relevant
- Parallel processing allows efficient training
- Positional encodings preserve sequence order

Next: Chapter **??** explores how we generate text from these predictions, examining sampling strategies, beam search, and the trade-offs between diversity and quality in autoregressive generation.

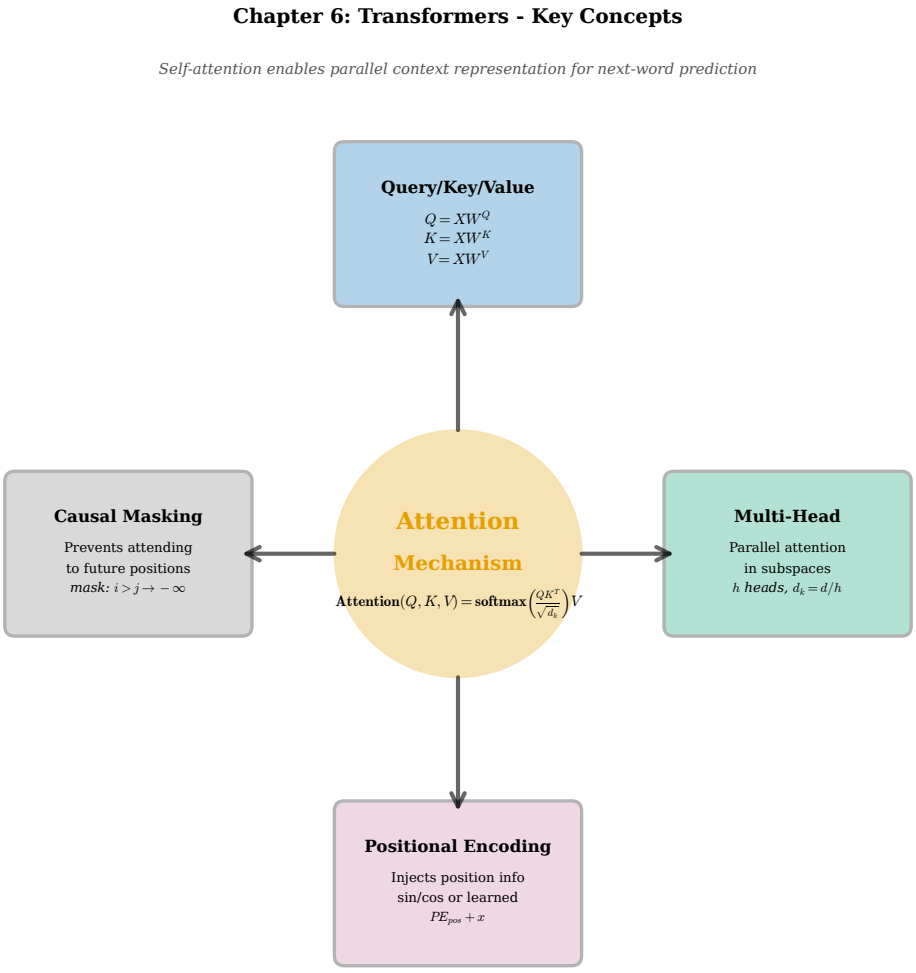


Figure 6.32: Final representation of our running example in a transformer. Each position has a context-enriched representation produced by multiple layers of attention. Position 23 (where we predict the next word) has attended to relevant words like “code”, “engineer”, and “submitted”, integrating their information into a representation that supports accurate next-word prediction.

Exercises

1. Consider a sequence of length $T = 5$ with model dimension $d_{\text{model}} = 4$. Write out the explicit computation of the attention weights $\alpha[32]$ and $\alpha[33]$ for position $i = 3$, assuming the query $\mathbf{q}[3] = [1, 0, -1, 0]$ and the keys $\mathbf{k}[1] = [1, 1, 0, 0]$, $\mathbf{k}[2] = [0, 1, 1, 0]$, $\mathbf{k}[3] = [-1, 0, 1, 1]$. Show the scaling, exponentiation, and normalization steps.

2. Prove that the attention output $\sum_{j=1}^{i} \alpha[ij]\mathbf{v}[j]$ is a convex combination of the value vectors when $\alpha[ij]$ is computed using softmax. What geometric interpretation does this give to the attention output?

3. Explain why causal masking is necessary for training autoregressive language models. What would happen if we trained a transformer without causal masking on next-word prediction, allowing each position to attend to all positions including future ones? Describe the failure mode in detail.

4. Given a transformer with $d_{\text{model}} = 512$, $n_{\text{heads}} = 8$, and $n_{\text{layers}} = 12$, calculate the total number of parameters in the multi-head attention modules (queries, keys, values, and output projection) across all layers. Ignore biases and other components like feed-forward networks and layer normalization.

5. Compute the sinusoidal positional encoding $\text{PE}(10, 0)$, $\text{PE}(10, 1)$, $\text{PE}(10, 2)$, $\text{PE}(10, 3)$ for position $t = 10$ and the first four dimensions, assuming $d_{\text{model}} = 128$. Verify that the encodings for dimensions 0 and 1 (sine and cosine at the same frequency) satisfy $\text{PE}(t, 0)^2 + \text{PE}(t, 1)^2 = 1$.

6. Rotary Position Embedding (RoPE) encodes relative position by rotating query and key vectors. Explain intuitively why rotating $\mathbf{q}[i]$ by angle θ_i and $\mathbf{k}[j]$ by angle θ_j causes the dot product $\mathbf{q}[i]^\top \mathbf{k}[j]$ to depend on the relative position $i - j$ rather than the absolute positions i and j. (Hint: consider the angle difference in the rotation.)

7. For our running example sentence "The software engineer who had been working remotely from the mountain cabin for several months finally submitted the code that would _____," identify three word pairs (one word in the context, one potential next word) where high attention weight would be justified. For each pair, explain what type of dependency (syntactic, semantic, or discourse) the attention captures.

8. Compare the memory complexity of RNNs and transformers for processing a sequence of length T with model dimension d_{model}. An RNN stores one hidden state of dimension d_{model}, while a transformer stores representations for all T positions. For what sequence length T does the transformer's memory usage become 10 times larger than the RNN's, assuming both use the same d_{model}?

9. **(Advanced)** In multi-head attention, different heads learn different attention patterns. Design a diagnostic task or probing method to determine whether a specific head has specialized for a particular linguistic phenomenon (e.g., subject-verb agreement, anaphora resolution, or semantic similarity). Describe what inputs you would provide and what outputs you would measure.

10. **(Advanced)** The scaling factor $1/\sqrt{d_{\text{model}}}$ in attention was motivated by controlling the variance of dot products. Derive the variance of the dot product $\mathbf{q}^\top \mathbf{k}$ assuming $\mathbf{q}$ and $\mathbf{k}$ are random vectors with i.i.d. entries drawn from a distribution with mean 0 and variance σ^2. Show that without scaling, the variance grows linearly with d_{model}.

11. **(Advanced)** Transformers process all positions in parallel, while RNNs process them sequentially. Compare the computational complexity (in big-O notation) of processing a sequence of length T with model dimension d_{model} for both architectures. Consider both the forward pass and the backward pass, and discuss the trade-off between parallelism and total computation.

12. **(Advanced)** Consider an ablation study where you remove one of the following components from a trained transformer: (a) causal masking, (b) positional encoding, (c) multi-head attention (replacing it with single-head attention), or (d) residual connections. For each ablation, predict the most likely failure mode or performance degradation on a language modeling task. Justify your predictions based on the role each component plays.

Bibliography

Index