Predicting the Next Word

From Shannon to ChatGPT

Test Compilation

Contents

Chapter 1

Introduction: The Problem of Prediction

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

- Defining the fundamental question: What word comes next?
- Connecting language modeling to information theory
- Establishing perplexity as our evaluation metric
- Understanding why prediction difficulty varies with context

1.1 What is Next-Word Prediction?

Consider the sentence fragment:

"The cat sat on the _____"

What word comes next? Most English speakers would guess "mat," "floor," "couch," or similar nouns describing surfaces where a cat might reasonably sit. This simple task—predicting the next word given preceding context—lies at the heart of language modeling and, more broadly, modern natural language processing. The question appears deceptively simple, yet answering it well requires capturing the full complexity of human language: syntactic rules that determine which word categories can follow "the," semantic knowledge that cats sit on physical surfaces rather than abstract concepts, and world knowledge that mats are common resting places for household pets. Every time a language model produces a response, translates a sentence, completes a search query, or generates code, it is fundamentally answering this question of what word or token should come next. The applications of accurate next-word prediction extend far beyond autocomplete: machine translation systems generate target language text one word at a time, speech recognition systems score hypothesized transcriptions by their language model probability, and modern AI assistants synthesize coherent multi-paragraph responses through iterated next-word prediction. Understanding this core task is thus essential for understanding the entire field of natural language processing.

A **language model** assigns probabilities to sequences of words, quantifying how likely each possible continuation is given the words that have come before. Given some context, the model tells us which continuations are likely and which are not, effectively encoding our expectations about language in a mathematical form that computers can manipulate. The "cat sat on the" context strongly favors surfaces over actions, and common surfaces over rare ones, because that is the pattern we observe in natural English text—the statistical regularities of language become encoded in the model's parameters. This probability assignment must satisfy the axioms of probability theory: all probabilities must be non-negative, and they must sum to exactly one over all possible next words, ensuring we have a valid distribution. The quality of a language model is determined by how well its probability assignments match the actual patterns of language: a good model assigns high probability to sequences that native speakers would find natural and low probability to sequences that are awkward

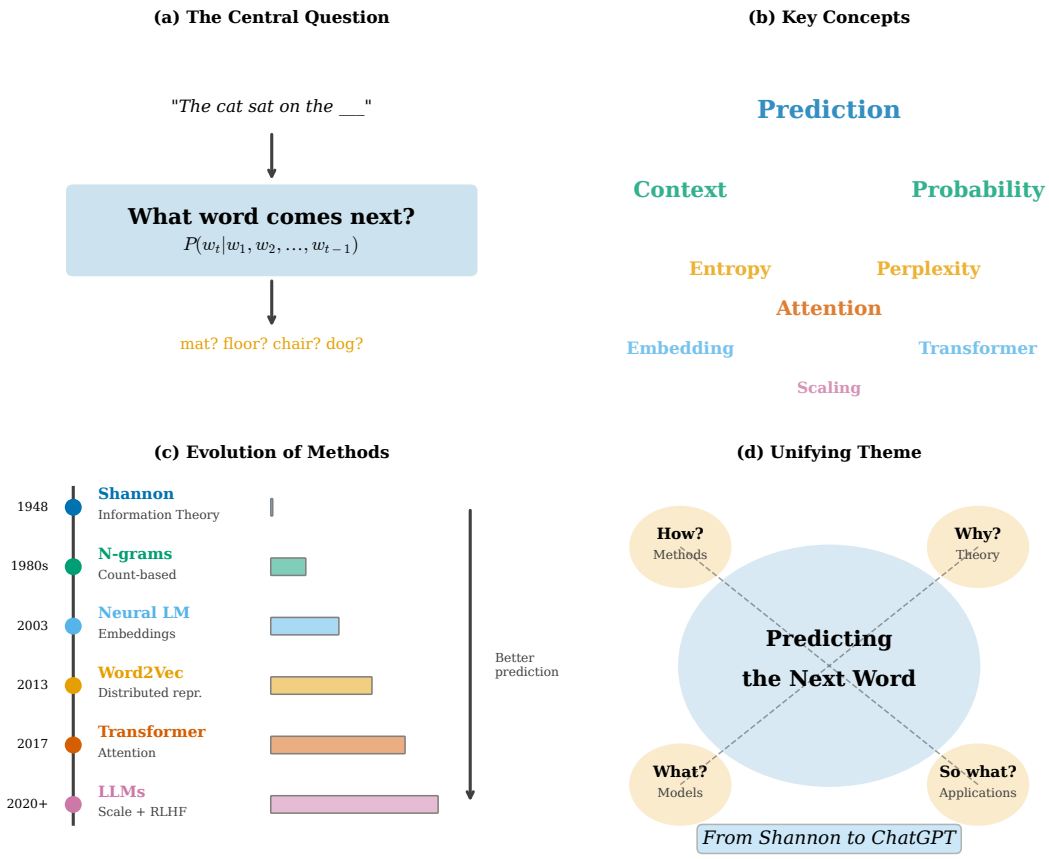


Figure 1.1: The central question of this book: predicting the next word. Panel (a) shows the core prediction problem with candidate words. Panel (b) displays key concepts that form the foundation of language modeling. Panel (c) traces the evolution from Shannon's information theory (1948) through n-grams, neural language models, Word2Vec, Transformers, to modern LLMs. Panel (d) presents the unifying theme connecting theory, methods, models, and applications.

or ungrammatical. Training a language model means adjusting its parameters so that its probability estimates become increasingly accurate on observed text.

Formally, we seek to model the probability distribution over the vocabulary $\mathcal{V}$ given a sequence of preceding words (the *context*). This mathematical formulation captures the essence of the language modeling problem: given everything that has been written or spoken so far, what word is most likely to come next? The vocabulary $\mathcal{V}$ represents the set of all possible words or tokens that the model can predict, typically ranging from tens of thousands of words for word-level models to hundreds of thousands of subword units for modern tokenizers. The context comprises all preceding words in the sequence, though as we shall see, different architectures handle context in fundamentally different ways—some considering only a fixed window of recent words, others attempting to capture the entire preceding history. The conditional probability distribution must satisfy the axioms of probability: every probability must be non-negative, and the probabilities over all possible next words must sum to exactly one, ensuring we have a valid distribution from which we could, in principle, sample the next word. This constraint shapes the design of neural language models, which typically use a softmax function to transform unconstrained outputs into valid probability distributions:

$$P(w_t \,|\, w_1, w_2, \ldots, w_{t-1}) \tag{1.1}$$

where $w_t \in \mathcal{V}$ is the target word and $w_1, \ldots, w_{t-1}$ is the context.

This probability distribution captures the structure of language at multiple levels, each contributing different constraints on what words may follow. At the syntactic level, grammar imposes strict ordering requirements: after the article "the," English syntax demands nouns, adjectives, or other noun phrase constituents, effectively

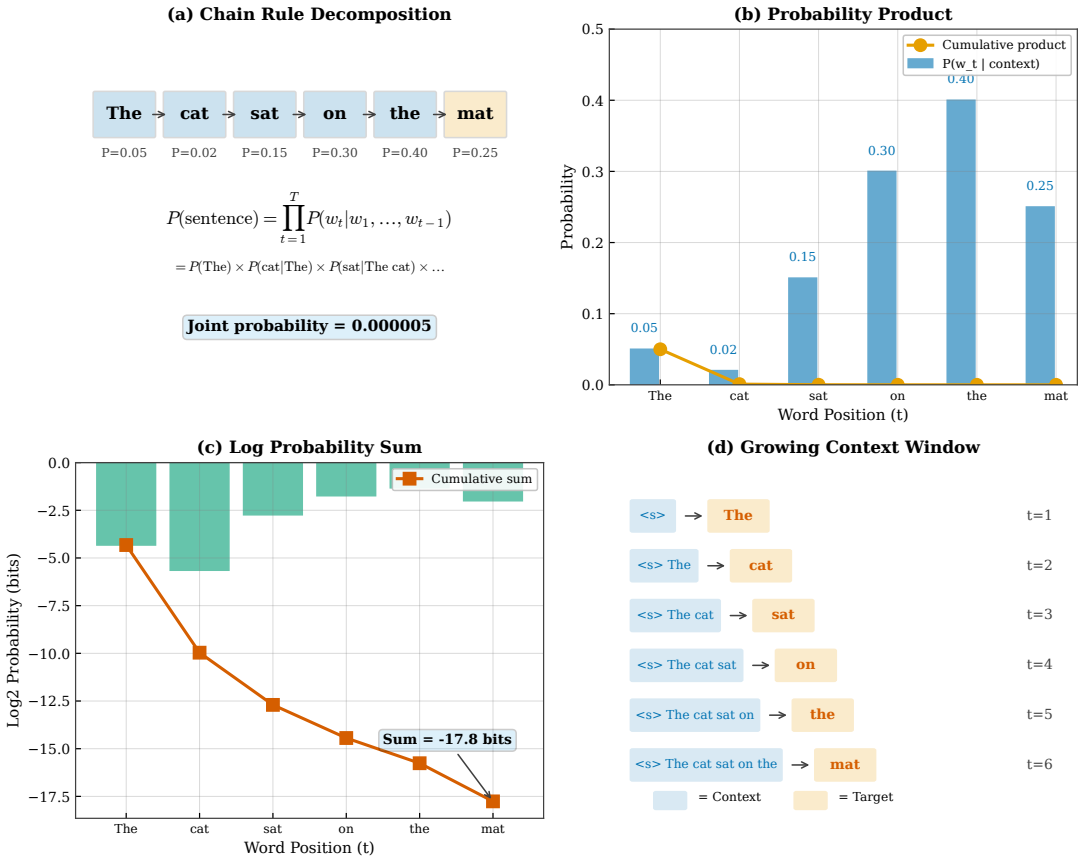


Figure 1.2: The chain rule decomposition of joint probability. Panel (a) shows how a sentence probability factors into conditional probabilities. Panel (b) illustrates the growing context at each position. Panel (c) demonstrates the multiplication of probabilities. Panel (d) shows numerically how small probabilities compound.

ruling out verbs, prepositions used alone, or conjunctions. These syntactic constraints arise from the hierarchical structure of language and dramatically reduce the space of plausible continuations at each position. Beyond syntax, semantic coherence further narrows predictions based on meaning. Given the phrase "cat sat on," we expect words denoting surfaces or locations—"mat," "floor," "windowsill"—rather than actions or abstract concepts, because the semantics of "sitting on" requires a physical support surface. World knowledge provides even stronger constraints in certain contexts: the phrase "The capital of France is" strongly predicts "Paris" not because of syntactic or semantic rules, but because of factual knowledge about geography that any competent English speaker possesses. Finally, discourse context tracks entities and relationships across sentences, so pronouns like "it" or "she" must resolve to previously mentioned referents, constraining predictions based on what the text has already established.

Definition 1.1 (Language Model)**.** *A* language model *is a probability distribution* $P(w_1, w_2, \ldots, w_T)$ *over sequences of words from a vocabulary* $\mathcal{V}$*. By the chain rule of probability:*

$$P(w_1, \ldots, w_T) = \prod_{t=1}^{T} P(w_t \,|\, w_1, \ldots, w_{t-1}) \tag{1.2}$$

This decomposition is exact and universal—it applies to any probability distribution over sequences. The challenge lies in estimating each conditional probability $P(w_t \,|\, w_1, \ldots, w_{t-1})$ from finite data. Figure 1.2 illustrates this decomposition visually.

1.1.1 The Chain Rule in Practice

The chain rule tells us that to compute $P(\text{“The cat sat on the mat”})$, we multiply:

$$\begin{aligned} &P(\text{The}) \times P(\text{cat} \mid \text{The}) \times P(\text{sat} \mid \text{The cat}) \\ &\times P(\text{on} \mid \text{The cat sat}) \times P(\text{the} \mid \text{The cat sat on}) \\ &\times P(\text{mat} \mid \text{The cat sat on the}) \end{aligned} \tag{1.3}$$

Each factor in this product requires estimating a conditional distribution over the entire vocabulary, conditioned on all preceding words in the sequence. For sentences of length T with vocabulary size $|\mathcal{V}| = V$, the naive approach would require storing V^T parameters to represent every possible sequence of that length—clearly intractable even for modest vocabulary sizes and sentence lengths. Consider a vocabulary of 50,000 words and sentences of just 10 words: the naive approach would require $50,000^{10}$ parameters, a number far exceeding the estimated 10^{80} atoms in the observable universe. This exponential explosion in the number of possible sequences is sometimes called the “curse of dimensionality,” and it represents the central computational challenge of language modeling. All language modeling techniques, from the simplest n-gram models that consider only local context to the most sophisticated Transformer architectures that attend over thousands of tokens, address this fundamental challenge through parameter sharing and structural assumptions that dramatically reduce the effective number of parameters while still capturing the essential patterns in language. The art of language modeling lies in choosing the right inductive biases—assumptions about which patterns in the data actually matter for prediction—that allow robust generalization from the finite training data to the infinite space of possible sentences that a model may encounter in deployment.

1.2 A Brief History: From Shannon to ChatGPT

The quest to predict the next word has a rich history spanning over seven decades [Jurafsky and Martin, 2024]. Figure 1.3 presents a visual timeline of key developments.

1.2.1 The Information-Theoretic Foundation (1948–1960s)

Claude Shannon’s seminal 1948 paper “A Mathematical Theory of Communication” [Shannon, 1948] introduced the concept of *entropy* as a mathematical measure of uncertainty, providing the theoretical foundation that would underpin all subsequent work in language modeling and communication systems alike. Shannon asked a deceptively simple question: How much information does each character in English text convey? This question connected the structure of language to the mathematics of information transmission, revealing that the redundancy and predictability inherent in natural language could be precisely quantified in bits—the same units used to measure digital information. Shannon’s insight was that uncertainty and information are two sides of the same coin: when we can predict the next symbol with high confidence, that symbol carries little information because it tells us nothing we did not already expect; conversely, when a symbol surprises us by deviating from our expectations, it carries a great deal of information precisely because it could not have been anticipated. This fundamental inverse relationship between predictability and information content remains central to how we understand and evaluate language models today: a model that achieves low entropy has learned to predict well, which means it has captured the statistical structure of the language.

In a famous 1951 experiment [Shannon, 1951], Shannon had human subjects guess letters one at a time in English text, exploiting the predictive capabilities of native speakers to estimate the inherent uncertainty of the language. When subjects guessed incorrectly, they were told the correct letter and continued from that point, building up context that improved subsequent predictions. The experimental methodology was elegant in its simplicity: by counting the number of guesses required at each position and analyzing the distribution of correct responses, Shannon could bound the entropy from both above and below, producing tight estimates of the information content of English. By analyzing how many guesses subjects required on average and applying information-theoretic bounds, Shannon estimated that English has approximately 1.0–1.5 bits of entropy per character—meaning each character conveys roughly one bit of information on average, far below the theoretical

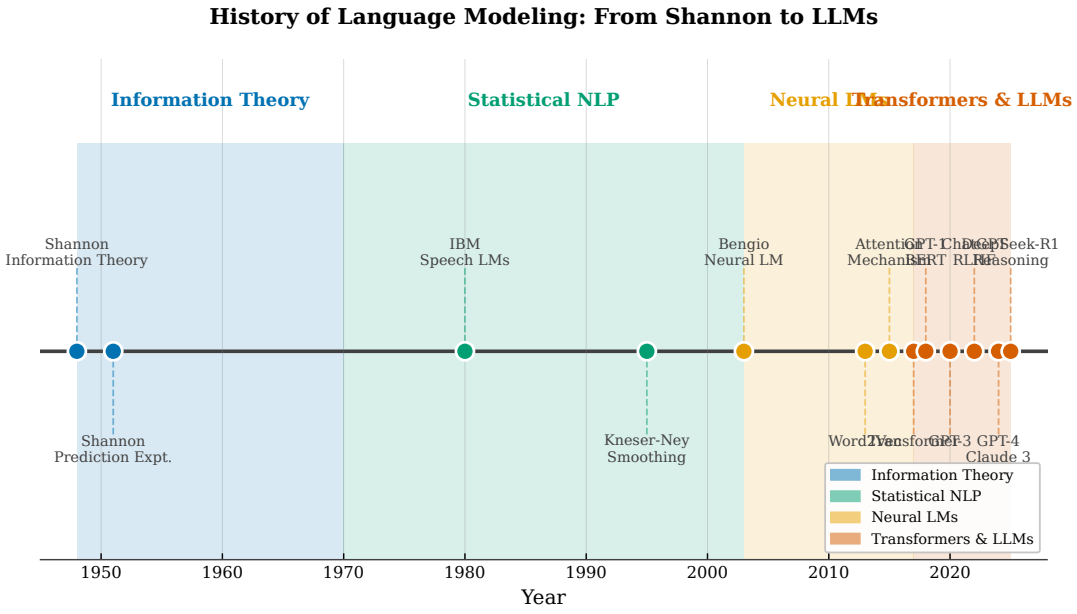


Figure 1.3: History of language modeling from 1948 to 2025. Panel (a) shows the timeline of key milestones. Panel (b) illustrates exponential growth in model parameters. Panel (c) displays corresponding improvements in perplexity. Panel (d) marks paradigm shifts in methodology.

maximum of about 4.7 bits that would be required for a uniformly random selection from the 26-letter alphabet. This finding revealed that English is highly redundant: roughly 75 percent of the information capacity of written text is predictable from context, leaving only about one bit per character of genuine unpredictability. This redundancy is why we can understand text with missing letters, why compression algorithms work so well on natural language, and why language models can achieve such impressive prediction accuracy.

Example 1.1 (Shannon's Guessing Game)**.** *Consider guessing letters in the word "INFORMATION" as shown in Figure 1.4. After observing the prefix "INFORM," a guesser familiar with English word patterns would likely predict "A" as the next letter, recognizing the common morphological pattern leading toward "INFORMATION" or "INFORMAL." Once "INFORMA" has been revealed, the prediction becomes nearly certain: "T" is the only plausible continuation in English, as "INFORMAT" leads uniquely toward "INFORMATION." This example illustrates the fundamental relationship between predictability and information content. When a letter is highly predictable—as "T" is after "INFORMA"—it conveys little new information to the reader, since the reader could have anticipated it with high confidence. Conversely, surprising or unexpected letters carry more information precisely because they could not have been predicted. This inverse relationship between predictability and information content lies at the heart of Shannon's information theory: high predictability corresponds to low entropy, which in turn means less information is conveyed per symbol.*

Shannon's experiments established a fundamental bound that continues to serve as a benchmark for language model evaluation: any language model for English must achieve at least 1.0–1.5 bits per character to match human prediction ability, reflecting the inherent entropy of written English as estimated through human performance. This bound represents a theoretical floor—the uncertainty that remains even when a predictor has access to all available contextual information and perfect knowledge of English, arising from the genuine unpredictability of natural language where multiple continuations may be equally valid. Remarkably, modern large language models approach and sometimes surpass this bound [Brown et al., 2020], achieving bits-per-character rates that rival or exceed the best human performance measured in Shannon's experiments over seven decades ago. This achievement suggests that these models have learned to capture most of the statistical regularities that human readers exploit when predicting text, a remarkable feat considering that the models learn entirely from raw text without explicit linguistic instruction. The convergence of machine and human performance on this fundamental task represents one of the great accomplishments of artificial intelligence research, though

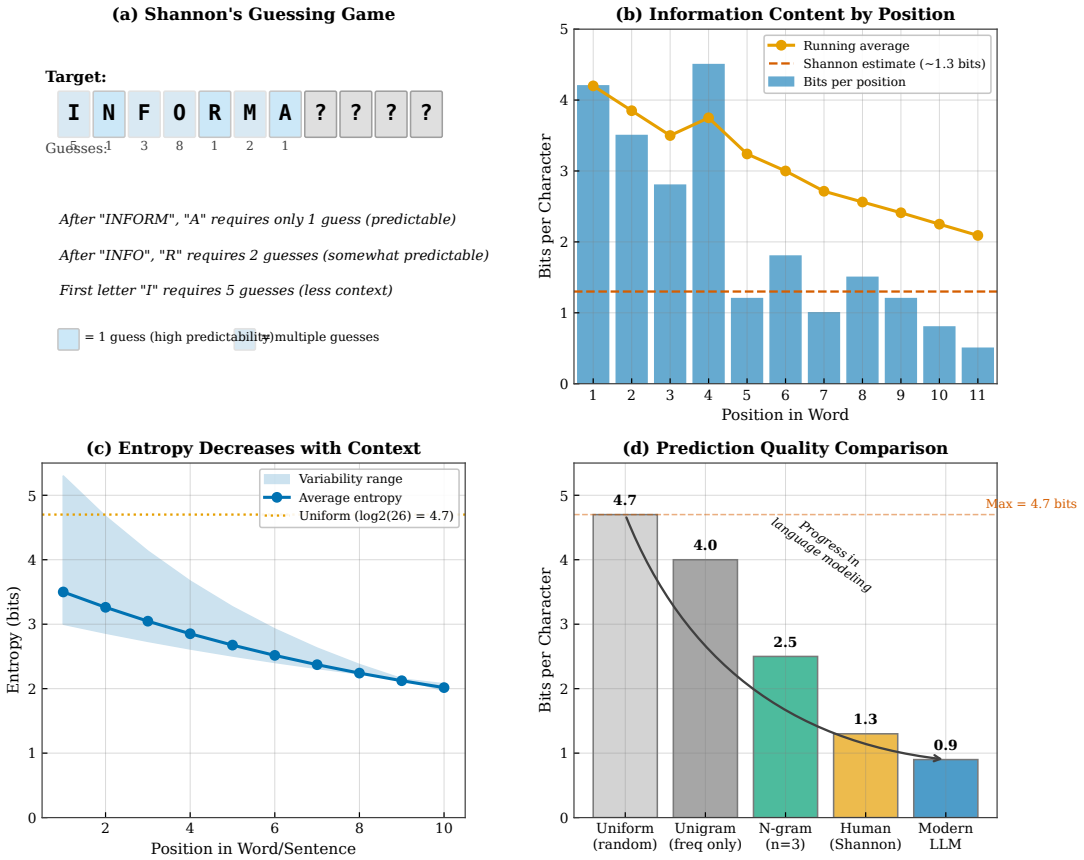


Figure 1.4: Shannon's guessing game experiment. Panel (a) shows the experimental setup where subjects guess letters sequentially. Panel (b) displays guess distributions for different contexts. Panel (c) illustrates how predictability varies with context. Panel (d) shows the relationship between guesses needed and entropy.

debates continue about whether this statistical competence reflects genuine language understanding or merely sophisticated pattern matching.

1.2.2 Statistical Methods (1970s–2000s)

The practical development of language models began with speech recognition research at IBM [Jelinek, 1990], where researchers discovered that adding a statistical language model to acoustic models dramatically improved transcription accuracy by guiding the recognizer toward word sequences that sounded similar but differed in linguistic plausibility. When acoustic evidence is ambiguous between "recognize speech" and "wreck a nice beach," the language model supplies the prior probability that makes the correct interpretation far more likely. This discovery launched decades of research into statistical language modeling, transforming it from a theoretical curiosity into an essential component of practical systems. **N-gram models**—which estimate $P(w_t \mid w_{t-n+1}, \ldots, w_{t-1})$ from simple word counts in a training corpus—became the dominant approach for decades due to their simplicity, computational efficiency, and surprisingly strong performance on many tasks despite their severe limitations. The key insight behind n-gram models is the Markov assumption: rather than conditioning on the entire preceding context which would be computationally intractable, we approximate by conditioning only on the previous $n-1$ words, making the estimation problem tractable while still capturing the local dependencies that account for much of language's predictability within short spans of text.

Several key developments during this era transformed language modeling from a theoretical curiosity into a practical technology. Good-Turing estimation, introduced by Good [1953], provided a principled method for handling rare events by using the frequency of frequencies to estimate probabilities for unseen n-grams. Building on this foundation, Kneser and Ney [1995] developed Kneser-Ney smoothing, which remains one of

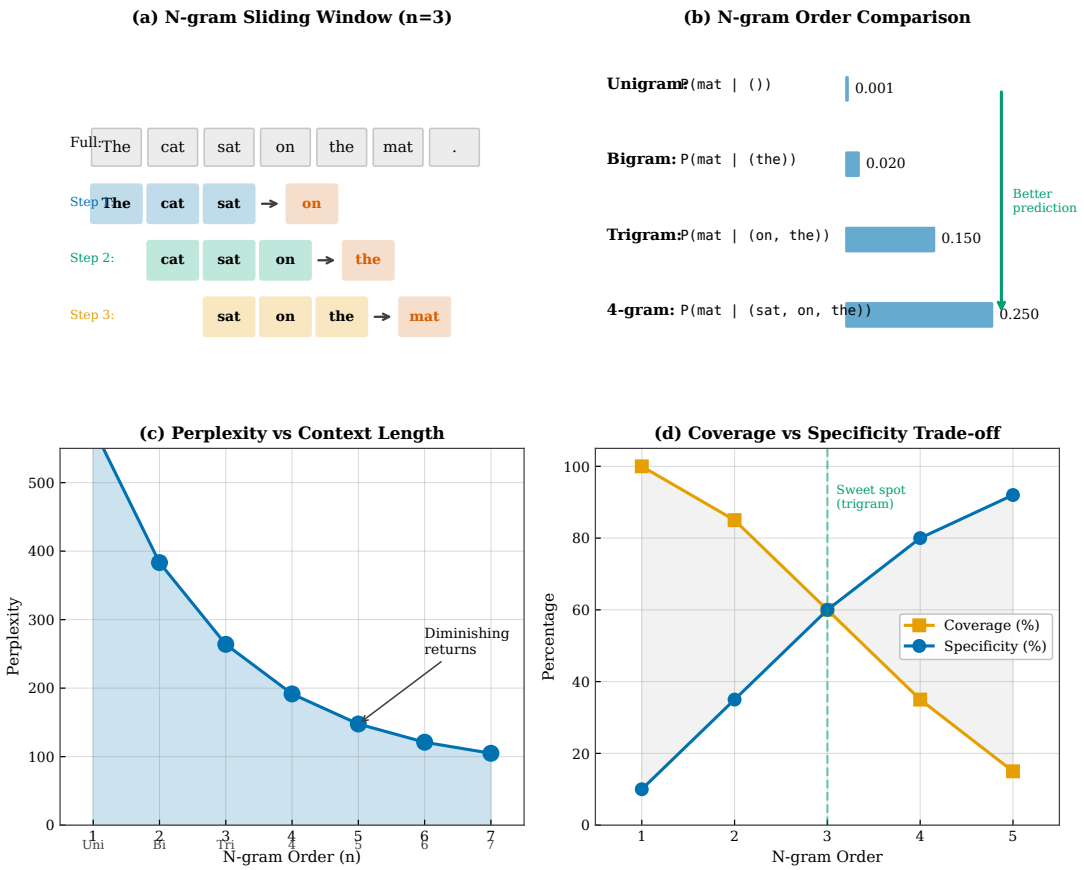


Figure 1.5: N-gram sliding window approach. Panel (a) shows unigram, bigram, and trigram windows. Panel (b) illustrates context limitation. Panel (c) displays the trade-off between context size and sparsity. Panel (d) demonstrates how n-grams miss long-range dependencies.

the most effective smoothing techniques for n-gram models by combining absolute discounting with a sophisticated backoff mechanism that considers how many distinct contexts a word appears in. The computational infrastructure of this era also advanced dramatically, enabling researchers to train language models on billions of words—a scale that would have been unimaginable in Shannon's time. These larger training corpora led to significant improvements in perplexity, demonstrating the fundamental importance of data scale for language modeling. Perhaps most importantly, this period saw the successful integration of language models with speech recognition systems at IBM and AT&T Bell Labs, and later with statistical machine translation systems. These applications demonstrated that language models had practical value beyond theoretical interest, setting the stage for the explosion of NLP applications that followed. Figure 1.5 illustrates how n-gram models use fixed-size context windows, trading off context length against data sparsity.

1.2.3 The Neural Revolution (2003–2017)

Bengio et al. [2003] introduced the first neural language model, representing words as dense vectors (*embeddings*) and using a feedforward network to predict the next word. This foundational work demonstrated that neural networks could learn meaningful representations of words—vectors in which semantically similar words cluster together—while simultaneously learning to predict text. The ideas from this paper sparked a revolution in natural language processing that unfolded over the following decade and a half, fundamentally changing how researchers approached the problem of modeling language. Mikolov et al. [2013] dramatically scaled these ideas with Word2Vec, demonstrating that word embeddings trained on massive corpora exhibited remarkable algebraic properties, such as the famous "king - man + woman = queen" analogy that captured semantic relationships through vector arithmetic. For sequence modeling, recurrent neural networks and particularly Long

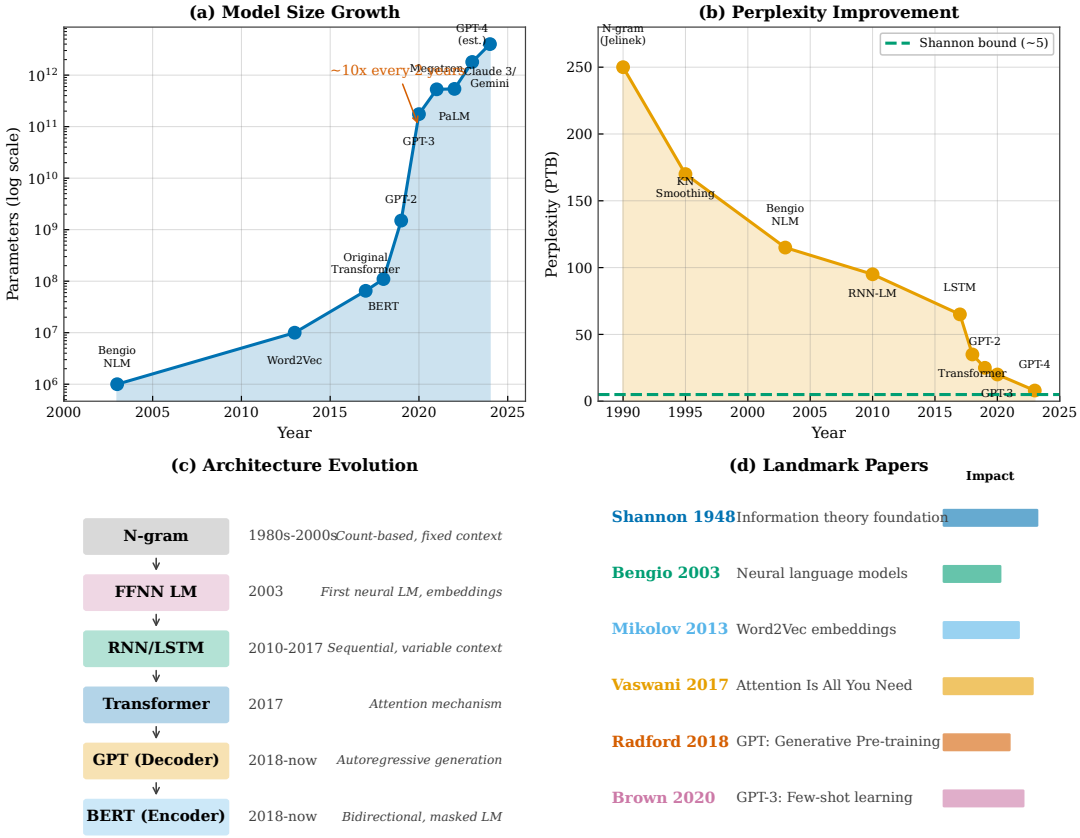


Figure 1.6: Evolution of language modeling architectures. Panel (a) shows the progression from count-based to neural approaches. Panel (b) displays architectural improvements over time. Panel (c) illustrates the growth in model capacity. Panel (d) demonstrates capability emergence across paradigms.

Short-Term Memory networks [Hochreiter and Schmidhuber, 1997] provided a way to process variable-length contexts by maintaining a hidden state that accumulates information over time, overcoming the fixed-context limitation of n-gram models and enabling models to capture dependencies across arbitrary distances. The final breakthrough came with the attention mechanism [Bahdanau et al., 2015], which allowed models to dynamically focus on relevant parts of the input, and its culmination in the Transformer architecture [Vaswani et al., 2017], which replaced recurrence entirely with self-attention and enabled unprecedented parallelization during training.

1.2.4 The Era of Large Language Models (2018–Present)

The Transformer architecture [Vaswani et al., 2017] enabled training on unprecedented scales by replacing sequential recurrent computation with parallel self-attention operations, allowing researchers to leverage modern GPU hardware to train models with billions of parameters on datasets containing hundreds of billions of tokens. This architectural innovation was transformative: while RNNs processed sequences one token at a time, requiring $O(n)$ sequential operations for a sequence of length n, Transformers could process all positions simultaneously, dramatically accelerating training and enabling the scaling that would define the following era. GPT-2 [Radford et al., 2019] demonstrated coherent multi-paragraph text generation that could maintain topic consistency and stylistic coherence over hundreds of words, surprising researchers with its ability to generate plausible news articles and stories; GPT-3 [Brown et al., 2020] showed emergent few-shot learning capabilities where the model could perform new tasks from just a handful of examples provided in the prompt, without any gradient updates to the model parameters. Subsequent work introduced instruction tuning [Ouyang et al., 2022], which fine-tuned models to follow explicit instructions, and preference optimization [Rafailov et al.,

2023], which aligned model outputs with human preferences through direct learning from human feedback. By 2024–2025, models like GPT-4, Claude, LLaMA [Touvron et al., 2023], and reasoning models like DeepSeek-R1 [DeepSeek-AI, 2025] exhibit sophisticated reasoning, multi-step problem solving, and natural dialogue abilities that would have seemed like science fiction just a decade earlier.

Figure 1.6 traces this evolution, showing how each paradigm shift brought qualitative improvements in prediction capability and enabled entirely new applications of language technology.

1.3 Information Theory Foundations

Understanding language modeling requires key concepts from information theory [Shannon, 1948], the mathematical framework that Shannon developed to study communication systems and that has proven equally applicable to the study of natural language. These concepts provide both the theoretical foundation for evaluating language models and deep intuition about the task's inherent difficulty. Information theory answers questions such as: How do we measure uncertainty? How do we quantify how well a model predicts? What is the theoretical limit of prediction accuracy? The answers to these questions—entropy, cross-entropy, and perplexity—form the vocabulary that researchers use to discuss language model quality, and understanding them is essential for anyone working in this field. The mathematical elegance of information theory lies in its ability to unify seemingly disparate phenomena under a common framework: the same concepts that describe optimal data compression apply equally to evaluating language models, because both tasks fundamentally involve predicting sequences of symbols from a probabilistic source. This section develops the core information-theoretic concepts that will recur throughout this book, providing the mathematical tools needed to precisely measure and compare language model performance.

1.3.1 Entropy

Definition 1.2 (Entropy)**.** *The* entropy *of a discrete random variable X with distribution $p(x)$ is:*

$$H(X) = -\sum_{x} p(x) \log_2 p(x) \tag{1.4}$$

measured in bits.

Entropy measures uncertainty or unpredictability in a probability distribution, quantifying how much "surprise" we should expect on average when sampling from that distribution. For a fair coin, $H(X) = 1$ bit, meaning we need exactly one yes/no question to resolve the uncertainty—will it be heads or tails? For a biased coin with $P(\text{heads}) = 0.9$, $H(X) \approx 0.47$ bits—less uncertain, so less entropy, because we already have a strong expectation that heads will appear. In the extreme case of a completely deterministic outcome, entropy equals zero: there is no uncertainty to resolve. The unit of entropy, the bit, corresponds to the uncertainty of a single fair coin flip—this provides an intuitive anchor for understanding larger entropy values. A vocabulary of 50,000 words under a uniform distribution would have entropy of approximately 15.6 bits, representing the information needed to specify one word among 50,000 equally likely alternatives. For language modeling, entropy captures how uncertain we are about the next word given the context, and the goal of training is to reduce this uncertainty by learning which words are more or less likely in different contexts. A well-trained language model dramatically reduces this entropy by concentrating probability mass on a small set of plausible continuations.

Theorem 1.1 (Maximum Entropy)**.** *For a discrete random variable X taking n possible values, entropy is maximized when X is uniformly distributed:*

$$H(X) \leq \log_2 n \tag{1.5}$$

with equality if and only if $p(x) = 1/n$ for all x.

Figure 1.7 visualizes entropy as a surface over probability distributions, showing how it peaks at the uniform distribution and decreases as the distribution becomes more concentrated.

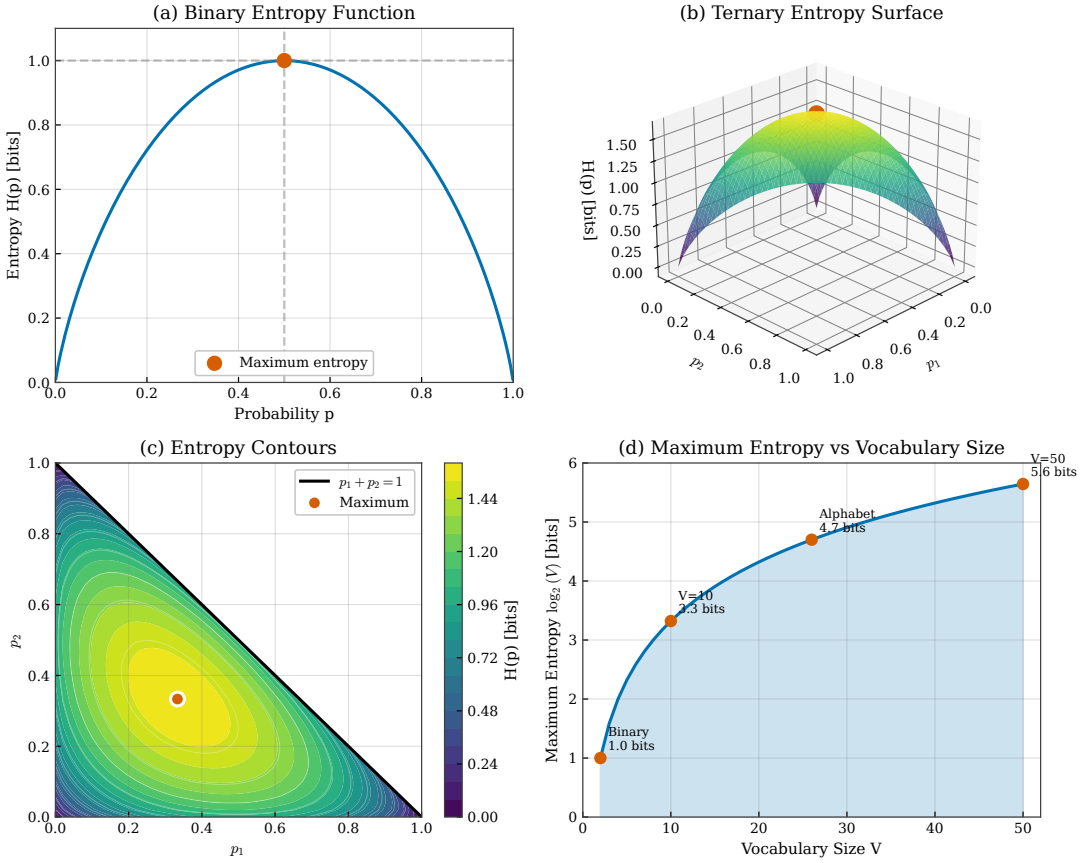


Figure 1.7: Entropy as a function of probability distribution. Panel (a) shows the binary entropy function $H(p) = -p\log p - (1-p)\log(1-p)$. Panel (b) displays a 3D entropy surface for three-outcome distributions. Panel (c) illustrates maximum entropy at uniform distribution. Panel (d) shows entropy contours on the probability simplex.

1.3.2 Cross-Entropy and Perplexity

When we have a model distribution q approximating the true distribution p:

Definition 1.3 (Cross-Entropy)**.**

$$H(p,q) = -\sum_x p(x)\log_2 q(x) \tag{1.6}$$

The cross-entropy measures how well q predicts samples from p. Figure 1.8 visualizes cross-entropy as a loss function, showing how it heavily penalizes confident wrong predictions. Cross-entropy equals the entropy $H(p)$ plus the KL divergence $D_{\mathrm{KL}}(p\,\|\,q)$:

$$H(p,q) = H(p) + D_{\mathrm{KL}}(p\,\|\,q) \tag{1.7}$$

so $H(p,q) \geq H(p)$ with equality when $q = p$.

For language modeling, we evaluate by computing the cross-entropy on a test corpus:

$$H(\text{corpus},\text{model}) = -\frac{1}{T}\sum_{t=1}^{T}\log_2 P(w_t\,|\,w_1,\ldots,w_{t-1}) \tag{1.8}$$

Definition 1.4 (Perplexity)**.**

$$PPL = 2^{H(corpus,model)} \tag{1.9}$$

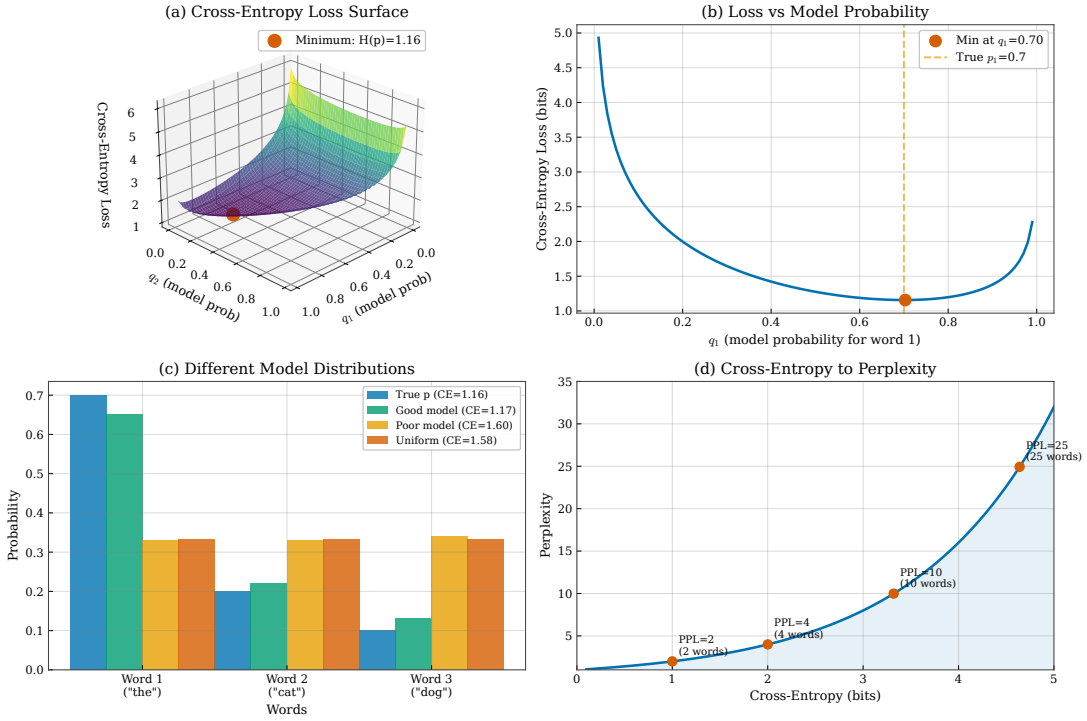


Figure 1.8: Cross-entropy as a loss function. Panel (a) shows the cross-entropy surface as a function of predicted probability. Panel (b) displays the gradient of cross-entropy loss. Panel (c) illustrates how cross-entropy penalizes confident wrong predictions. Panel (d) compares cross-entropy with other loss functions.

Perplexity can be interpreted as the effective vocabulary size over which the model spreads its probability mass: a perplexity of 100 means the model is "as confused" as if uniformly choosing among 100 equally likely words at each position, while a perplexity of 10 indicates concentration on roughly 10 plausible options. This interpretation makes perplexity intuitive: instead of thinking about bits and logarithms, we can think about how many words the model is effectively considering at each step. Lower perplexity indicates better prediction, with the theoretical minimum of 1 achieved only when the model assigns probability 1 to the correct word at every position—perfect prediction with zero uncertainty. State-of-the-art language models on benchmarks like WikiText-103 achieve perplexities below 10, meaning they effectively narrow down each prediction to about 10 plausible candidates on average, a remarkable feat given vocabularies containing tens of thousands of words. The progression from early n-gram models with perplexities around 100–200 to modern Transformers with perplexities below 10 represents a fundamental advance in our ability to capture the statistical structure of language, corresponding to an order-of-magnitude reduction in the effective uncertainty at each prediction step.

Example 1.2 (Perplexity Interpretation)**.** *Consider predicting the next word after "The capital of France is" as illustrated in Figure 1.10. A uniform model that assigns equal probability to all 50,000 words in its vocabulary would have a perplexity of exactly 50,000, reflecting maximum uncertainty—the model has no ability to distinguish likely continuations from unlikely ones. In contrast, a model that correctly recognizes this context and assigns probability 0.8 to "Paris" while distributing the remaining 0.2 probability across other words would achieve a perplexity of approximately 1.5, indicating that on average the model is nearly as certain as if it were choosing between just 1–2 equally likely words. A perfect model that assigns probability 1 to the correct word at every position would achieve perplexity 1, representing complete certainty. This example demonstrates why perplexity serves as such an intuitive metric: it directly measures how "confused" the model is, with lower values indicating greater predictive confidence. The logarithmic relationship between perplexity and probability means that improvements from 100 to 50 perplexity represent the same relative gain as improvements from 50 to 25.*

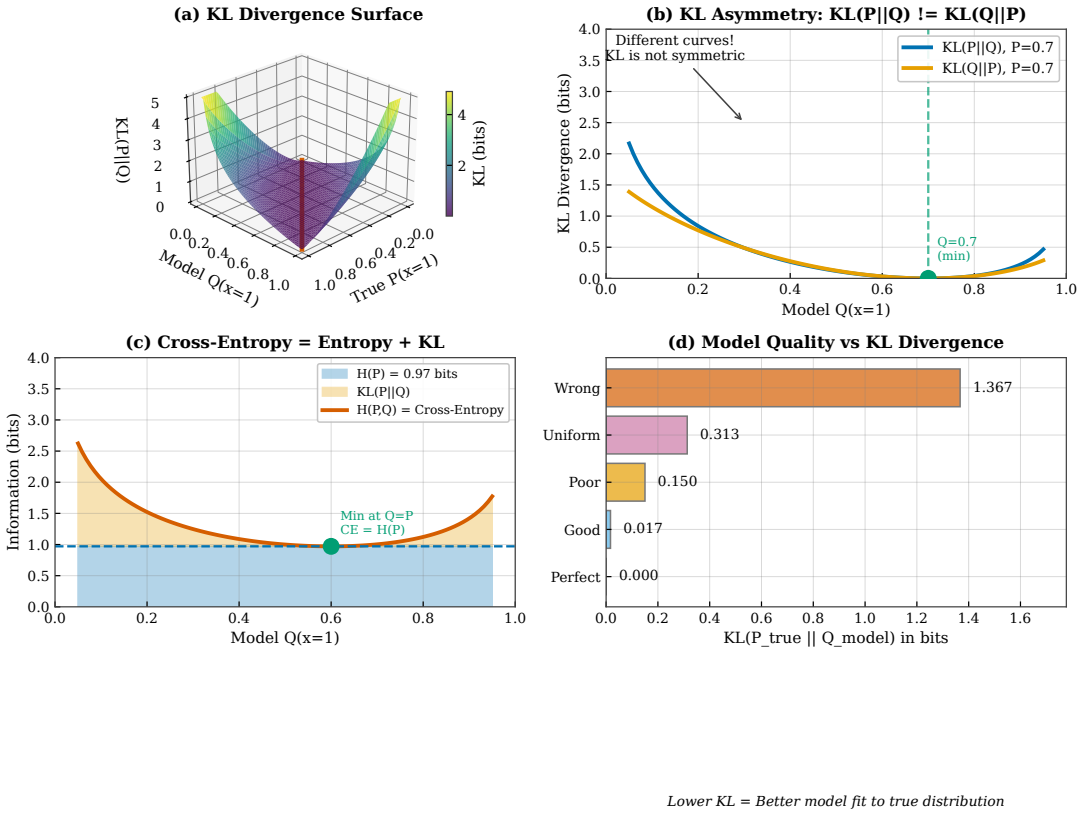


Figure 1.9: KL divergence visualization. Panel (a) shows asymmetry: $D_{\mathrm{KL}}(p \,\|\, q) \neq D_{\mathrm{KL}}(q \,\|\, p)$. Panel (b) displays a 3D surface of KL divergence. Panel (c) illustrates the relationship between cross-entropy and KL divergence. Panel (d) demonstrates mode-covering vs. mode-seeking behavior.

1.3.3 Bits Per Character

An alternative metric that normalizes for vocabulary differences is **bits per character (BPC)**:

$$\mathrm{BPC} = \frac{1}{C} \sum_{t=1}^{T} \log_2 \frac{1}{P(w_t \mid w_{<t})} \tag{1.10}$$

where C is the total number of characters in the corpus, effectively normalizing the information content by the length of the raw text rather than by the number of tokens. BPC is particularly useful for comparing models with different tokenization schemes: a character-level model predicting individual letters, a word-level model predicting whole words, and a subword model using byte-pair encoding would all have different perplexities that are not directly comparable, but their BPC values can be meaningfully compared because they measure the same underlying quantity—the information required to encode each character of the original text. This normalization is crucial in modern NLP research where tokenization strategies vary widely: GPT-2 and GPT-3 use byte-pair encoding with different vocabulary sizes, BERT uses WordPiece tokenization, and some models operate directly on characters or bytes. Without a common metric like BPC, comparing these models would require making arbitrary assumptions about how to convert between token-level and character-level measurements. BPC also connects directly to Shannon's entropy estimates from human experiments, enabling meaningful comparison between machine and human language prediction capabilities across seven decades of research.

Shannon's experiments suggest human-level English prediction corresponds to approximately 1.0–1.5 BPC. Figure 1.11 shows how modern language models have approached and, in some cases, surpassed this bound.

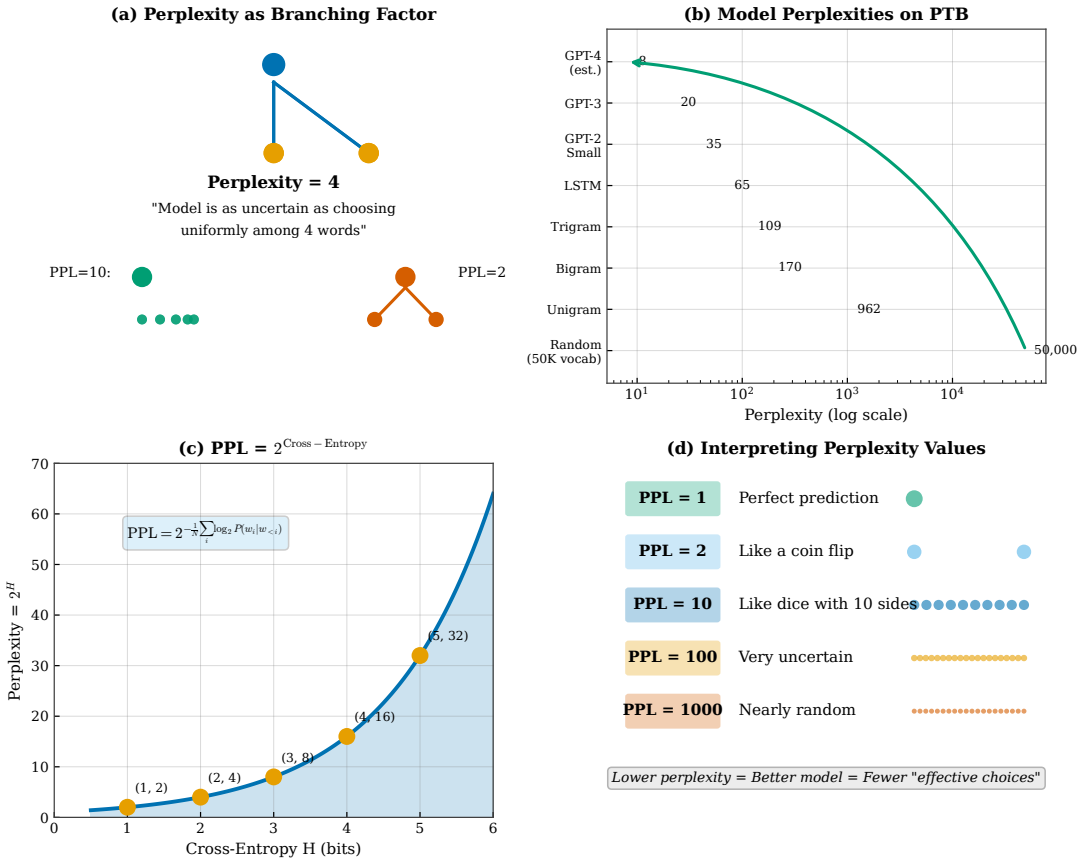


Figure 1.10: Interpreting perplexity. Panel (a) shows perplexity as effective vocabulary size. Panel (b) displays the relationship between perplexity and prediction confidence. Panel (c) compares perplexities of different models. Panel (d) illustrates how perplexity varies across text difficulty.

1.3.4 The Log Probability Space

Working with probabilities directly leads to numerical underflow for long sequences. We instead work in *log probability space*:

$$\log P(w_1, \ldots, w_T) = \sum_{t=1}^{T} \log P(w_t \mid w_1, \ldots, w_{t-1}) \tag{1.11}$$

This transformation, illustrated in Figure 1.12, converts products to sums, avoiding the exponential decay of probability values. The negative log probability $-\log P(w_t \mid w_{<t})$ is the *surprisal* of word w_t—the number of bits needed to encode it given the context.

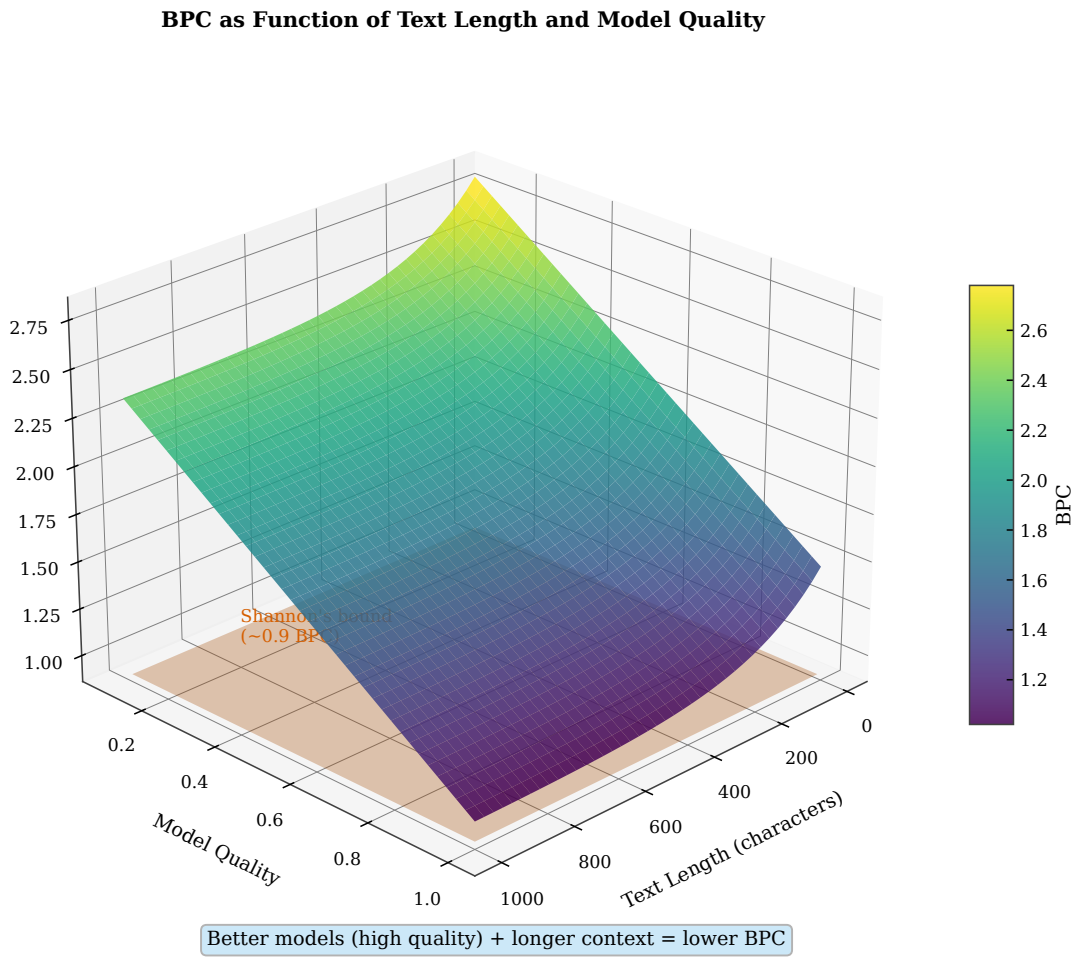


Figure 1.11: 3D surface showing bits per character (BPC) as a function of text length and model quality. Better models and longer contexts lead to lower BPC, approaching Shannon's theoretical bound of approximately 0.9 BPC for English.

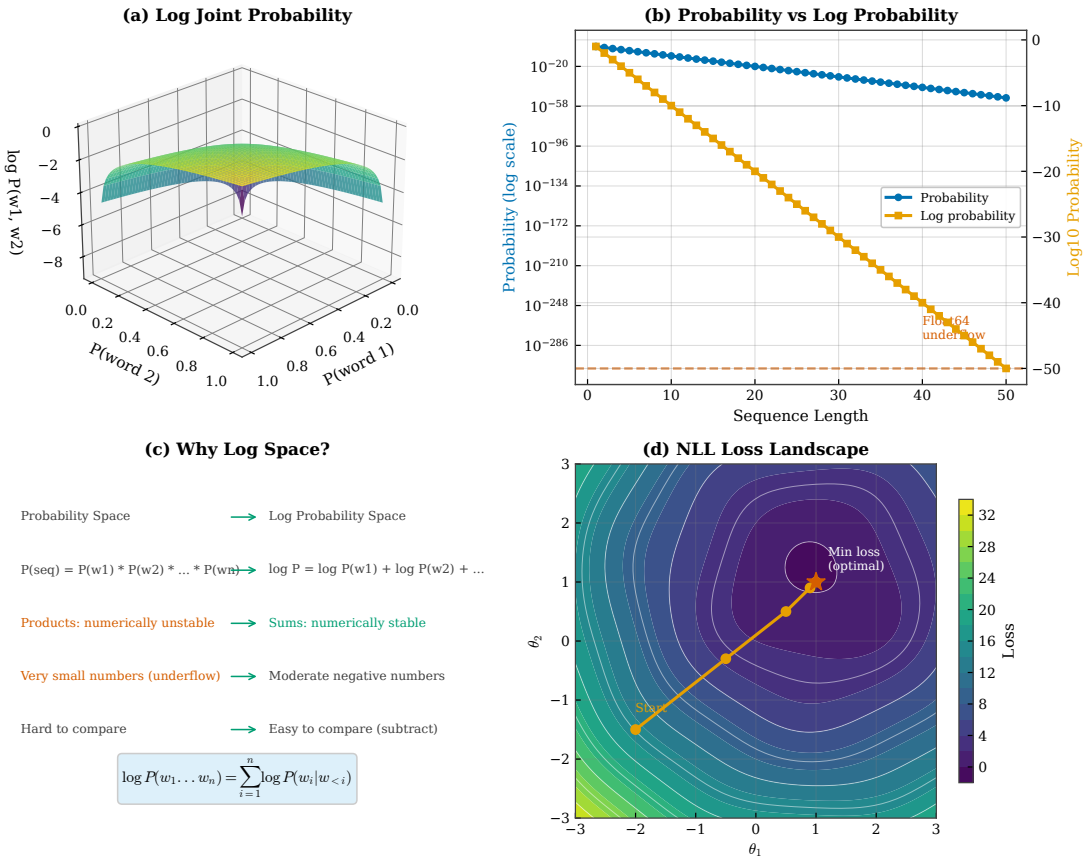


Figure 1.12: Working in log probability space. Panel (a) shows a 3D visualization of the log transformation. Panel (b) demonstrates why products become sums. Panel (c) illustrates numerical stability advantages. Panel (d) displays the log-sum-exp trick for computing normalizers.

1.4 The Probability Distribution Over Words

A language model must output a valid probability distribution over the vocabulary at each position, satisfying the fundamental axioms of probability theory: all values must be non-negative, and they must sum to exactly one across all possible next words. Understanding this constraint is essential for model design, as it determines how neural networks must transform their internal computations into valid probability outputs and influences architectural choices at every stage of the prediction pipeline. For a vocabulary of 50,000 words, the model must produce 50,000 non-negative numbers that sum to one—a high-dimensional constrained output that requires careful handling. This section explores the mathematical structure of probability distributions over discrete vocabularies, the softmax function that neural networks use to produce valid distributions, and the geometric interpretation of these constraints that helps build intuition for model behavior. The probability simplex provides the geometric setting for understanding these distributions, while the softmax function provides the computational mechanism for mapping unconstrained neural network outputs onto this constrained space. Together, these concepts illuminate the final stage of every neural language model: the projection from internal representations to probability distributions over words.

1.4.1 The Probability Simplex

Definition 1.5 (Probability Simplex)**.** *The* probability simplex Δ^{V-1} *is the set of all valid probability distributions over V outcomes:*

$$\Delta^{V-1} = \left\{ \mathbf{p} \in \mathbb{R}^V : p_i \geq 0, \sum_{i=1}^{V} p_i = 1 \right\} \tag{1.12}$$

Figure 1.13 visualizes the probability simplex for a three-word vocabulary. Every point on this triangular surface represents a valid probability distribution. The vertices represent deterministic predictions (probability 1 on one word), while the center represents the uniform distribution.

1.4.2 The Softmax Function

Neural language models produce *logits*—unconstrained real numbers for each vocabulary word that can be positive, negative, or zero—which must be converted to valid probabilities that are non-negative and sum to one. The term "logit" comes from the log-odds function in statistics, reflecting the interpretation of these values as unnormalized log-probabilities: higher logits correspond to words the model considers more likely, while lower logits correspond to less likely words, with the scale being completely arbitrary until normalization is applied. Raw logits reflect the model's internal scoring of each word's plausibility but do not satisfy probability axioms—they can be any real number, positive or negative, and they certainly do not sum to one. The **softmax function** accomplishes the essential transformation from unconstrained scores to valid probabilities, exponentiating each logit to ensure non-negativity and then normalizing by the sum to ensure the outputs sum to one. This two-step process—exponentiation followed by normalization—is computationally convenient because it allows the neural network to output any real numbers while guaranteeing valid probability distributions at the output, regardless of the input context or the learned model parameters:

$$\text{softmax}(\mathbf{z})_i = \frac{\exp(z_i)}{\sum_{j=1}^{V} \exp(z_j)} \tag{1.13}$$

The *temperature* parameter τ controls the sharpness of the distribution:

$$\text{softmax}(\mathbf{z}/\tau)_i = \frac{\exp(z_i/\tau)}{\sum_{j=1}^{V} \exp(z_j/\tau)} \tag{1.14}$$

As shown in Figure 1.14, the temperature parameter controls the entropy of the output distribution in a continuous manner. When temperature approaches zero, the softmax function converges to the argmax operation, placing all probability mass on the highest-scoring word and producing deterministic, greedy predictions that always select the single most likely token. At temperature equal to one, the standard softmax transformation

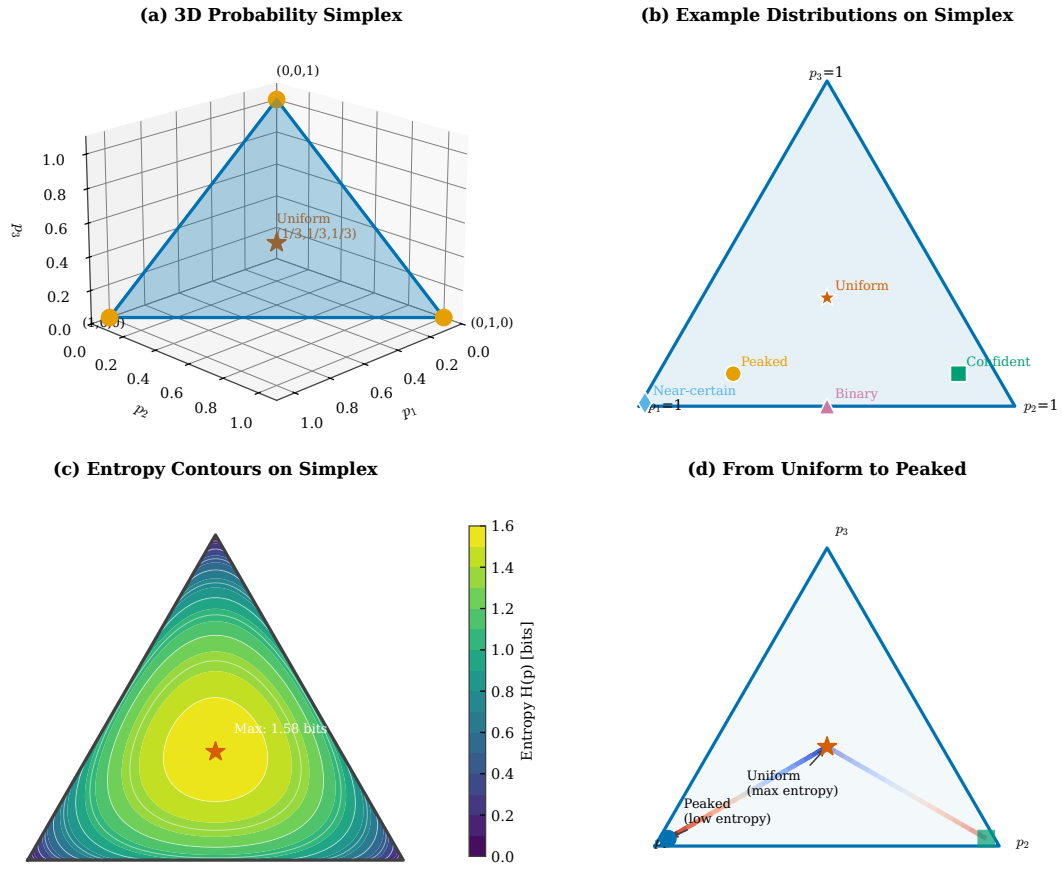


Figure 1.13: The probability simplex. Panel (a) shows the 3D simplex for three-word vocabulary. Panel (b) displays entropy contours on the simplex. Panel (c) illustrates how neural networks map to the simplex via softmax. Panel (d) demonstrates the effect of temperature on distribution sharpness.

applies, preserving the relative log-probability ratios from the logits exactly as the model learned them during training. As temperature increases beyond one, the distribution becomes progressively flatter, with the extreme case of temperature approaching infinity yielding a uniform distribution regardless of the input logits, where every word becomes equally likely and sampling becomes purely random. This temperature-controlled sharpening or smoothing proves essential for text generation, where practitioners balance the tension between coherent but repetitive low-temperature outputs and diverse but potentially incoherent high-temperature samples. The choice of temperature depends heavily on the application: code generation and formal writing benefit from low temperatures that favor precision, while creative writing and brainstorming benefit from higher temperatures that encourage novelty. Chapter 7 explores these decoding trade-offs in depth, examining how temperature interacts with other sampling strategies to produce high-quality generated text.

1.4.3 Conditional Probability Trees

Another perspective on language model outputs is the conditional probability tree, a powerful visualization that reveals the exponential structure of sequence generation. In this view, each node represents a partial sequence of words, and each edge represents the conditional probability of extending that sequence by one additional word. The root node represents the empty context (or a given prompt), and each path from root to leaf represents a complete generated sequence with total probability equal to the product of all edge weights along the path—this is simply the chain rule decomposition rendered as a tree. The branching factor at each node equals the vocabulary size, so even short sequences produce astronomically large trees: a vocabulary of 50,000 words and sequences of just 10 words yields $50,000^{10}$ possible paths, far more than could ever be enumerated explicitly.

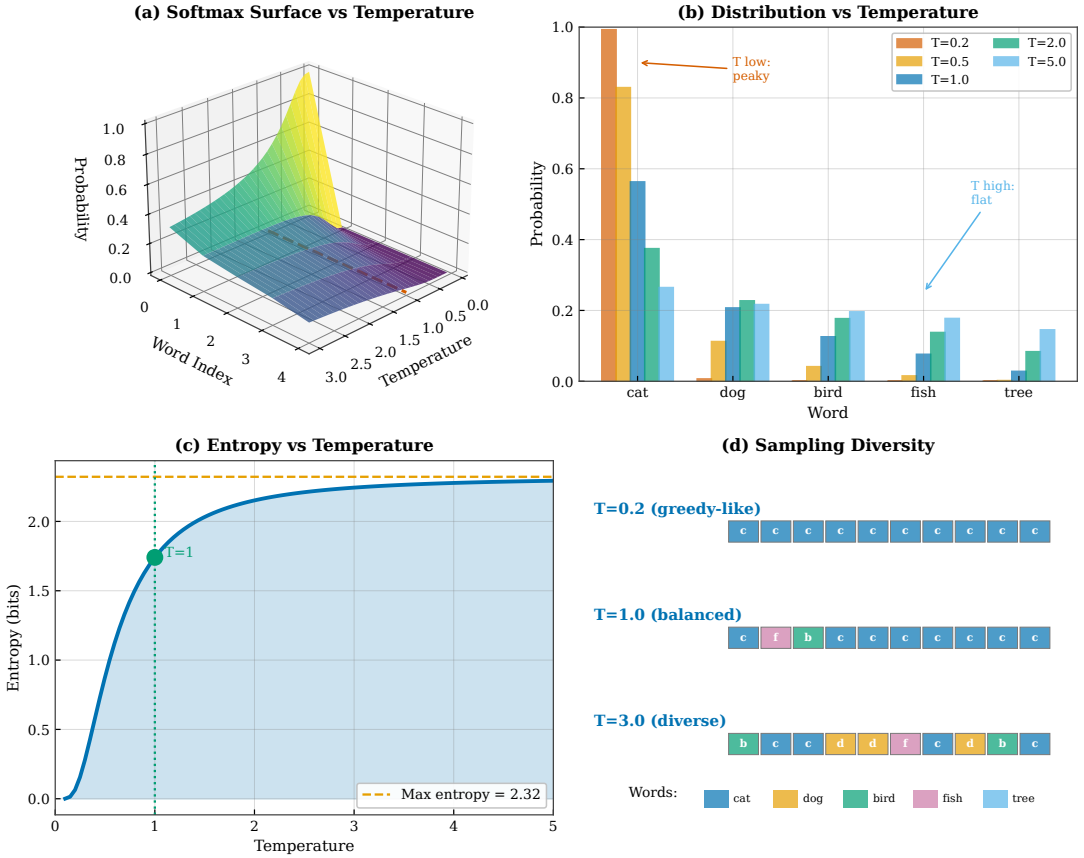


Figure 1.14: Softmax temperature effects. Panel (a) shows a 3D surface of softmax output as a function of logits and temperature. Panel (b) displays how temperature affects distribution entropy. Panel (c) illustrates low ($\tau < 1$) vs. high ($\tau > 1$) temperature behavior. Panel (d) demonstrates temperature's effect on sampling diversity.

This exponential explosion is both the challenge and the opportunity of language modeling: the challenge because exhaustive search is impossible, and the opportunity because a good language model concentrates probability mass on a tiny fraction of the possible sequences, making efficient decoding algorithms possible by pruning the low-probability branches.

Figure 1.15 shows how the conditional probability factorization creates a tree structure over all possible sequences. This view is essential for understanding decoding algorithms (Chapter 7).

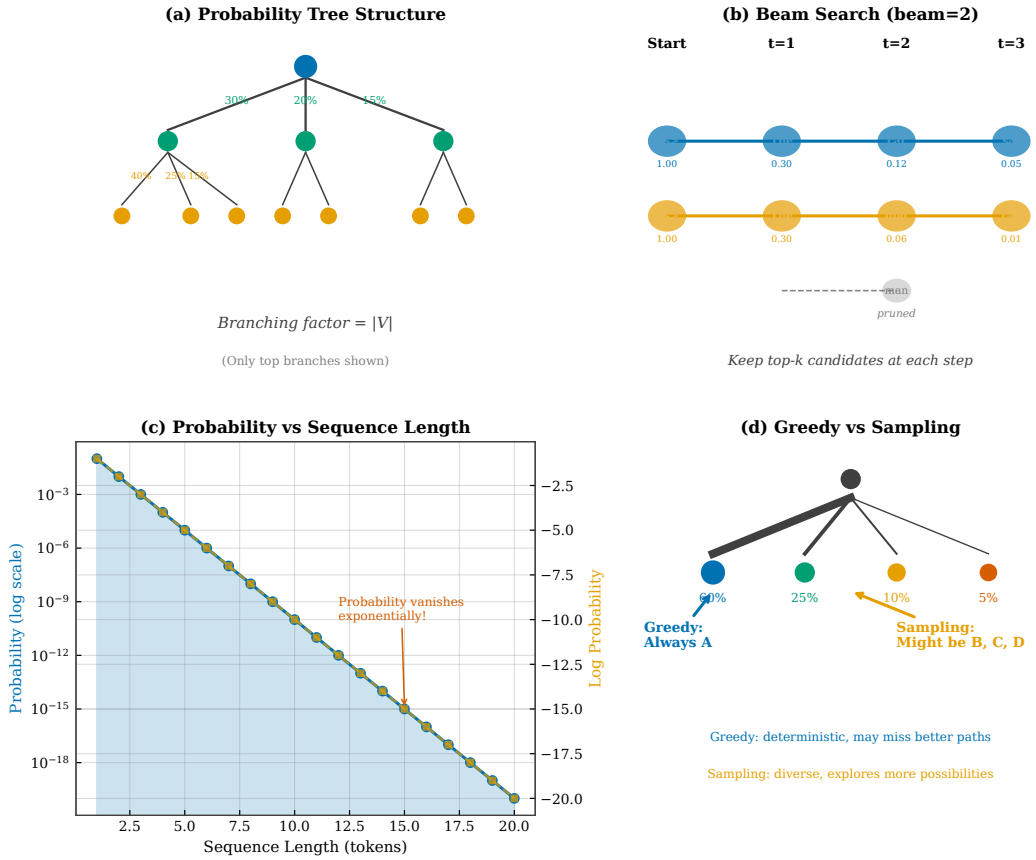


Figure 1.15: Conditional probability as a tree. Panel (a) shows the tree structure of sequence probabilities. Panel (b) illustrates how paths through the tree correspond to sentences. Panel (c) displays probability mass distribution across branches. Panel (d) demonstrates beam search traversal of the tree.

1.5 Linguistic Foundations

Language exhibits statistical regularities at multiple scales—from individual character sequences to discourse-level patterns spanning paragraphs—that language models must capture to make accurate predictions. Understanding these regularities helps us design better models by informing our choice of architecture, training data, and evaluation metrics, and it helps us interpret model behavior by revealing which linguistic phenomena a model has successfully learned versus which remain challenging. The statistical structure of language is remarkably consistent across different corpora and even across different languages, suggesting that some of these regularities reflect deep properties of human communication rather than arbitrary conventions of particular language communities. This section examines some of the fundamental statistical properties of natural language, from the heavy-tailed word frequency distribution described by Zipf's law to the hierarchical structure of linguistic knowledge from phonology through pragmatics. These linguistic foundations inform both the design of language models and the interpretation of their successes and failures: a model that fails to capture Zipf's law will waste capacity on rare words, while a model that ignores syntactic structure will make predictions that violate basic grammaticality constraints that any native speaker would recognize immediately.

1.5.1 Zipf's Law

Theorem 1.2 (Zipf's Law)**.** *The frequency of a word is approximately inversely proportional to its rank:*

$$f(r) \propto \frac{1}{r^{\alpha}} \tag{1.15}$$

where r is the rank and $\alpha \approx 1$ for natural language [Zipf, 1949].

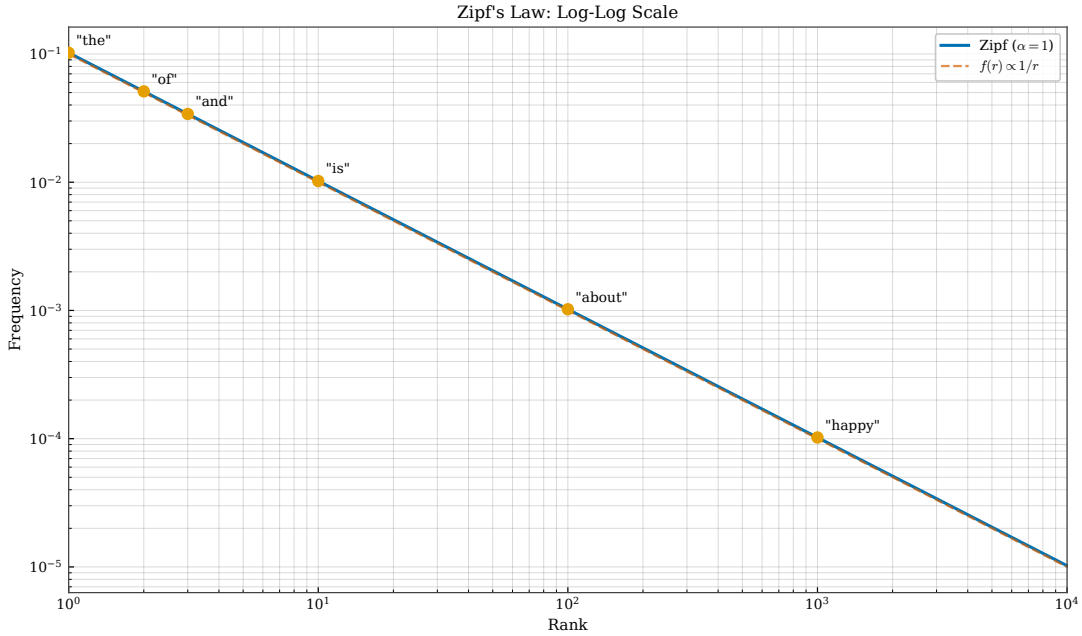


Figure 1.16: Zipf's law in natural language. Panel (a) shows the power-law relationship between rank and frequency on a log-log plot. Panel (b) displays the cumulative distribution. Panel (c) compares Zipf distributions across languages. Panel (d) illustrates implications for vocabulary coverage.

This power-law distribution, visualized in Figure 1.16, has profound implications for language modeling. The most striking consequence is the extreme concentration of probability mass: a small number of high-frequency words such as "the," "of," "and," and "to" account for a disproportionate fraction of all tokens in any corpus. In English text, the top 100 words typically cover more than 50 percent of all token occurrences, while the top 1,000 words cover approximately 80 percent. This concentration means that a language model encounters the same common words repeatedly during training, providing abundant data for learning their distributions. However, the flip side of this concentration is the "long tail" phenomenon: the vast majority of vocabulary items are rare, appearing only a handful of times even in billion-word corpora. A model trained on such data will inevitably encounter word combinations during evaluation that never appeared during training. This sparsity problem motivates the smoothing techniques discussed in Chapter 2, which redistribute probability mass from observed to unobserved events, as well as the subword tokenization methods of Chapter 3, which break rare words into more frequent constituent units.

1.5.2 The Vocabulary Problem

English contains hundreds of thousands of words in current use, plus proper names for people, places, and organizations, technical terminology from every field of human knowledge, and neologisms that emerge continuously as culture and technology evolve. The Oxford English Dictionary catalogs over 170,000 words in current use, and that figure excludes most technical jargon and proper nouns, which together easily push the effective vocabulary into the millions for any truly comprehensive language model. Any fixed vocabulary that a language model uses will inevitably encounter **out-of-vocabulary (OOV)** words during deployment—words that never appeared in the training data and thus have no learned representation. This vocabulary problem is fundamental to language modeling and has motivated significant research into solutions ranging from simple smoothing techniques to sophisticated subword tokenization methods. The challenge is particularly acute for languages with rich morphology, where a single root can generate dozens or hundreds of inflected forms, and for technical domains where specialized terminology proliferates. Even for English, new words enter the language constantly—"selfie," "cryptocurrency," and "COVID" were unknown just years before becoming ubiquitous—making the vocabulary problem an ongoing challenge rather than a one-time engineering decision.

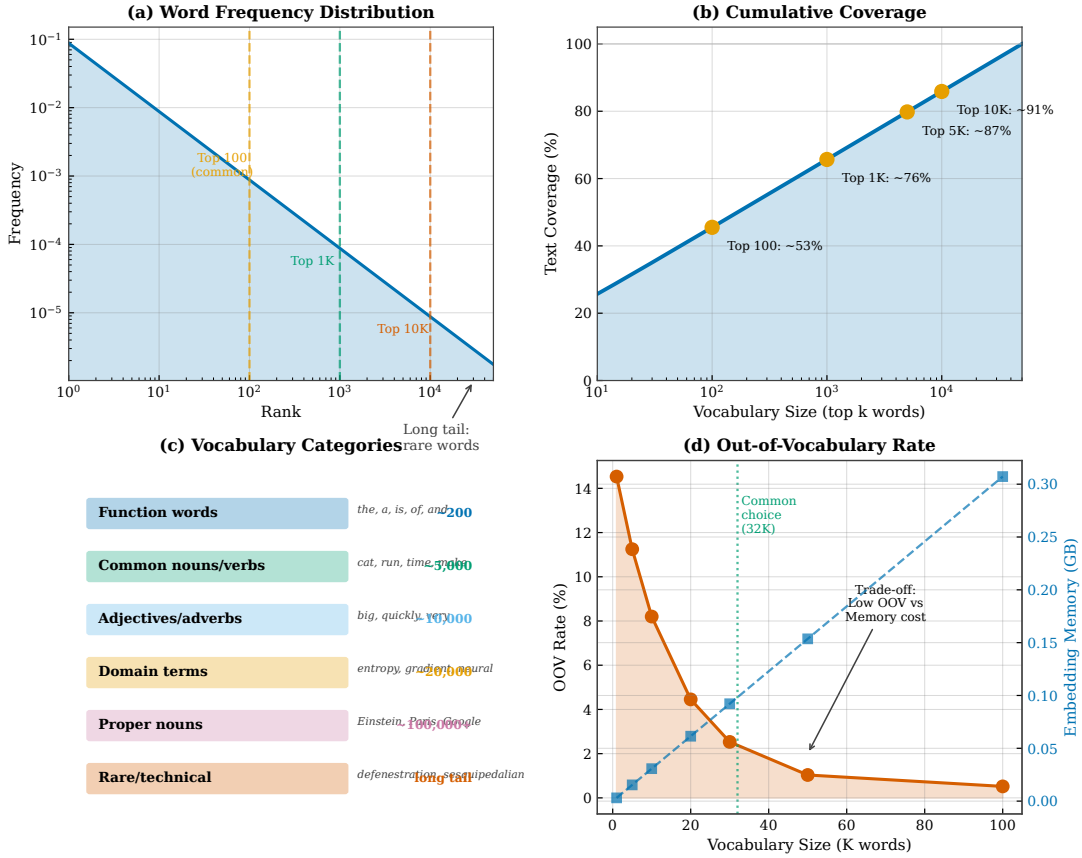


Figure 1.17: Vocabulary structure in language models. Panel (a) shows frequency-based vocabulary ordering. Panel (b) displays the relationship between vocabulary size and OOV rate. Panel (c) illustrates subword decomposition strategies. Panel (d) compares closed vs. open vocabulary approaches.

Figure 1.17 illustrates the vocabulary problem and the various solutions that have been developed to address it. The classical approach, detailed in Chapter 2, employs smoothing techniques that reserve a portion of probability mass for unseen words, typically by mapping all out-of-vocabulary items to a special unknown token and estimating its probability from held-out data. While simple and effective for moderate OOV rates, this approach loses all information about the specific unknown word encountered. A more sophisticated solution, which has become standard in modern systems and receives thorough treatment in Chapter 3, is subword tokenization. Methods such as Byte Pair Encoding and WordPiece decompose words into frequently occurring character sequences, so that even a never-before-seen word like "cryptocurrency" can be represented as a sequence of known subunits such as "crypto" and "currency." This approach effectively creates an open vocabulary: any string of characters can be tokenized, eliminating the OOV problem entirely while retaining morphological information that helps the model generalize to novel words. Neural language models have further embraced open-vocabulary approaches, sometimes operating directly at the character or byte level to achieve complete coverage of any possible input.

1.5.3 Language Structure and Prediction

Natural language has structure at multiple levels, each affecting prediction difficulty differently and requiring different types of knowledge to exploit effectively. These levels range from the subword phonological patterns that govern which letter sequences are pronounceable, through morphological rules for word formation, syntactic constraints on phrase and sentence structure, semantic requirements for meaningful interpretation, and pragmatic conventions for coherent discourse. A complete language model must capture patterns at all these levels, and the history of language modeling can be viewed as a progression toward architectures capable of

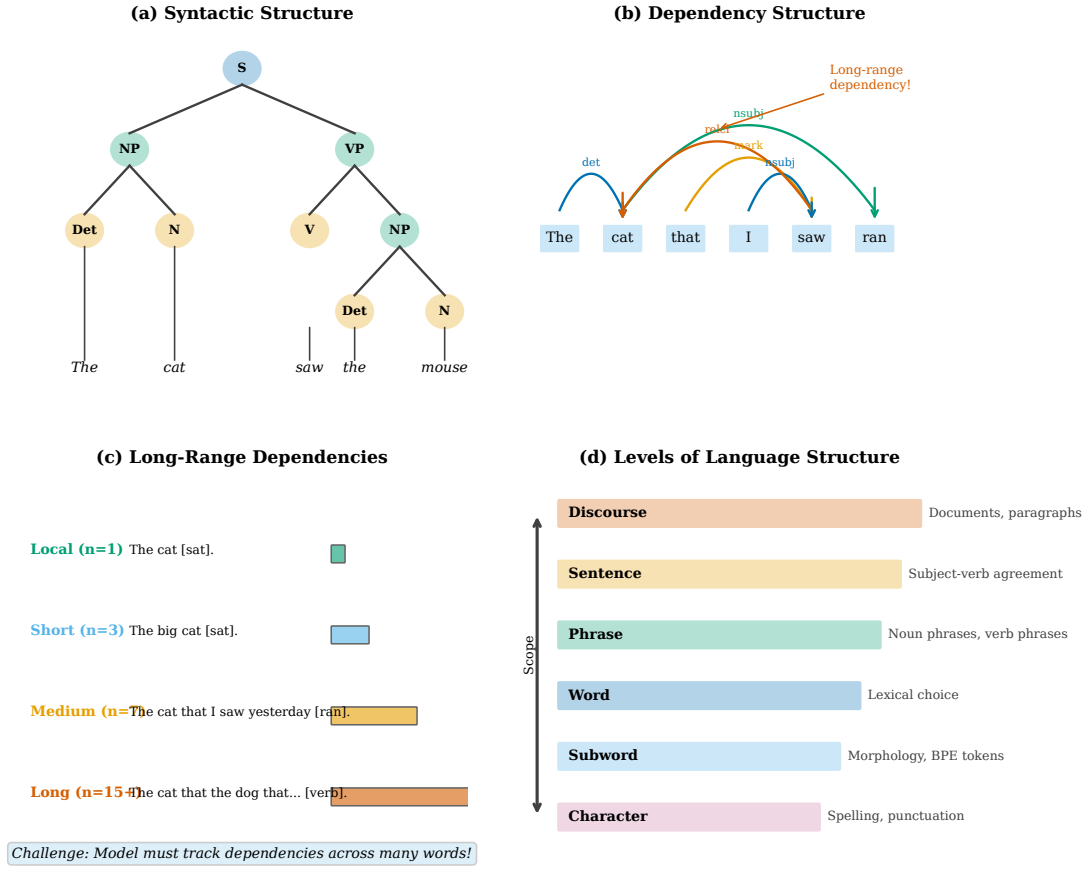


Figure 1.18: Language structure and prediction. Panel (a) shows phonological constraints on word sequences. Panel (b) displays morphological patterns. Panel (c) illustrates syntactic structure. Panel (d) demonstrates semantic coherence requirements.

learning increasingly sophisticated linguistic structure. Early n-gram models primarily captured local phonological and syntactic patterns within their limited context windows, while recurrent neural networks extended the reach to capture some longer-range syntactic dependencies and semantic coherence. Modern Transformer-based models have demonstrated remarkable ability to capture all of these linguistic levels to varying degrees, including pragmatic phenomena like tracking entities across long documents and maintaining consistent narrative voice throughout extended generations. The question of how well neural language models truly understand each of these linguistic levels—versus merely mimicking surface patterns without genuine comprehension—remains an active area of research and ongoing scholarly debate in the computational linguistics community.

These structural levels form a hierarchy of constraints that language models must learn to capture, with each level operating somewhat independently while also interacting with the others. At the phonological level, the sound patterns of a language constrain which letter sequences are plausible: English permits “str” at the beginning of words but not “tsr,” and these phonotactic constraints help predict spelling even for novel words that the model has never encountered during training. Morphological structure governs word formation through prefixes, suffixes, and root combinations, so knowing that “un-” typically attaches to adjectives helps predict that “unhappy” is more likely than “unsleep.” Syntactic structure imposes grammatical constraints on word order and phrase structure, determining that determiners precede nouns in English and that subjects typically precede verbs in declarative sentences. Semantic constraints require that word sequences make sense: “colorless green ideas sleep furiously” is syntactically valid but semantically anomalous, and language models must learn to penalize such combinations that violate real-world plausibility. Finally, pragmatic structure encompasses discourse-level phenomena including pronoun resolution, topic continuity, and speaker intent, requiring models to track entities and relationships across sentence boundaries. Figure 1.18 illustrates how each of these

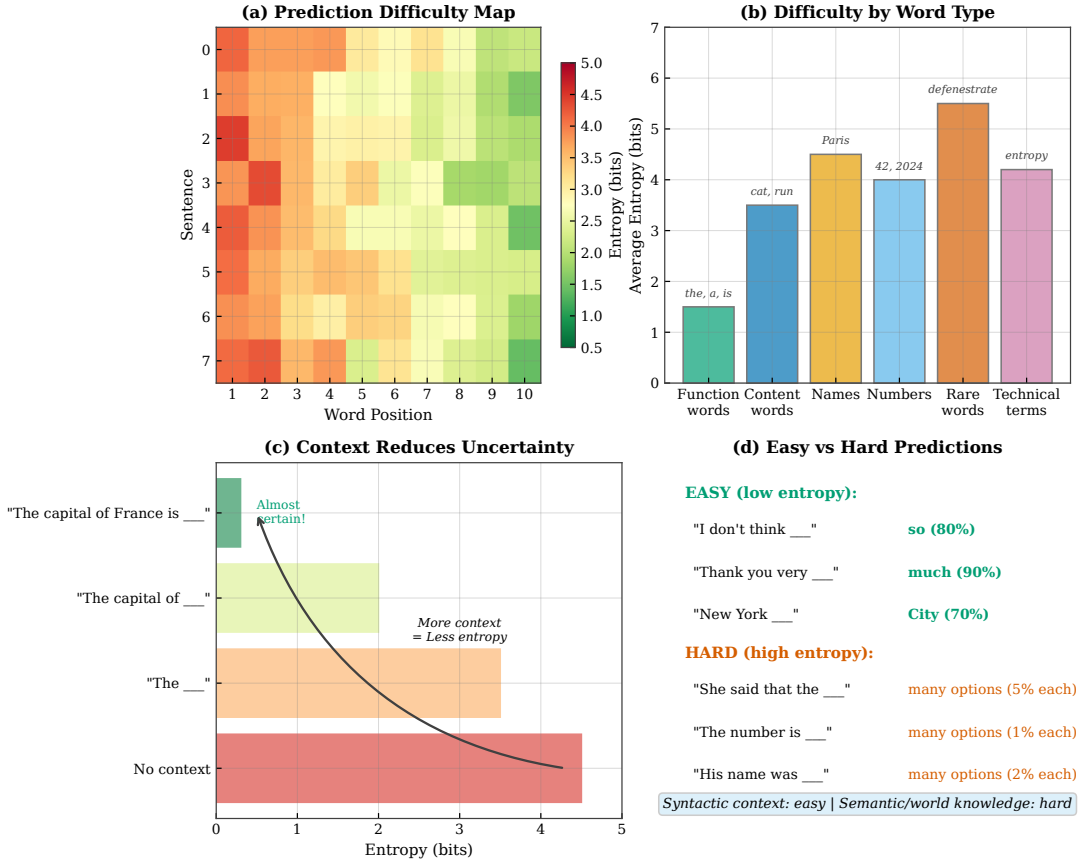


Figure 1.19: Variation in prediction difficulty. Panel (a) shows surprisal distribution across word positions. Panel (b) displays how context length affects difficulty. Panel (c) illustrates easy vs. hard prediction cases. Panel (d) demonstrates the relationship between difficulty and information content.

levels contributes different constraints on prediction.

1.6 Why Some Words Are Harder to Predict

Prediction difficulty varies dramatically across positions in text, ranging from nearly deterministic cases where only one word is plausible to highly ambiguous positions where dozens of words might reasonably follow. Understanding this variation is crucial for model evaluation and development, as it reveals both the strengths and limitations of different modeling approaches and highlights the linguistic phenomena that remain challenging even for state-of-the-art systems. Some predictions are trivially easy: after the phrase “the United States of,” the word “America” follows with near certainty, requiring only basic knowledge of common proper noun phrases. Other predictions approach randomness: the first word of a news article could be almost anything, from “Scientists” to “The” to “Breaking,” depending on topic and style. Between these extremes lies a rich spectrum of prediction difficulty that depends on syntactic constraints, semantic coherence, world knowledge, and discourse context. A key insight from Shannon’s information theory is that this variation in predictability directly corresponds to variation in information content: easy-to-predict words carry little information because they tell us nothing unexpected, while hard-to-predict words carry substantial information precisely because they surprise us. This relationship between predictability and information makes the study of prediction difficulty central to understanding both language models and language itself.

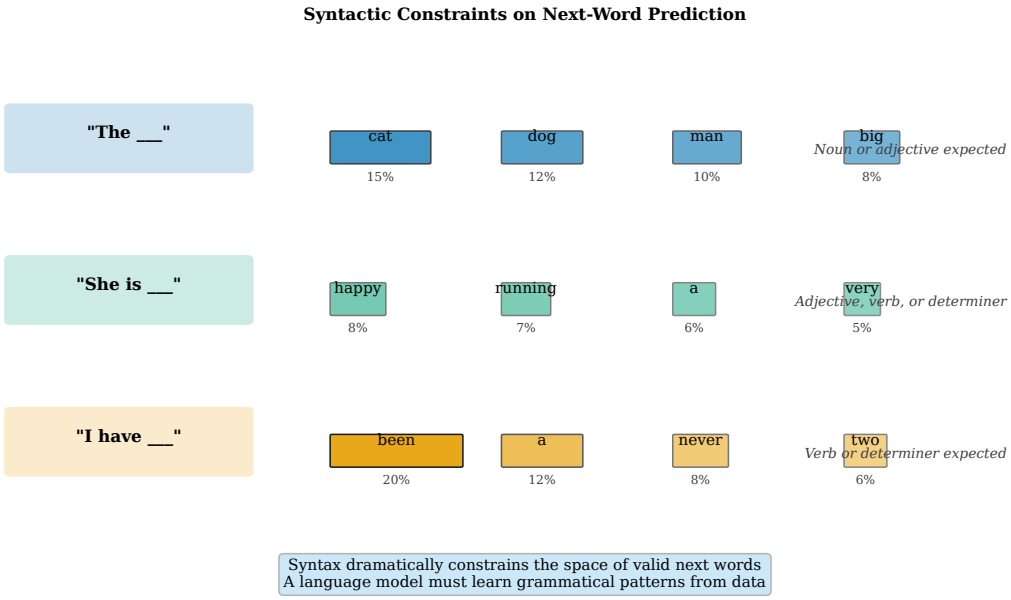


Figure 1.20: Syntactic constraints on next-word prediction. Grammar imposes strict ordering requirements that dramatically reduce the space of plausible continuations at each position.

1.6.1 Context Length and Information

Words at the beginning of a sentence are harder to predict than words later in the sentence, because less context is available to constrain the possibilities—the very first word of a document could be almost anything, while the tenth word is heavily constrained by the nine words preceding it. This relationship between context length and prediction difficulty follows directly from information theory: longer contexts reduce conditional entropy by providing more evidence to discriminate between possible continuations. Similarly, function words (“the,” “of,” “is”) are generally more predictable than content words (“quantum,” “fascinating,” “Constantinople”), because function words serve grammatical roles that are strongly determined by syntactic structure, while content words carry the specific semantic payload that varies widely depending on topic and domain. This distinction between predictable function words and unpredictable content words has important implications for evaluation: most of a language model’s perplexity comes from predicting content words, even though function words account for a larger fraction of tokens in running text. A model that perfectly predicts all function words but struggles with content words would still have high perplexity, because the content words carry the bulk of the information and thus dominate the cross-entropy calculation.

Figure 1.19 shows how surprisal varies across a sentence, revealing which positions carry more information.

1.6.2 Prediction Examples

To make these abstract notions of prediction difficulty concrete, we examine specific examples that span the full range from highly predictable to genuinely uncertain. Figure 1.20 provides illustrative cases organized by difficulty level, demonstrating how different types of linguistic knowledge contribute to prediction accuracy. The highly predictable cases typically involve fixed expressions, collocations, or contexts where syntax and semantics combine to leave only one plausible continuation—these are the positions where language models achieve their lowest perplexity and where even simple n-gram models perform reasonably well. The moderately predictable cases require more sophisticated reasoning about semantic coherence and world knowledge, often admitting several plausible continuations that a good model should assign comparable probabilities. The most difficult cases involve genuine ambiguity where multiple completions are equally valid, positions at the start of new topics where context provides little constraint, or situations requiring specialized domain knowledge that may not appear in general training corpora. Understanding these difficulty levels helps us interpret model

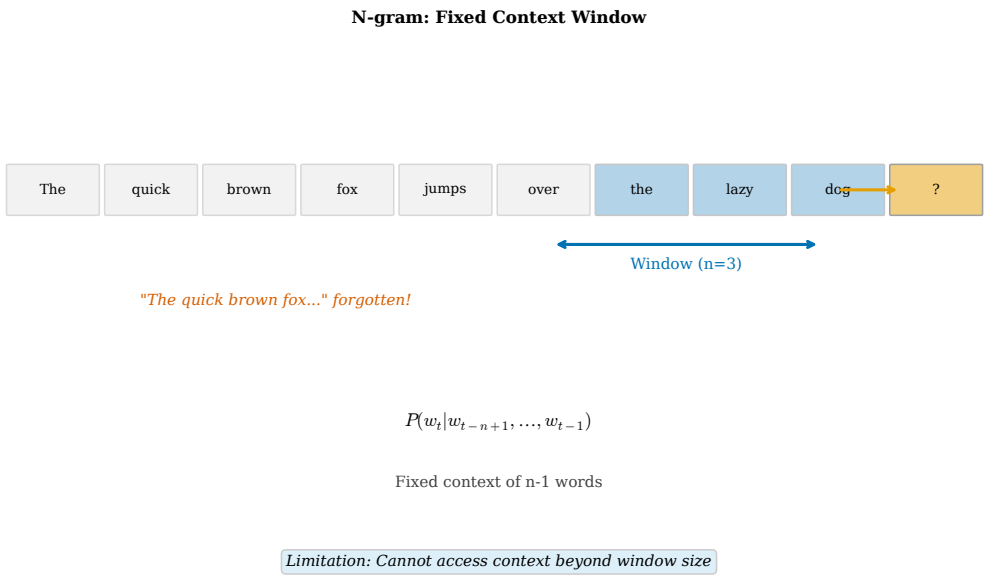


Figure 1.21: N-gram context window architecture. N-gram models use a fixed window of the previous $n-1$ words, making them fast and simple but unable to capture dependencies beyond that window.

performance: a model should not be penalized heavily for failing to predict the specific word chosen in a genuinely ambiguous context, but should be penalized for missing obvious predictions that human readers would find trivially predictable.

Example 1.3 (Easy Prediction)**.** *"She closed her _____" → "eyes" (high probability)*

Example 1.4 (Medium Prediction)**.** *"The meeting was scheduled for _____" → "Monday"/"Tuesday"/"next week"*

Example 1.5 (Hard Prediction)**.** *"The researcher's findings about _____" → Requires domain knowledge*

1.6.3 Context Window Comparison

The amount and type of context a model can access fundamentally determines its prediction capabilities, and different model architectures handle context in fundamentally different ways that reflect both computational constraints and inductive biases about which patterns matter for prediction. N-gram models use a fixed window of the previous $n-1$ words, making them fast and simple but unable to capture dependencies that span beyond that window—the choice of context in "The professor who taught the course that the students loved was" cannot help predict the final verb form because it lies outside any practical n-gram window. Recurrent neural networks theoretically access unbounded history through their hidden state, but in practice suffer from vanishing gradients that make distant context difficult to use effectively, despite architectural innovations like LSTM gates designed to ameliorate this problem. Transformer models use self-attention to directly access any position within a fixed context window, enabling effective use of long-range context up to the window limit but requiring computational resources that scale quadratically with sequence length. Understanding these architectural trade-offs in context handling is essential for choosing appropriate models for different applications and for appreciating the advances discussed in subsequent chapters:

Figure 1.21 compares how different architectures handle context, foreshadowing the detailed treatment in later chapters.

1.7 Training and Evaluation Paradigms

Before diving into specific model architectures, we establish the basic paradigm for training and evaluating language models that will recur throughout this book. Training a language model means adjusting its parameters to maximize the probability it assigns to observed text, while evaluation measures how well the trained model predicts held-out text that was not seen during training. The separation between training and evaluation data is crucial: we want to measure how well the model generalizes to new text, not merely how well it memorizes the training corpus, because real-world deployment will always involve text the model has never seen before. Good generalization is the hallmark of a successful language model, distinguishing genuine learning from mere memorization. This section introduces the mathematical framework of maximum likelihood estimation, the smoothing techniques that address data sparsity, and the metrics we use to quantify prediction quality. These foundations apply regardless of whether the model is a simple n-gram counter or a billion-parameter neural network. The mathematical objective is the same—maximize the probability of observed sequences—but the methods for achieving this objective vary dramatically across the different modeling approaches we will study in subsequent chapters.

1.7.1 Maximum Likelihood Estimation

The standard approach to training language models is **maximum likelihood estimation (MLE)**. Given a training corpus $\mathscr{D} = \{w_1, \ldots, w_T\}$, we maximize:

$$\mathscr{L}(\theta) = \sum_{t=1}^{T} \log p_\theta(w_t | w_1, \ldots, w_{t-1}) \tag{1.16}$$

Equivalently, we minimize cross-entropy loss (negative log-likelihood). Figure 1.22 visualizes the optimization landscape and the relationship between likelihood and our evaluation metric.

1.7.2 Smoothing Techniques

For count-based models, **smoothing** addresses the zero-probability problem for unseen n-grams:

The simplest approach, known as add-one or Laplace smoothing, adds a count of one to every possible n-gram before computing probabilities, ensuring that no combination receives zero probability. While this ensures no n-gram has zero probability, it tends to redistribute too much mass to unseen events, often degrading perplexity substantially on held-out test data. A refinement called add-k smoothing uses a fractional count less than one, typically tuned on held-out data to find the optimal balance between coverage of unseen events and fidelity to observed frequencies in the training corpus. Good-Turing smoothing takes a more principled approach by examining the "frequency of frequencies"—how many n-grams appear exactly once, exactly twice, and so on—and using this information to estimate the total probability mass that should be allocated to unseen events. The most sophisticated technique, Kneser-Ney smoothing, combines absolute discounting with a clever modification to the backoff distribution: instead of using the raw frequency of a word in lower-order contexts, it uses the number of distinct contexts in which the word appears, which better captures a word's versatility as a continuation rather than simply its overall frequency. Figure 1.23 compares these approaches, which we explore in depth in Chapter 2.

1.7.3 Evaluation Metrics

Figure **??** summarizes the key evaluation metrics used to assess language model quality across different dimensions and use cases. Perplexity serves as the primary intrinsic metric, directly measuring the model's ability to predict held-out text without reference to any downstream task or application-specific considerations. Lower perplexity indicates better predictions, with state-of-the-art models achieving perplexities below 10 on standard benchmarks such as WikiText-103. For comparing models with different tokenization schemes, bits per character provides a normalization that accounts for vocabulary differences: a character-level model and a subword

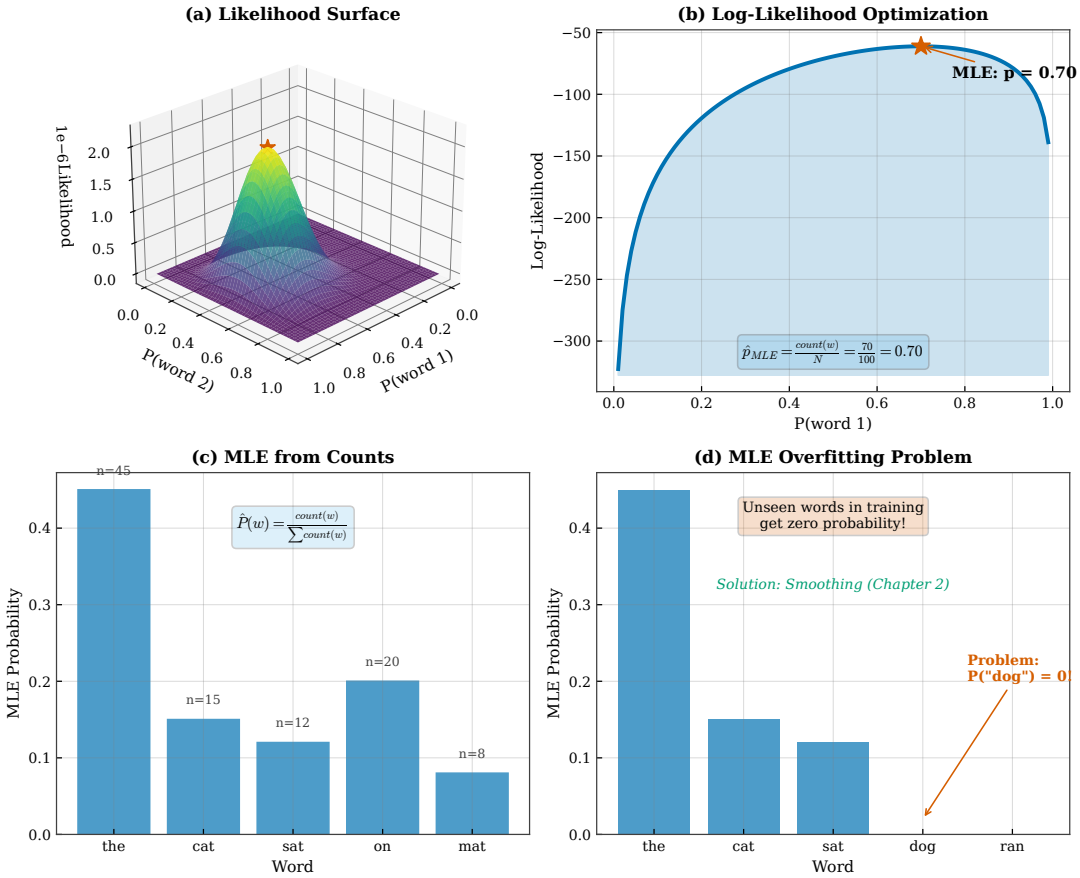


Figure 1.22: Maximum likelihood estimation for language models. Panel (a) shows a 3D likelihood surface over parameter space. Panel (b) illustrates gradient ascent optimization. Panel (c) displays the relationship between likelihood and perplexity. Panel (d) demonstrates overfitting on small datasets.

model can be directly compared through their BPC scores, even though their perplexities would be incomparable due to different prediction granularities. Beyond these intrinsic metrics, researchers increasingly evaluate language models through downstream task performance, measuring accuracy on question answering, BLEU scores on translation, or success rates on code generation benchmarks. These extrinsic evaluations capture whether the model's predictions translate into useful capabilities, though they introduce additional confounds from task-specific architectures and decoding strategies. The relationship between intrinsic perplexity and extrinsic task performance is generally positive but imperfect, motivating the use of multiple complementary evaluation approaches in practice.

1.7.4 Training Data Scale

Modern language models require massive training corpora:

Figure 1.25 shows the exponential growth in training data over the past three decades, a trajectory that has been essential to improving language model quality. In the 1990s, researchers worked with corpora containing millions of words, typically drawn from newswire text or carefully curated collections. By the 2000s, advances in storage and processing enabled training on billions of words from web crawls, substantially reducing perplexity and enabling more robust generalization. The 2020s have witnessed an explosion to trillions of tokens, with models like GPT-3 training on hundreds of billions of tokens and subsequent models pushing into the multi-trillion token regime. This exponential scaling has been accompanied by increasing sophistication in data curation: modern training corpora blend multiple sources including books, academic papers, code repositories, and filtered web text, with careful attention to deduplication, quality filtering, and domain balance. The scaling laws discussed in Chapter 10 formalize the relationship between data quantity and model performance,

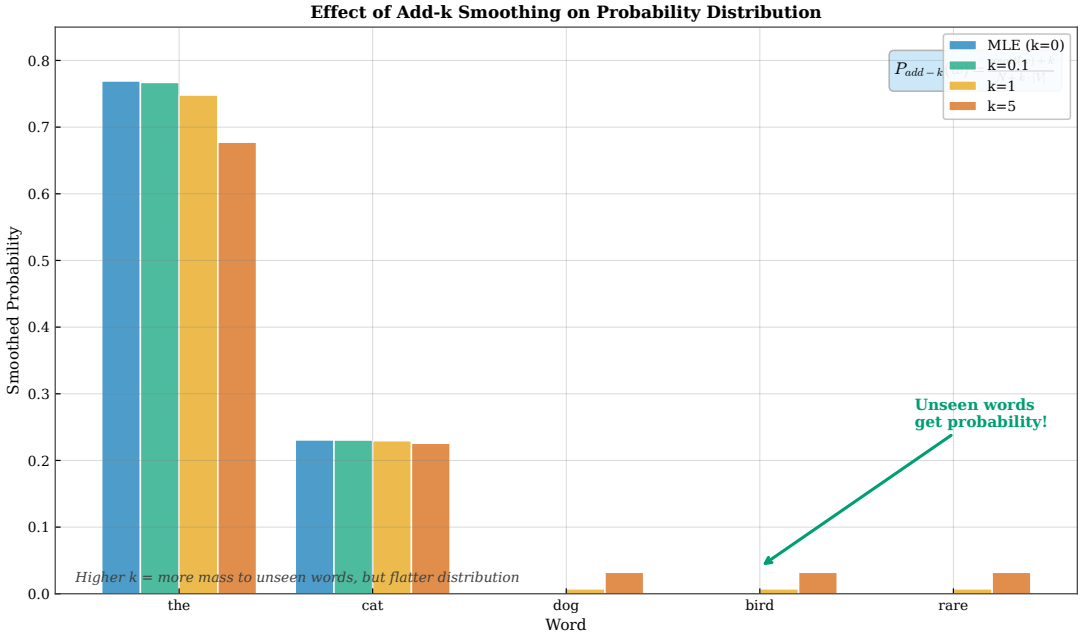


Figure 1.23: Effect of add-k smoothing on probability distributions. Different values of k redistribute probability mass from seen words to unseen words, addressing the zero-probability problem.

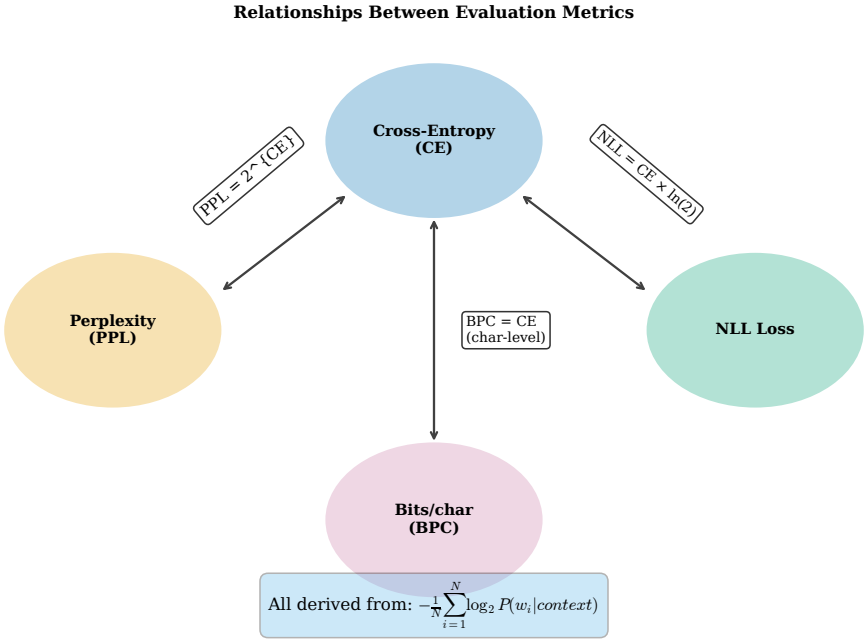


Figure 1.24: Relationships between evaluation metrics. Perplexity, cross-entropy, NLL loss, and bits-per-character are all mathematically related through the fundamental log-probability calculation.

revealing that larger models require proportionally more data to reach their optimal perplexity. This interplay between model capacity and data scale has become one of the central themes of modern large language model development.

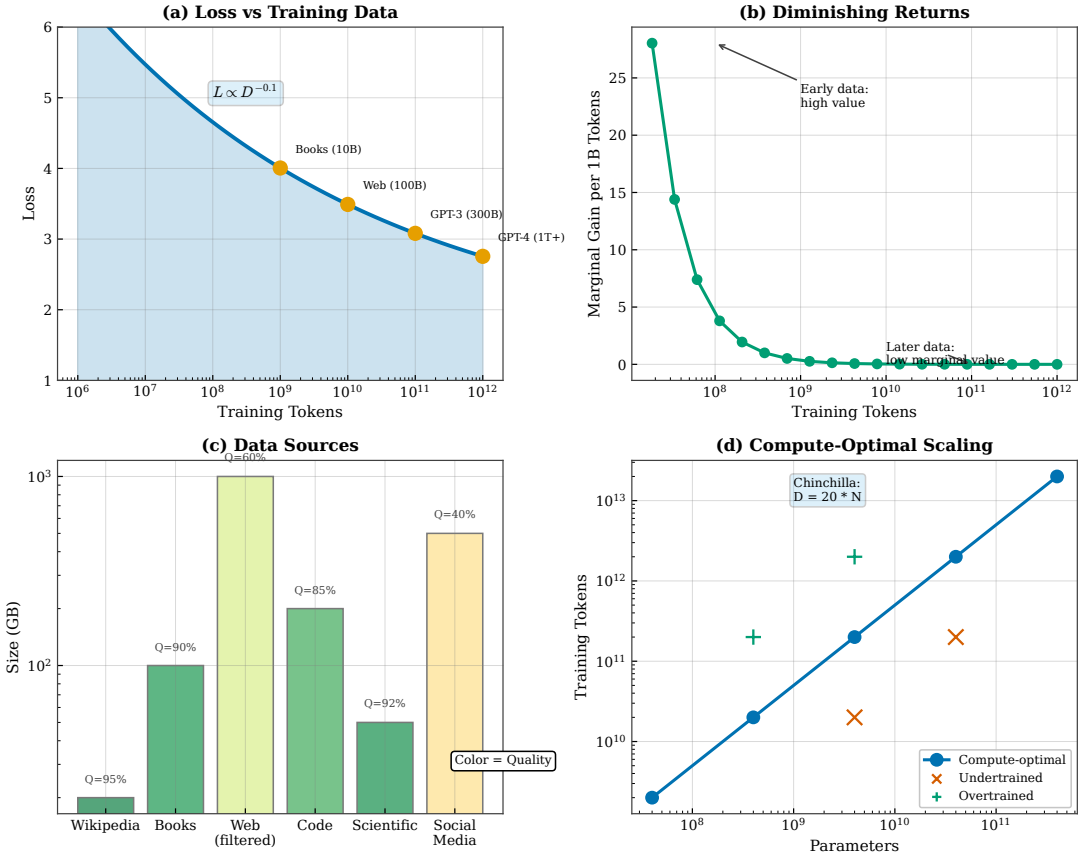


Figure 1.25: Training data scale in language modeling. Panel (a) shows the exponential growth in training data over time. Panel (b) displays the relationship between data size and model performance. Panel (c) illustrates data composition (books, web, code). Panel (d) demonstrates diminishing returns at scale.

1.8 Information Flow in Language Models

Understanding how information flows from input to prediction helps build intuition for the architectures we will study in subsequent chapters and reveals the common structure underlying seemingly different approaches to language modeling. Despite their apparent diversity, all language models share a fundamental processing pipeline that transforms raw text into probability distributions, and recognizing this shared structure makes it easier to understand each architecture's innovations and limitations. Every language model, regardless of its specific architecture, must solve the same fundamental problems: converting discrete text into a form that can be processed mathematically through tokenization and embedding, encoding contextual information into a representation that captures relevant patterns through some form of sequence processing, and transforming that representation back into a probability distribution over possible next words through a projection and normalization step. The differences between architectures lie in how they implement these stages, particularly the crucial encoding stage where context is compressed and transformed: n-grams use simple lookup tables, RNNs accumulate information through recurrent hidden states, and Transformers use attention mechanisms to selectively combine information across positions. By examining this general pipeline, we can appreciate both what different architectures share and where they diverge in their approaches to representation and computation, setting the stage for the detailed treatment of each approach in subsequent chapters.

Figure 1.26 illustrates the general pipeline that transforms raw text into probability predictions. The first stage, tokenization, converts the input text into a sequence of discrete tokens drawn from a fixed vocabulary, handling the boundary between continuous character streams and the discrete symbols that neural networks process. The second stage, embedding, maps each discrete token to a continuous vector in a high-dimensional space, enabling the model to learn rich representations of word meaning and capture similarities between related

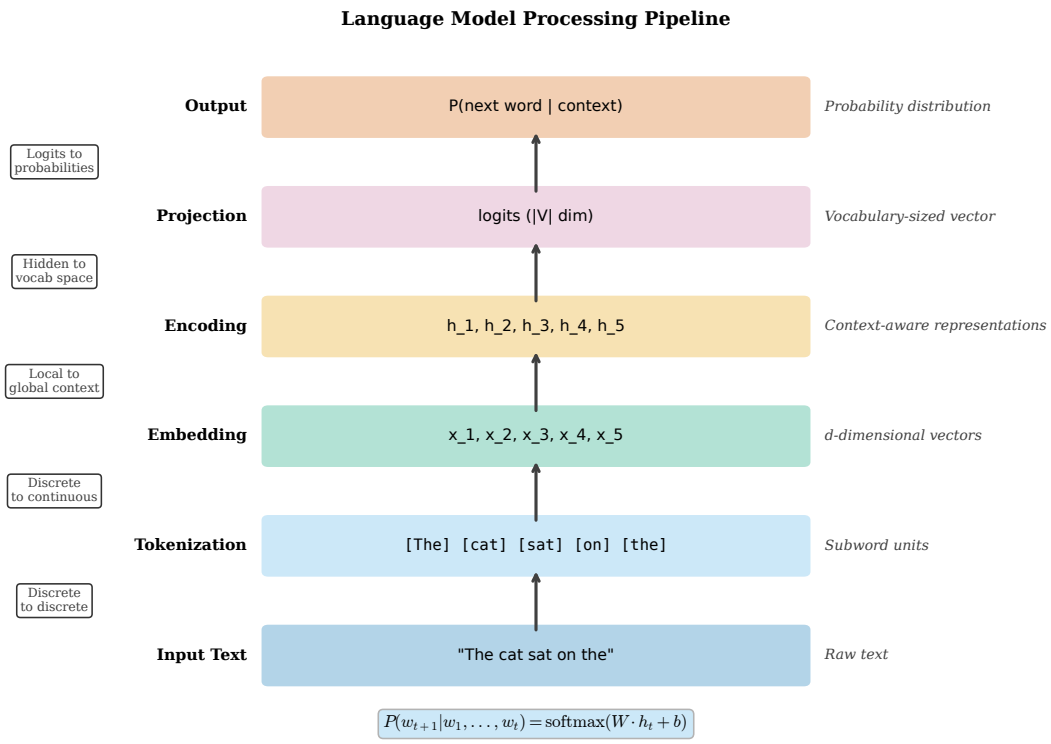


Figure 1.26: Language model processing pipeline. Every language model transforms raw text through tokenization, embedding, encoding, projection, and softmax normalization to produce a probability distribution over possible next words.

tokens. The third stage, encoding, constitutes the heart of the language model: it processes the sequence of embedding vectors and produces a contextual representation that summarizes the relevant information from the preceding context. This encoding stage is where the major architectural innovations occur—n-grams use lookup tables, RNNs use recurrent computation, and Transformers use self-attention. The final stage, projection, maps the contextual representation back to vocabulary space, producing a probability distribution over all possible next tokens through a linear transformation followed by softmax normalization. This pipeline, in various forms, appears in every modern language model, and the key differences between architectures lie primarily in the encoding stage and how context is represented and processed.

1.9 Comparing Language Model Approaches

Before diving into specific architectures in subsequent chapters, we preview the trade-offs between different approaches to give readers a roadmap of what lies ahead and to motivate the architectural choices we will examine in detail throughout this book. Each approach to language modeling makes different trade-offs along multiple dimensions including prediction accuracy as measured by perplexity, computational efficiency during both training and inference, memory requirements that determine what hardware can run the model, interpretability of predictions and internal representations, and ability to capture long-range dependencies that span many words or even sentences. No single architecture dominates across all these dimensions, which explains why multiple approaches remain relevant for different applications and deployment contexts. A simple n-gram model running on a smartphone can provide instant autocomplete suggestions with minimal battery drain, while the same device could never run a billion-parameter Transformer that achieves state-of-the-art perplexity. Conversely, applications requiring nuanced understanding of long documents must sacrifice the speed and simplicity of n-grams for the sophisticated context modeling that only neural architectures provide. Understanding these trade-offs helps practitioners choose appropriate models for their specific constraints and

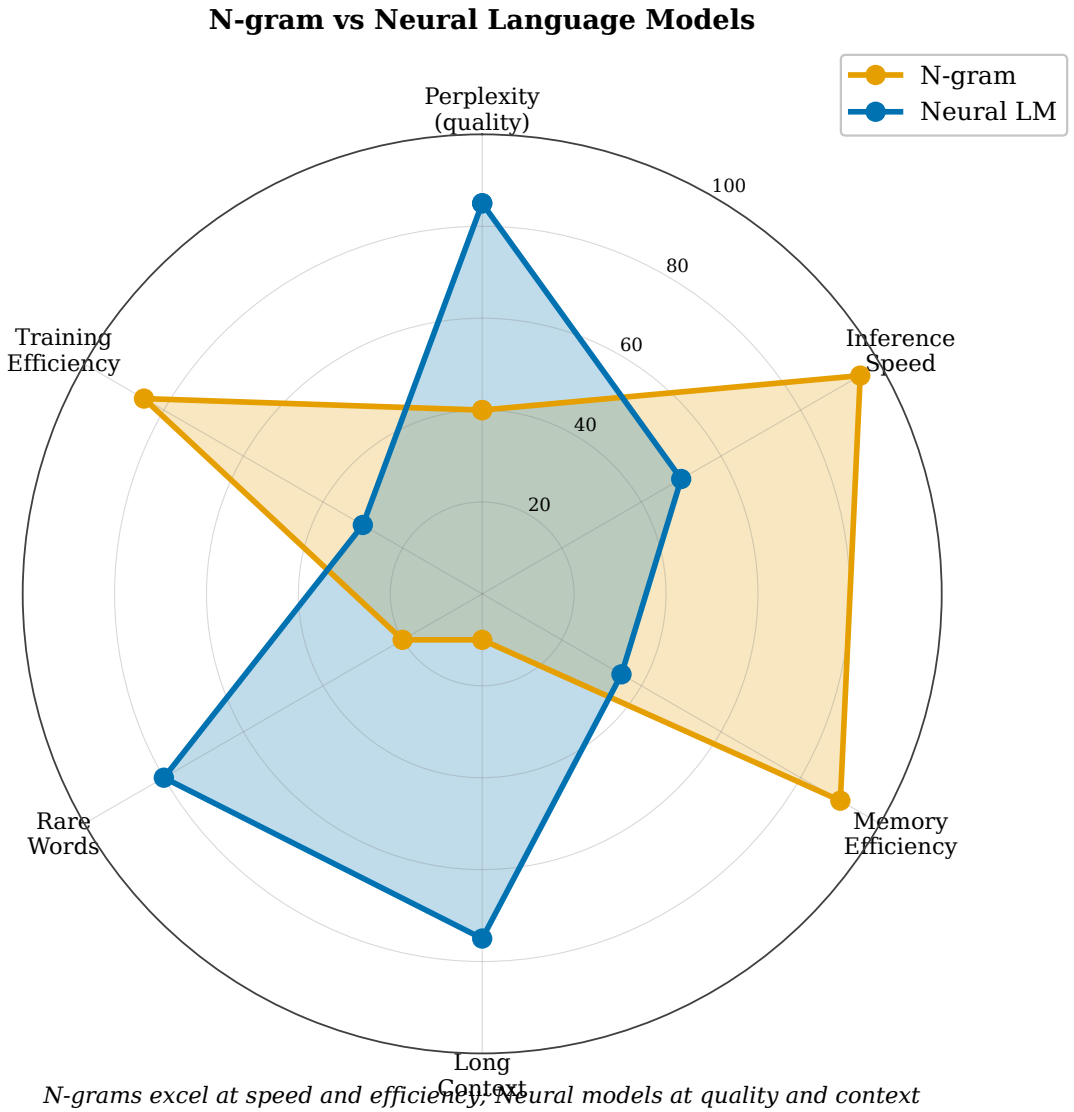


Figure 1.27: Radar chart comparing n-gram and neural language models across multiple dimensions. N-grams excel at speed and memory efficiency while neural models achieve better perplexity and long-range context.

helps researchers identify opportunities for improvement by recognizing which dimensions current approaches sacrifice and whether architectural innovations might recover that lost ground.

Figure 1.27 compares four major approaches across multiple dimensions including perplexity, inference speed, memory requirements, and interpretability. N-gram models offer the fastest inference and most interpretable predictions, since probabilities derive directly from observable count statistics, but their fixed and limited context windows prevent them from capturing long-range dependencies and lead to higher perplexity on complex text. Recurrent neural networks and LSTMs address the context limitation by maintaining hidden states that theoretically encode unbounded history, achieving substantially lower perplexity than n-grams, but their sequential processing prevents parallelization and limits training efficiency. The Transformer architecture revolutionized the field by enabling full parallelization through self-attention, achieving even lower perplexity and faster training, though the quadratic memory scaling with sequence length poses challenges for very long contexts. Large language models built on Transformer foundations exhibit emergent abilities—including in-context learning, chain-of-thought reasoning, and instruction following—that smaller models lack entirely, but these capabilities require enormous computational resources for both training and inference. No single model is best for all tasks, and the choice depends on computational budget, latency requirements, task complexity, and whether emergent capabilities are necessary for the application at hand.

1.10 Prediction Spotlight

Throughout this book, we track prediction quality using a running example that demonstrates how different modeling approaches handle the same prediction task, allowing readers to build intuition for the progressive improvements in language modeling that each chapter introduces. This running example serves as a concrete touchstone that grounds abstract architectural discussions in observable prediction behavior: rather than simply asserting that Transformers outperform n-grams, we show how the predictions differ on identical input. The example we use involves predicting the continuation of a sentence about research findings, a context that requires both syntactic knowledge about English sentence structure and semantic knowledge about how researchers typically describe their results. By returning to this same example across multiple chapters, readers can directly compare how a bigram model's prediction based on only the previous word differs from an LSTM's prediction that incorporates the full sentence history, which in turn differs from a Transformer's attention-weighted integration of all context positions. This comparative approach makes the benefits of architectural advances tangible and memorable, transforming abstract perplexity improvements into concrete differences in predicted word distributions. Consider the prompt:

"The researcher analyzed the data and found that the results _____"

Model	Prediction	Quality
Uniform (50K vocab)	Random word	Terrible
Unigram	"the" (most common)	Poor
Human (Shannon-level)	"were" or "showed"	Good
Modern LLM	"were consistent with..."	Excellent

Table 1.1: Prediction quality across model types. This chapter establishes the baseline; subsequent chapters show progressive improvement.

The journey from uniform random to sophisticated context-aware prediction is the story of this book.

1.11 Context Representation: The Central Challenge

How This Chapter Represents Context
The fundamental question in language modeling is: How do we represent the context $w_1, \ldots, w_{t-1}$ to predict w_t?

- **Context representation**: This chapter introduces context as the key variable, without specifying how to encode it
- **Key insight**: Different contexts require different predictions—"the cat sat on the" vs. "the capital of France is"
- **Limitation**: We have not yet specified how to encode context computationally

Different approaches make different trade-offs:

- **Fixed window**: Use only the last $n-1$ words (n-grams, Chapter 2)
- **Unbounded history**: Theoretical ideal, but computationally challenging
- **Compressed representation**: Encode context in a fixed-size vector (Chapters 5–6)

This chapter has introduced the fundamental problem of next-word prediction and the information-theoretic framework we use to evaluate solutions; subsequent chapters develop increasingly sophisticated approaches to actually solving this problem through better context representation. The evolution of context representation—from the fixed windows of n-gram models through the recurrent hidden states of LSTMs to the dynamic attention mechanisms of Transformers—constitutes the central technical arc of this book, with each advance enabling qualitatively better predictions by capturing more of the structure inherent in natural language. Understanding context representation as the central challenge unifies the diverse techniques we will study: every architectural innovation can be understood as a new way to encode what came before in a form that supports accurate prediction of what comes next. This perspective helps us appreciate both why older methods work as well as they do and why newer methods improve upon them. The n-gram's fixed window is a crude but computationally efficient context representation; the RNN's hidden state is a more flexible but sequentially constrained representation; the Transformer's attention-weighted combination is yet more flexible and parallelizable. Each step in this progression expands what information about the context can flow into the prediction, bringing us closer to the theoretical ideal of conditioning on all available information.

1.12 Roadmap of This Book

Figure 1.28 shows the structure of this book and the dependencies between chapters. The book begins with foundational material in Chapters 2 and 3, covering n-gram language models and tokenization respectively—these topics establish the vocabulary and baselines against which all subsequent methods are compared. Chapters 4 through 6 introduce the neural revolution, progressing from static word embeddings through recurrent neural networks to the Transformer architecture that dominates modern language modeling. Chapter 7 addresses the decoding problem: given a trained language model, how do we actually generate text? This chapter bridges the gap between having a probability distribution and producing useful output. Chapter 8 examines training at scale, covering optimization algorithms, distributed training, and data considerations. Chapters 9 and 10 explore the frontier of large language models and the scaling laws that govern their behavior, including emergent capabilities and the compute-optimal training debate. Chapter 11 covers post-training alignment: instruction tuning, RLHF, and DPO methods that transform raw language models into helpful assistants. Chapter 12 addresses efficiency through quantization, pruning, and distillation techniques that make deployment practical. Finally, Chapter 13 surveys applications from translation to code generation.

The book accommodates different reading goals through multiple paths. Researchers seeking comprehensive understanding should read sequentially from Chapters 1 through 13, building intuition at each stage. Practitioners focused on deploying modern systems can follow a condensed path through Chapters 1, 3, 6, 7, 9, 12, and 13, covering the essential knowledge for working with contemporary language models. Readers seeking a quick conceptual overview should focus on Chapters 1, 6, 9, and 13 to understand the core ideas and current state of the field. Those with a theoretical bent should prioritize Chapters 1, 2, 4, 6, 8, 10, and 11, which contain the deepest mathematical content and formal analysis.

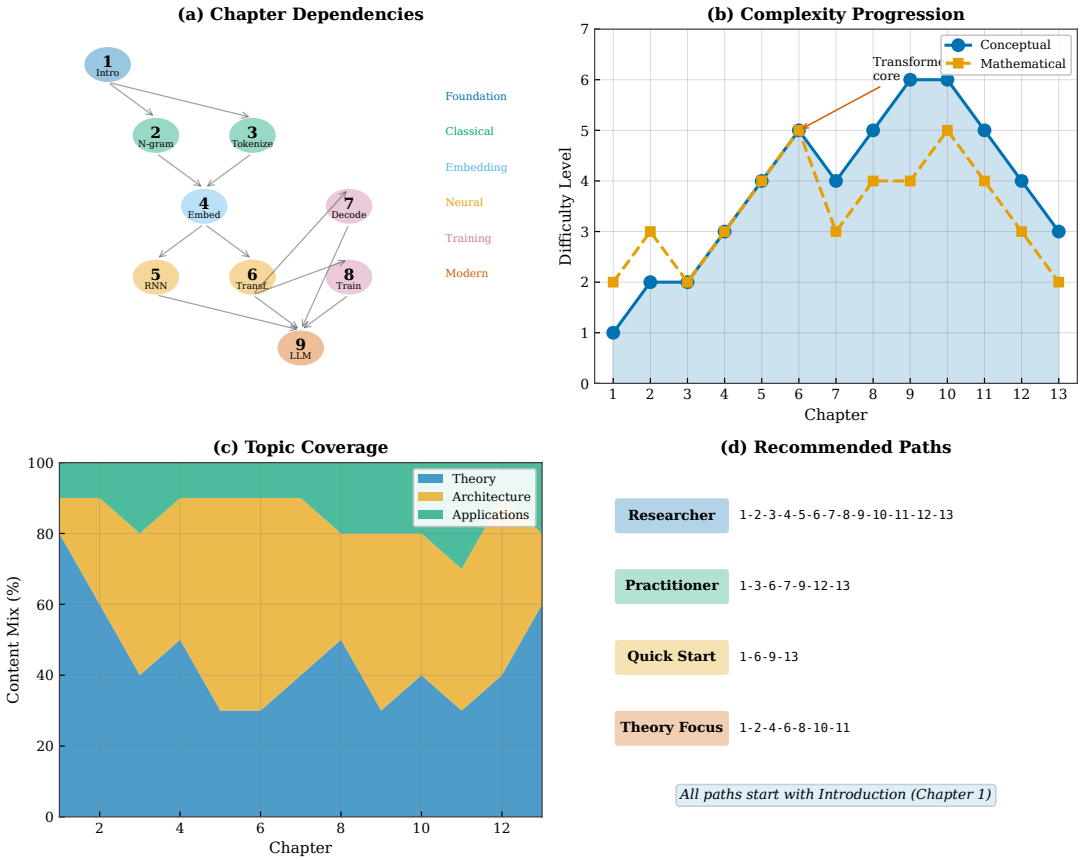


Figure 1.28: Book structure and reading paths. Panel (a) shows chapter dependencies as a directed graph. Panel (b) displays complexity progression across chapters. Panel (c) illustrates topic coverage (theory, architecture, applications). Panel (d) suggests reading paths for different goals.

1.13 Summary

We can now predict better because:

- We understand the formal objective: $P(w_t \mid w_1, \ldots, w_{t-1})$
- Information theory provides our theoretical foundation and evaluation metric (perplexity)
- The historical arc from Shannon to modern LLMs shows progressive improvements in context representation
- Linguistic regularities (Zipf's law) inform our modeling choices

Next: Chapter 2 develops our first concrete solution—n-gram language models that use fixed-length context windows.

Exercises

1. Prove that entropy $H(X)$ is maximized when X is uniformly distributed over n values.

2. Given a corpus where word w appears with frequency f_w, derive the maximum likelihood estimate for $P(w)$ and show it equals $f_w / \sum_v f_v$.

3. Show that perplexity of a uniform distribution over V words equals V.

4. Calculate the entropy of a bigram distribution where $P(\text{cat} \mid \text{the}) = 0.6$, $P(\text{dog} \mid \text{the}) = 0.3$, and $P(\text{bird} \mid \text{the}) = 0.1$.

5. Prove that cross-entropy $H(p,q) \geq H(p)$ with equality if and only if $q = p$. (Hint: Use Jensen's inequality or the non-negativity of KL divergence.)

6. * Derive the relationship between cross-entropy loss and KL divergence for language models. Show that minimizing cross-entropy is equivalent to minimizing KL divergence to the empirical distribution.

7. Show that for a language model, perplexity equals the geometric mean of the inverse probabilities: $\text{PPL} = \left(\prod_{t=1}^{T} \frac{1}{P(w_t \mid w_{<t})}\right)^{1/T}$.

8. **Research Question**: Read Shannon's 1951 paper on predicting English text. How do his human prediction experiments compare to modern language model perplexities? What does this suggest about the lower bound on English entropy?

9. **Research Question**: Why might Zipf's law emerge in natural language? Discuss at least two proposed explanations from the literature (e.g., least effort principle, preferential attachment).

10. **Research Question**: Modern LLMs achieve perplexities below 10 on common benchmarks like WikiText-103. What does this suggest about the entropy of English text? Is there a theoretical limit?

11. **Research Question**: Compare the chain rule decomposition (left-to-right) with other possible decompositions. What are the implications for bidirectional models like BERT?

12. * Prove that the entropy rate of a stationary stochastic process $\lim_{n\to\infty} \frac{1}{n} H(w_1, \ldots, w_n)$ exists and equals $\lim_{n\to\infty} H(w_n | w_1, \ldots, w_{n-1})$.

13. * Design an experiment to empirically estimate the entropy of English text using human subjects, following Shannon's methodology. What controls would you need? How would you account for individual differences in language knowledge?

Bibliography

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2015.

Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. A neural probabilistic language model. *Journal of Machine Learning Research*, 3:1137–1155, 2003.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, et al. Language models are few-shot learners. *NeurIPS*, 33:1877–1901, 2020.

DeepSeek-AI. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint*, 2025.

Irving J Good. The population frequencies of species and the estimation of population parameters. *Biometrika*, 40(3-4):237–264, 1953.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

Frederick Jelinek. *Self-organized language modeling for speech recognition*. Morgan Kaufmann, 1990.

Dan Jurafsky and James H Martin. *Speech and Language Processing*. Prentice Hall, 3rd edition, 2024. Draft available at https://web.stanford.edu/ jurafsky/slp3/.

Reinhard Kneser and Hermann Ney. Improved backing-off for m-gram language modeling. In *ICASSP*, volume 1, pages 181–184, 1995.

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

Long Ouyang, Jeffrey Wu, Xu Jiang, et al. Training language models to follow instructions with human feedback. *NeurIPS*, 35:27730–27744, 2022.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI Blog*, 1(8):9, 2019.

Rafael Rafailov, Archit Sharma, Eric Mitchell, et al. Direct preference optimization: Your language model is secretly a reward model. *arXiv preprint arXiv:2305.18290*, 2023.

Claude E Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3): 379–423, 1948.

Claude E Shannon. Prediction and entropy of printed english. *Bell System Technical Journal*, 30(1):50–64, 1951.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *NeurIPS*, 30, 2017.

George Kingsley Zipf. Human behavior and the principle of least effort. *Addison-Wesley Press*, 1949.

Index

Predicting the Next Word

From Shannon to ChatGPT

Test Compilation - Chapter 2

Contents

Chapter 2

N-gram Language Models

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

- Introducing the Markov assumption to make prediction tractable
- Estimating probabilities directly from word counts in corpora
- Developing smoothing techniques to handle unseen word sequences
- Understanding the fundamental trade-offs between context and data sparsity

2.1 N-grams and Next-Word Prediction

The fundamental challenge of next-word prediction lies in estimating conditional probabilities of the form $P(w_t \mid w_1, w_2, \ldots, w_{t-1})$, where the conditioning context can grow arbitrarily long as we proceed through a text. As we saw in Chapter **??**, naive estimation of these probabilities faces an exponential explosion: a vocabulary of just 50,000 words and contexts of length 10 would require storing $50,000^{10}$ parameters, a number exceeding the atoms in the observable universe. N-gram models provide the first practical solution to this challenge by making a bold simplifying assumption: the probability of the next word depends only on the preceding $n-1$ words, not the entire history. This **Markov assumption** transforms an intractable estimation problem into one that can be solved by counting occurrences in a training corpus. Despite their simplicity, n-gram models dominated natural language processing for over three decades, powering speech recognition systems, machine translation engines, spelling correctors, and countless other applications. Understanding n-gram models is essential not only for historical perspective but because they illuminate fundamental concepts—sparsity, smoothing, and the bias-variance trade-off—that remain central to modern neural language models.

The term "n-gram" refers to a contiguous sequence of n words from a text. A unigram is a single word, a bigram is a pair of consecutive words, a trigram is a triple, and so forth. When we speak of an "n-gram model," we mean a language model that conditions next-word predictions on the preceding $n-1$ words, using the statistics of n-grams observed in training data. Figure 2.1 illustrates this as a Markov chain where states represent words and transition probabilities capture the likelihood of moving from one word to another. The Markov property states that the future is conditionally independent of the past given the present: knowing the current state provides all the information needed to predict the next state, making earlier history irrelevant. In the context of language modeling, this means that once we know the last $n-1$ words, learning about words even further back provides no additional predictive power—a strong assumption that is clearly false for natural language but nevertheless proves useful as a computational approximation. The chain structure also reveals that n-gram models define a generative process: we can generate text by starting from a special beginning-of-sentence token and repeatedly sampling from the conditional distribution until we reach an end-of-sentence token, with each sampled word depending only on its immediate predecessors.

The choice of n represents a fundamental trade-off in language modeling that we will encounter repeatedly

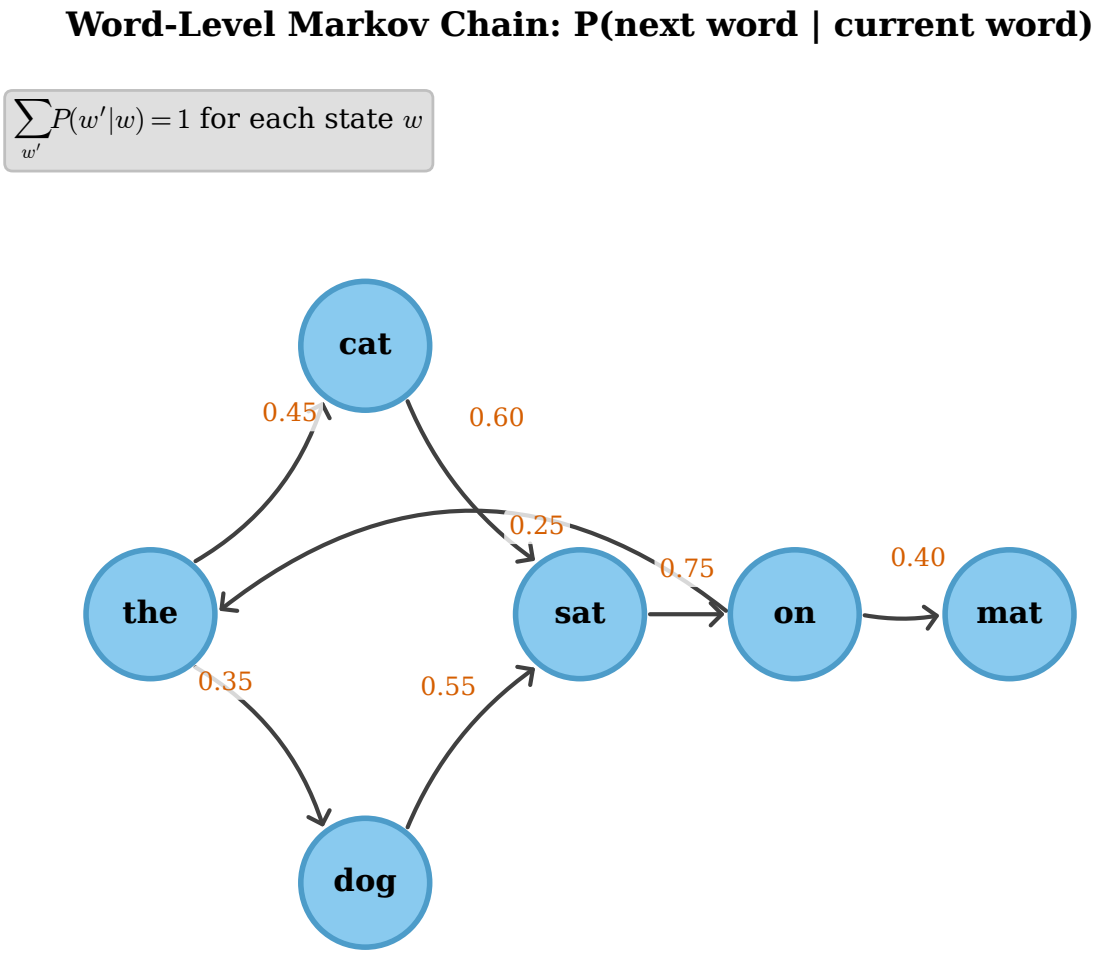


Figure 2.1: A word-level Markov chain illustrating n-gram dependencies. Each node represents a word state, and edges show transition probabilities between consecutive words. The probabilities on outgoing edges from each state sum to one, ensuring a valid probability distribution. This visualization captures the essence of bigram models where predictions depend only on the immediately preceding word.

throughout this book. Larger values of n capture more context, potentially improving prediction accuracy by considering longer-range dependencies in the text. A trigram model can distinguish between "New York" and "New Jersey" where a bigram model seeing only "New" cannot; a 5-gram model can capture idiomatic phrases like "at the end of the day" that shorter models would miss. However, larger n values come with severe costs: the number of possible n-grams grows exponentially with n, as there are $|\mathcal{V}|^n$ possible sequences of length n for a vocabulary $\mathcal{V}$. This exponential growth means that most n-grams will never appear in any finite training corpus, creating the **sparsity problem** that dominated n-gram research for decades. When we encounter a word sequence never seen in training, the raw count-based probability estimate is zero, which causes catastrophic failures when computing sequence probabilities since a single zero factor makes the entire product zero. The entire field of smoothing techniques arose to address this challenge, redistributing probability mass from seen events to unseen events in principled ways that we will explore in detail in Section 2.4.

Figure 2.2 visualizes how different n-gram orders define progressively larger context windows for next-

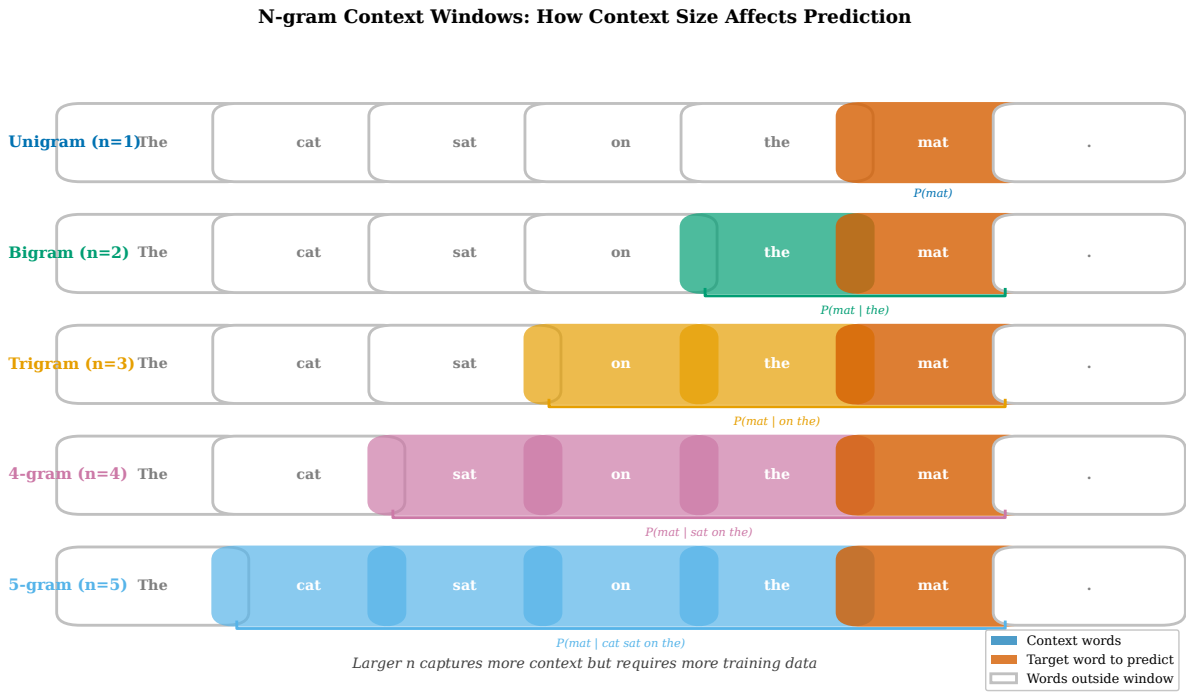


Figure 2.2: N-gram context windows from unigram through 5-gram. The target word to predict is highlighted in red, while context words are shown in the corresponding n-gram color. Larger n-gram orders capture more context but require exponentially more training data to estimate reliably. The probability notation shows how the conditioning set grows with n.

word prediction. The unigram model ignores context entirely, predicting words based solely on their overall frequency in the training corpus—"the" is predicted with high probability regardless of what preceded it, simply because "the" is the most common word in English. This context-free approach captures the basic distribution of words but misses all sequential structure. Bigram models condition on exactly one preceding word, capturing basic sequential dependencies like the tendency for articles to precede nouns, verbs to precede their objects, and prepositions to precede noun phrases. This single word of context provides substantial predictive power: knowing the previous word was "the" dramatically changes predictions compared to knowing it was "quickly." Trigram models extend this to two preceding words, capturing phrase-level patterns and disambiguating cases where single-word context is insufficient. Each additional word of context provides more information about the likely continuation but requires correspondingly more training data to estimate the resulting probabilities reliably. The growth in required data is not merely linear but combinatorial, as longer contexts create exponentially more possible patterns to estimate. In practice, n-gram models with $n > 5$ rarely improve performance because the contexts become so specific that they almost never repeat between training and test data, making the additional context effectively useless even though it theoretically contains valuable information about the intended continuation.

The historical dominance of n-gram models from the 1980s through the 2010s reflects their remarkable practical utility despite their theoretical limitations. At IBM, researchers developed language models that transformed speech recognition from a curiosity into a practical technology by combining acoustic models with n-gram language models that favored linguistically plausible word sequences [Jelinek, 1990]. When the acoustic signal was ambiguous between "recognize speech" and "wreck a nice beach," the language model strongly favored the former, dramatically improving transcription accuracy. Similar success followed in statistical machine translation, where target-language n-gram models guided the decoder toward fluent output even when the translation model produced awkward word-by-word correspondences. Spelling correction, text classification, authorship attribution, and information retrieval all benefited from n-gram language models during this golden age of statistical NLP. Figure 2.3 traces this history, showing how n-gram models evolved from theoret-

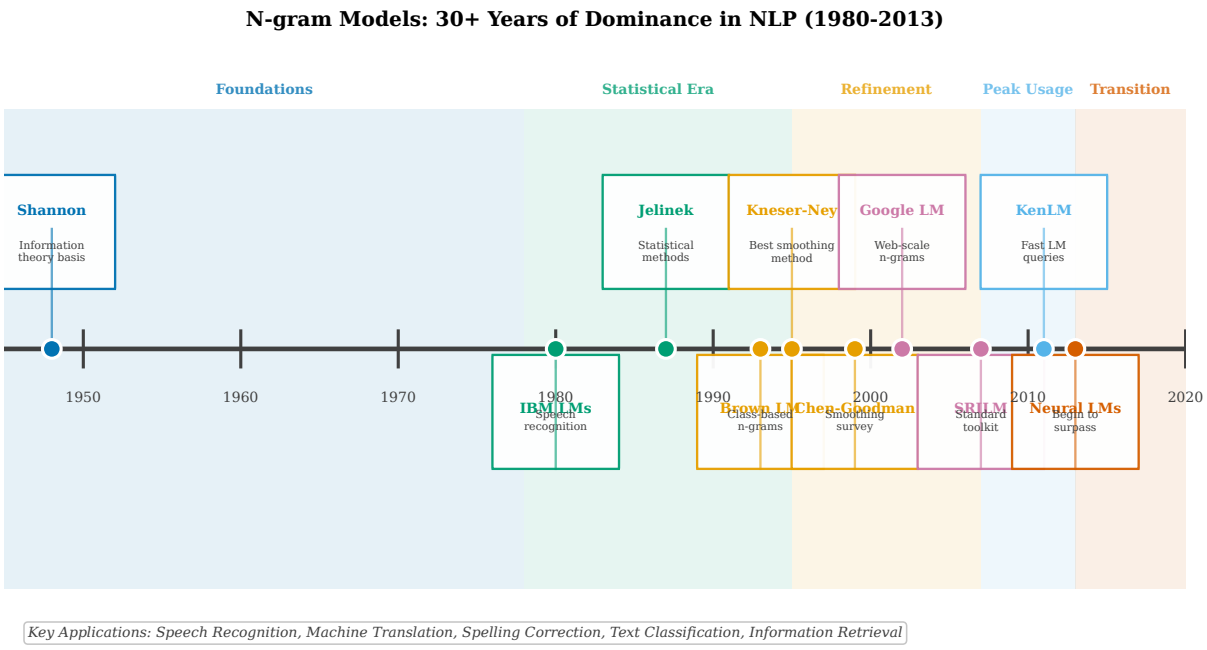


Figure 2.3: Historical impact of n-gram models in NLP from 1948 to 2013. Key milestones include Shannon's information-theoretic foundations, IBM's speech recognition systems, the development of sophisticated smoothing techniques, and the eventual transition to neural approaches. N-gram models dominated practical NLP for over three decades.

ical curiosity to practical necessity before eventually yielding to neural approaches that could overcome their fundamental limitations.

The transition from n-gram to neural language models beginning around 2013 did not render n-gram concepts obsolete; rather, it provided new tools to address the same fundamental challenges that have always confronted language modeling. The sparsity problem that motivated decades of n-gram smoothing research reappears in neural models as the challenge of generalizing from training data to unseen inputs—neural networks must also learn to handle novel word combinations not explicitly observed during training. The bias-variance trade-off between short contexts that estimate reliably but predict poorly versus long contexts that could predict well but estimate unreliably finds its neural analog in the trade-off between model capacity and overfitting: larger neural networks can capture more complex patterns but risk memorizing training data rather than learning generalizable rules. The principle that we should back off from sparse contexts to more reliable shorter contexts anticipates the hierarchical representations that deep neural networks learn, where earlier layers capture local patterns like character n-grams and morphology while later layers integrate global context spanning entire sentences or documents. Even the specific techniques matter: the continuation probability insight from Kneser-Ney smoothing—that word versatility matters more than raw frequency—foreshadows the distributional semantics that neural word embeddings capture. Understanding n-gram models thus provides not merely historical context but conceptual foundations that illuminate the design of modern architectures and the challenges they continue to face in predicting the next word.

2.2 The Markov Assumption and Maximum Likelihood Estimation

The mathematical foundation of n-gram language models rests on two pillars: the chain rule of probability, which decomposes joint probabilities into products of conditionals, and the Markov assumption, which truncates these conditionals to depend only on recent history. To compute the probability of a sentence $w_1, w_2, \ldots, w_T$, we apply the chain rule to factor the joint probability as $P(w_1, w_2, \ldots, w_T) = P(w_1) \cdot P(w_2|w_1) \cdot P(w_3|w_1, w_2) \cdots P(w_T|w_1, \ldots, w_{T-1})$. This decomposition is mathematically exact—it follows directly from the definition of conditional probability and holds for any probability distribution over sequences—but the condi-

tioning contexts grow without bound as we proceed through the sentence, making direct estimation from finite data impossible. With a vocabulary of 50,000 words, estimating $P(w_{100}|w_1,\ldots,w_{99})$ would require observing the same 99-word prefix multiple times, which essentially never happens. The k-th order Markov assumption resolves this intractability by asserting that $P(w_t \mid w_1,\ldots,w_{t-1}) \approx P(w_t \mid w_{t-k},\ldots,w_{t-1})$: the probability of the next word depends only on the preceding k words, regardless of what came before. For bigram models ($k = 1$), this means $P(w_t \mid w_1,\ldots,w_{t-1}) \approx P(w_t \mid w_{t-1})$, reducing the infinite-history problem to one requiring only pairwise word statistics that appear frequently enough to estimate reliably. This assumption is linguistically naive—long-range dependencies abound in natural language, from pronoun resolution to discourse coherence—but it transforms an intractable estimation problem into one that can be solved by the simple operation of counting co-occurrences in a training corpus.

Once we adopt the Markov assumption, the question becomes how to estimate the required conditional probabilities from a training corpus. The principle of **Maximum Likelihood Estimation** (MLE) provides the answer: choose parameter values that maximize the probability the model assigns to the observed training data. For n-gram models, this principle leads to a remarkably simple recipe: the MLE estimate of a conditional probability equals the ratio of counts. For bigrams, $\hat{P}_{\text{MLE}}(w_2|w_1) = \frac{\text{count}(w_1,w_2)}{\text{count}(w_1)}$, where $\text{count}(w_1,w_2)$ is the number of times the bigram (w_1,w_2) appears in training data and $\text{count}(w_1)$ is the number of times w_1 appears. This formula has intuitive appeal: if "the cat" appears 847 times and "the" appears 3,412 times, we estimate $P(\text{cat}|\text{the}) = 847/3412 \approx 0.248$. The MLE principle can be derived by writing the likelihood function as a product over all training bigrams, taking the logarithm to convert products to sums, and differentiating with respect to the probability parameters subject to the constraint that probabilities sum to one. The result is the count ratio formula, which emerges as the unique maximizer of the training data likelihood.

Figure 2.4 provides a three-dimensional visualization of bigram probabilities for a small vocabulary, offering geometric intuition for the statistical structure that n-gram models capture. The dramatic height variations across the surface reveal the non-uniform structure of natural language: certain word transitions are highly probable while most are vanishingly rare, creating a landscape of sharp peaks rising from a nearly flat plain. The tall peaks correspond to grammatically natural sequences—articles followed by nouns, nouns followed by verbs, verbs followed by prepositions, and conjunctions followed by pronouns—while the flat regions near zero represent word pairs that rarely or never co-occur, such as consecutive articles or prepositions followed by punctuation. This visualization makes concrete the linguistic regularities that n-gram models capture: despite treating words as atomic symbols with no understanding of meaning or grammar, the statistical patterns of co-occurrence encode substantial information about language structure. The model has no concept of "noun" or "verb" as grammatical categories, yet the probability surface implicitly reflects these categories through the patterns of which words follow which. The sparse nature of the surface, with most values near zero and occasional tall peaks, also foreshadows the sparsity problem that dominates n-gram research: as vocabulary size grows from tens to thousands to tens of thousands, the proportion of zero-probability cells increases rapidly toward 100%, and we must develop sophisticated techniques to assign non-zero probability to the unseen word pairs that inevitably appear in test data.

The count matrix underlying MLE estimation reveals the severity of the sparsity problem. Figure 2.5 displays both the raw bigram counts and a binary visualization of observed versus unobserved pairs. Even with a vocabulary of just ten words, many cells in the matrix contain zeros—word pairs that never appeared in training data. The logarithmic color scale in the left panel shows counts spanning several orders of magnitude, from single occurrences to thousands, while the right panel starkly divides the matrix into seen (non-zero) and unseen (zero) cells. The sparsity statistics are sobering: even this tiny vocabulary exhibits significant gaps in coverage. For realistic vocabularies of 50,000 or more words, the matrix contains $|\mathscr{V}|^2 = 2.5 \times 10^9$ cells, and even massive training corpora of billions of words cannot hope to observe all possible bigrams. The situation worsens dramatically for higher-order n-grams: trigram models have $|\mathscr{V}|^3$ possible contexts, 4-gram models have $|\mathscr{V}|^4$, and so on. This exponential growth in the parameter space relative to the polynomial growth of training data is the fundamental mathematical obstacle that n-gram models must overcome.

Figure 2.6 traces the complete MLE pipeline from corpus to probability estimates. The training sentences yield bigram counts through a simple enumeration: we slide a window of width two across each sentence, incrementing the count for each observed word pair. The count table aggregates these observations, showing how

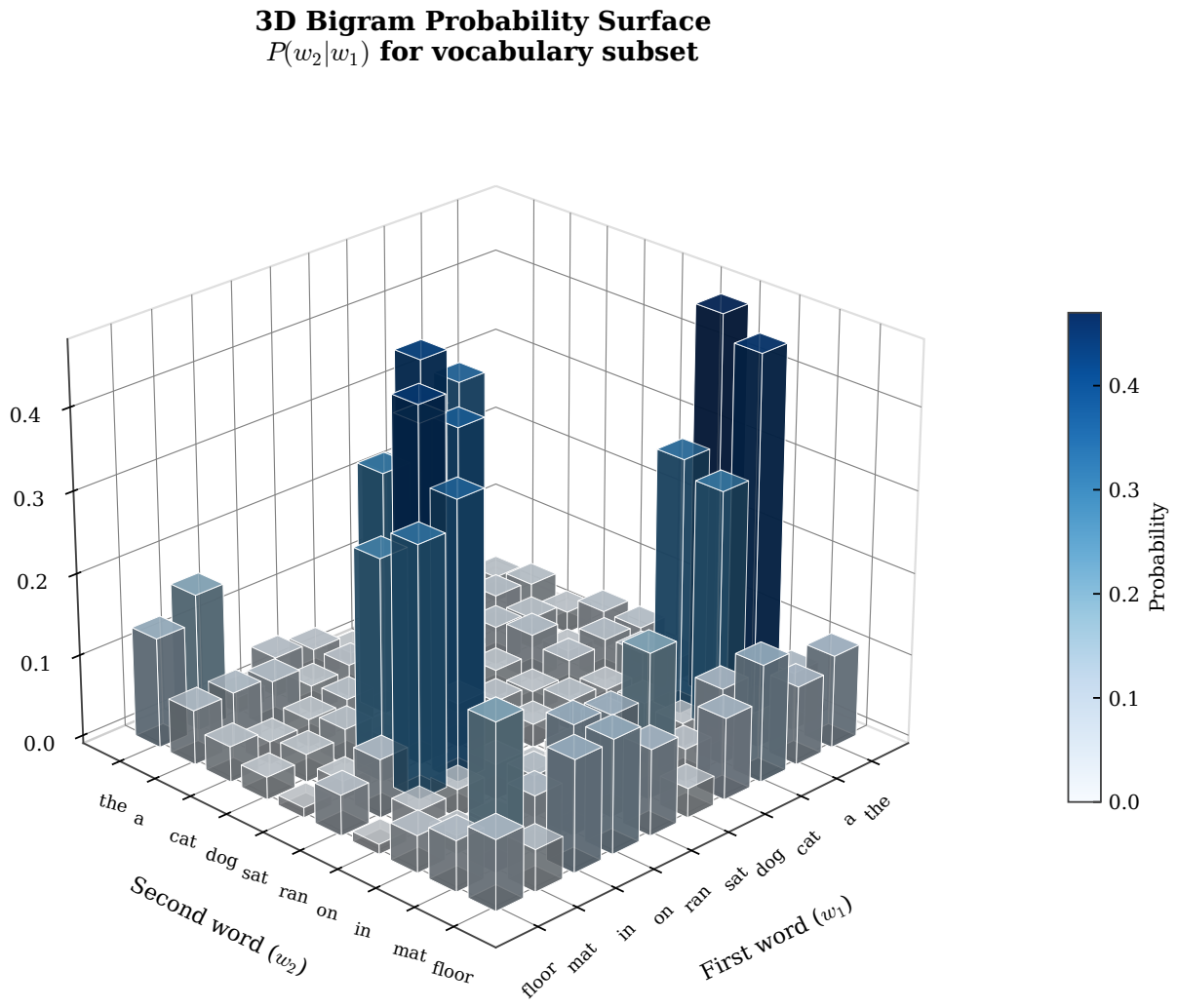


Figure 2.4: 3D visualization of bigram probabilities $P(w_2|w_1)$ for a vocabulary subset. Each bar represents the conditional probability of the second word given the first. The height variations reveal linguistic structure: articles precede nouns, nouns precede verbs, and verbs precede prepositions with high probability. Most word pairs have near-zero probability, illustrating the sparsity inherent in natural language.

frequently each bigram appears. The normalization step converts counts to probabilities by dividing each bigram count by the count of its first word, ensuring that the conditional distribution $P(\cdot|w_1)$ sums to one for each conditioning word w_1. The example calculations make the arithmetic concrete: $P(\text{sat}|\text{cat}) = 2/3 \approx 0.67$ means that two-thirds of the time we see "cat" in training, it is followed by "sat." The key insight box emphasizes MLE's Achilles' heel: if a bigram never appears in training, its count is zero, and dividing zero by any positive number yields zero. This means MLE assigns probability zero to any word sequence containing an unseen bigram, which is catastrophic when evaluating test data that inevitably contains novel word combinations.

The consequences of zero probabilities extend beyond individual bigrams to entire sentences. Consider computing the probability of a sentence using the chain rule: $P(w_1, \ldots, w_T) = \prod_{t=1}^{T} P(w_t|w_{t-1})$. If even a single factor in this product is zero, the entire sentence probability becomes zero, regardless of how well the model predicted the other words. This means that a single unseen bigram—perhaps a newly coined term, a rare proper name, or an unusual but grammatical construction—causes the model to assign zero probability to an entire sentence, text, or document. When used in applications like speech recognition or machine translation, this manifests as the model declaring certain outputs impossible rather than merely improbable. The problem worsens when we compute perplexity, the standard evaluation metric for language models, which involves

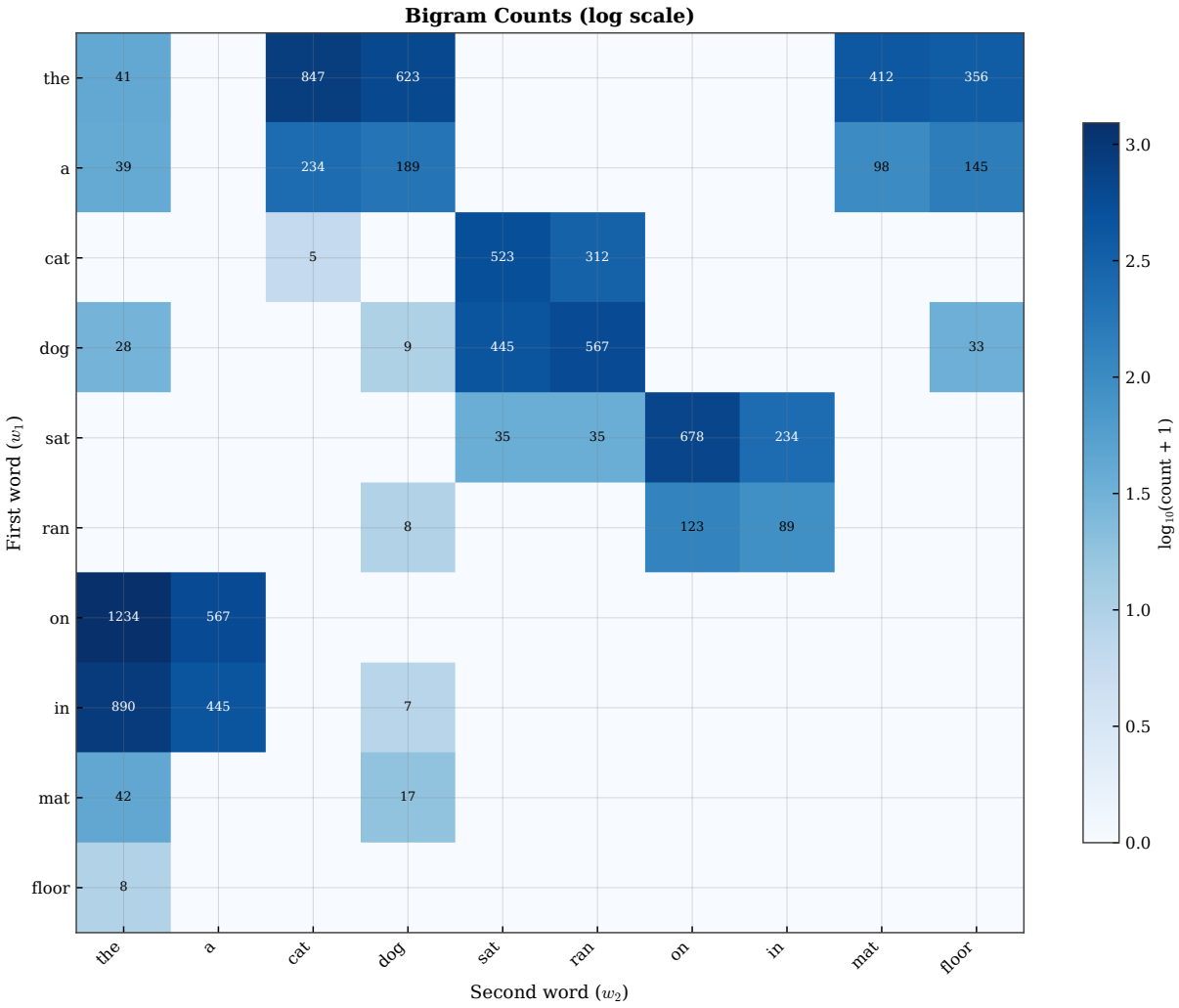


Figure 2.5: Bigram count matrix visualization. (a) Raw counts on a logarithmic scale, showing the wide dynamic range from rare to common bigrams. (b) Binary observed/unobserved pattern, revealing that most potential bigrams never appear in training data. The sparsity statistics quantify the severity of this problem even for small vocabularies.

taking the geometric mean of inverse probabilities: if any probability is zero, the perplexity becomes infinite, making model comparison meaningless. Smoothing techniques, which we explore in Section 2.4, address this problem by redistributing probability mass from observed events to unobserved events, ensuring that no n-gram receives exactly zero probability while still favoring patterns observed in training.

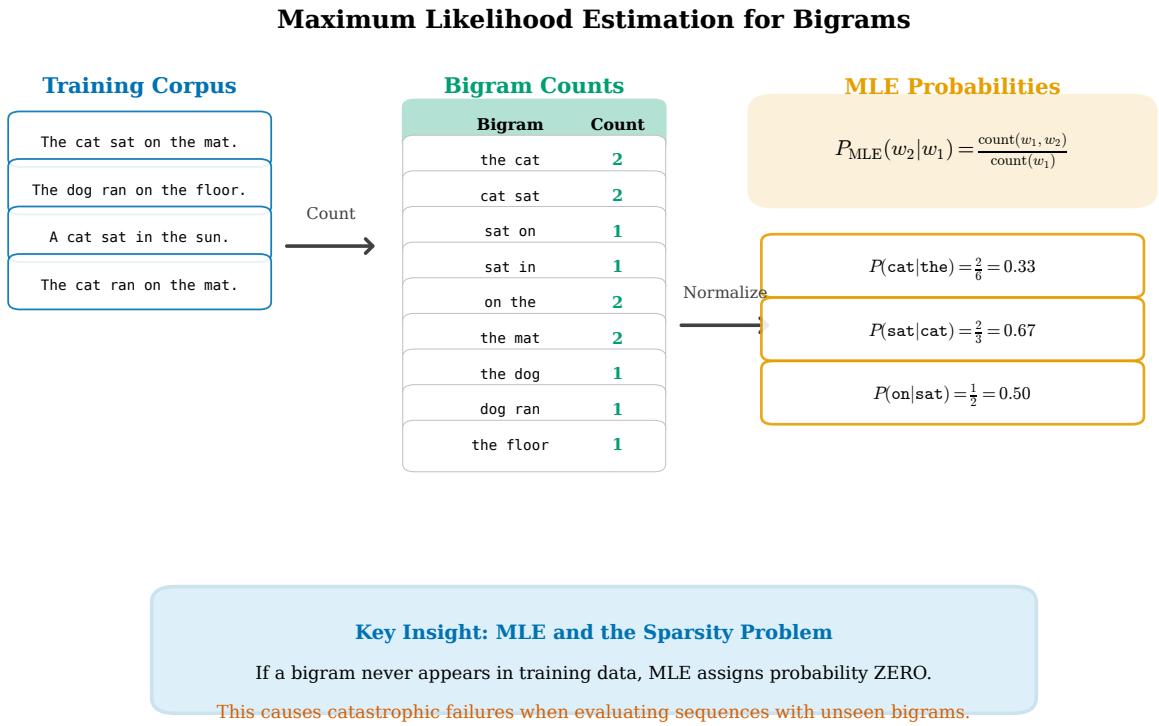


Figure 2.6: Maximum Likelihood Estimation process for bigram probabilities. The training corpus is processed to extract bigram counts, which are then normalized by unigram counts to produce probability estimates. Example calculations show how count ratios yield conditional probabilities. The key insight highlights MLE's critical weakness: unseen bigrams receive zero probability.

2.3 The Sparsity Problem

The fundamental obstacle facing n-gram language models is **sparsity**: the vast majority of possible word sequences never appear in any finite training corpus, yet many of these unseen sequences are perfectly valid and will inevitably occur in test data. This problem is not merely technical but mathematical: the number of possible n-grams grows exponentially with the vocabulary size and n-gram order, while the amount of training data we can collect grows at best polynomially. For a vocabulary of $|\mathcal{V}| = 50{,}000$ words, the number of possible bigrams is $|\mathcal{V}|^2 = 2.5$ billion, the number of possible trigrams is $|\mathcal{V}|^3 = 125$ trillion, and the number of possible 4-grams exceeds 6×10^{18}. Even the largest text corpora containing billions of words cannot hope to observe more than a tiny fraction of these possibilities. The consequence is stark: raw MLE estimates assign zero probability to almost all word sequences, making the model useless for evaluating new text that contains even a single unseen n-gram. Understanding the sparsity problem in depth is essential because the entire edifice of smoothing techniques arose specifically to address this challenge.

Figure 2.7 illustrates the catastrophic nature of zero probabilities with a concrete example. Consider evaluating the perfectly reasonable sentence "The cat sat on the velvet cushion" using a bigram model trained on some corpus. If the bigram "velvet cushion" never appeared in training—perhaps because the corpus lacked interior decorating content—the MLE estimate assigns $P(\text{cushion}|\text{velvet}) = 0$. When we compute the sentence probability using the chain rule, this zero propagates through the product: $P(\text{sentence}) = 0.15 \times 0.25 \times 0.40 \times 0.45 \times 0.12 \times 0.00 \times \ldots = 0$. The model declares the sentence impossible, assigning it the same probability as genuine gibberish like "The the sat velvet on cushion." This behavior is clearly wrong—the sentence is grammatical and meaningful—but it follows inevitably from MLE's reliance on observed counts. The problem worsens with higher-order n-grams: trigram models require seeing specific three-word sequences, 4-gram models require four-word sequences, and so on. The more context we include, the more specific the required

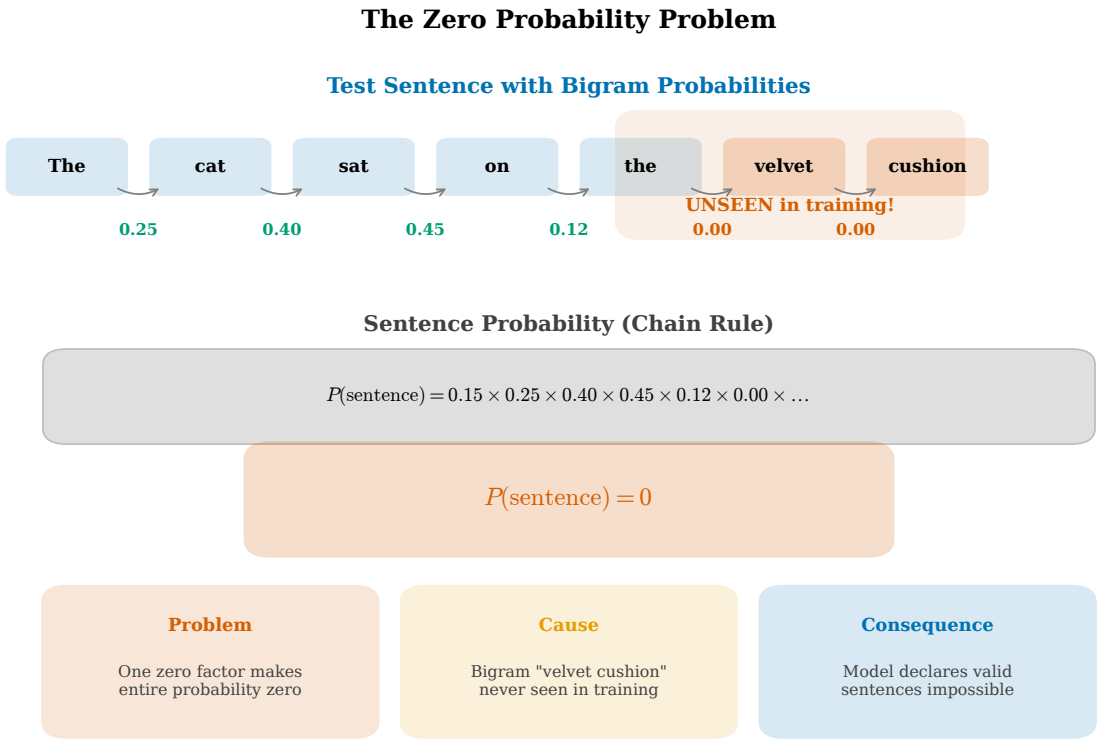


Figure 2.7: The zero probability problem in action. A single unseen bigram ("velvet cushion") causes the entire sentence probability to become zero when using MLE estimation. The chain rule multiplies all bigram probabilities, so one zero factor annihilates the entire product regardless of how well the model predicted other transitions.

observations become, and the less likely we are to have seen exactly what we need.

The mathematical structure of sparsity becomes clear when we examine how the parameter space grows relative to available data. Figure 2.8 quantifies this mismatch between model capacity and data availability. The left panel shows the exponential growth of possible n-grams: with a 50,000-word vocabulary, bigrams number in the billions, trigrams in the trillions, and 4-grams in the quintillions. These numbers dwarf even the largest available corpora. The right panel shows the consequence: as we add more training data, the fraction of possible n-grams we observe increases, but the growth is sublinear—doubling our corpus does not double our coverage. For trigrams and higher orders, even corpora of billions of words observe only a minuscule fraction of the possible parameter space. This creates a fundamental barrier: no amount of data collection can overcome the exponential growth in model parameters. The only solution is to share statistical strength across similar contexts, which is precisely what smoothing techniques accomplish.

The coverage curves in Figure 2.9 reveal the practical implications of sparsity. The left panel shows that adding more training data yields diminishing returns: the curve flattens as corpus size increases, indicating that new text increasingly contains n-grams we have already seen rather than novel ones. This behavior follows from Heaps' law, which states that vocabulary growth is sublinear in corpus size, and extends to n-grams of all orders. The right panel shows the frequency distribution of n-grams, revealing the long tail that characterizes natural language: a small number of common n-grams (the "head") account for most of the tokens in any corpus, while an enormous number of rare n-grams (the "long tail") appear only once or twice each. These singleton and doubleton n-grams, called hapax legomena and dis legomena respectively, pose a particular challenge: their low frequency means our count-based estimates are highly unreliable, yet they collectively account for a substantial portion of the vocabulary and cannot simply be ignored.

Figure 2.10 provides a three-dimensional perspective on how sparsity depends jointly on vocabulary size and n-gram order. The surface rises steeply from the lower-left corner (small vocabulary, bigrams) toward the

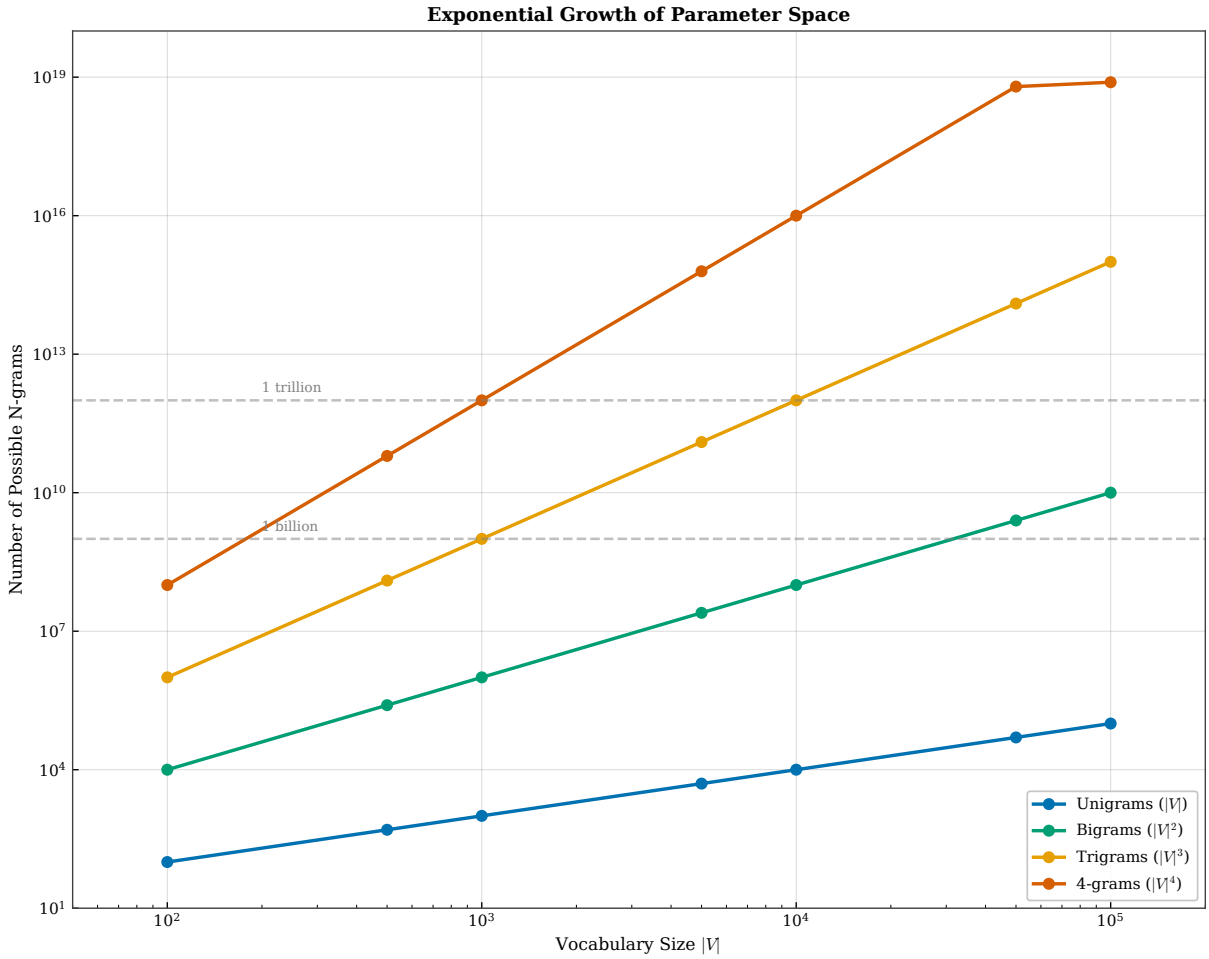


Figure 2.8: The curse of dimensionality in n-gram models. (a) The number of possible n-grams grows exponentially with vocabulary size and n-gram order, quickly exceeding trillions of possibilities. (b) Coverage of possible n-grams increases sublinearly with corpus size, meaning even billions of words observe only a small fraction of the parameter space.

upper-right (large vocabulary, higher-order n-grams), approaching 100% sparsity—meaning almost none of the possible n-grams have been observed. This visualization makes viscerally clear why practitioners rarely use n-grams beyond order 5: the parameter space becomes so vast relative to available data that the additional context provides no benefit. The surface also reveals an important interaction: increasing vocabulary size has a multiplicative effect with n-gram order, so the sparsity penalty for using a larger vocabulary is more severe for higher-order models. This creates pressure to limit vocabulary size through techniques like replacing rare words with a generic unknown token, trading model expressiveness for improved statistical reliability.

The statistical structure underlying sparsity is captured by Zipf's law, visualized in Figure 2.11. George Kingsley Zipf, a Harvard linguist, observed in the 1930s and 1940s that word frequencies in natural language follow a remarkably consistent power law: the frequency of a word is approximately inversely proportional to its rank when words are sorted by decreasing frequency, expressed mathematically as $f(r) \propto r^{-\alpha}$ where r is the rank and the exponent $\alpha \approx 1$ for most natural languages. The most common word in English ("the") appears thousands of times more frequently than words in the long tail of the distribution, dominating any corpus by sheer repetition while rare technical terms may appear only once. The left panel shows this relationship on a log-log scale, where a straight line indicates a power law relationship—the remarkable linearity spanning several orders of magnitude confirms that language statistics follow this pattern with surprising consistency across diverse corpora and genres. The right panel shows the cumulative distribution function: the top 100 most frequent words account for roughly 50% of all tokens in typical text, and a few thousand high-frequency

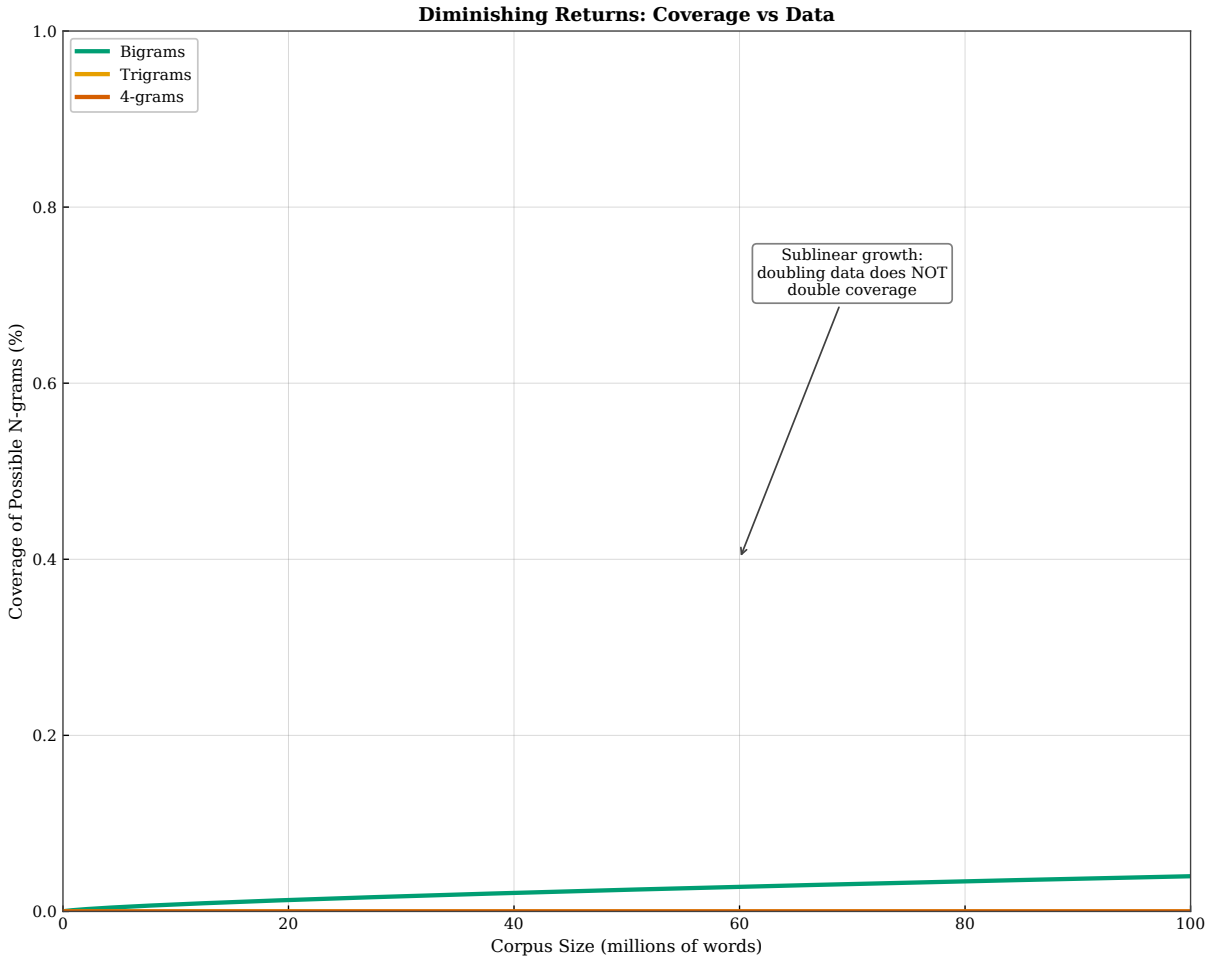


Figure 2.9: N-gram coverage patterns. (a) Coverage of possible n-grams grows sublinearly with corpus size, exhibiting severe diminishing returns. (b) The long-tail distribution of n-gram frequencies: a small number of n-grams account for most tokens, while the vast majority are rare, appearing only once or twice in even large corpora.

words cover 80-90% of the tokens we encounter in everyday reading. Yet that remaining 10-20% of tokens comprises tens of thousands of rare words, each essential for expressing specific technical concepts, proper names, and nuanced distinctions. This heavy-tailed distribution explains why n-gram sparsity is so mathematically severe: if individual words are rare, their combinations are exponentially rarer still, following the product of their individual probabilities. A bigram involving two rare words inherits the rarity of both components, and trigrams compound this effect cubically rather than linearly as each additional word multiplies the sparsity. The Zipfian distribution is not a peculiarity of English but appears across all documented human languages and even constructed languages like Esperanto, strongly suggesting it emerges from fundamental properties of efficient communication systems that balance speaker effort (favoring reuse of common words) against listener comprehension (requiring precision through specialized vocabulary for specific meanings).

The sparsity problem motivates the entire field of smoothing, which we develop in the following section as the central technical contribution of n-gram language modeling research. The key insight is that while we cannot observe all possible n-grams in any finite corpus, we can use what we have observed to make educated guesses about what we have not observed. If we have seen “the blue car” and “the red car” but never “the green car,” we might reasonably infer that “the green car” is plausible by analogy—colors are interchangeable in this construction, suggesting that any color could follow “the” before “car.” Smoothing techniques formalize this intuition in mathematical terms, redistributing probability mass from observed events to similar unobserved events according to various principles. The simplest approaches add small pseudo-

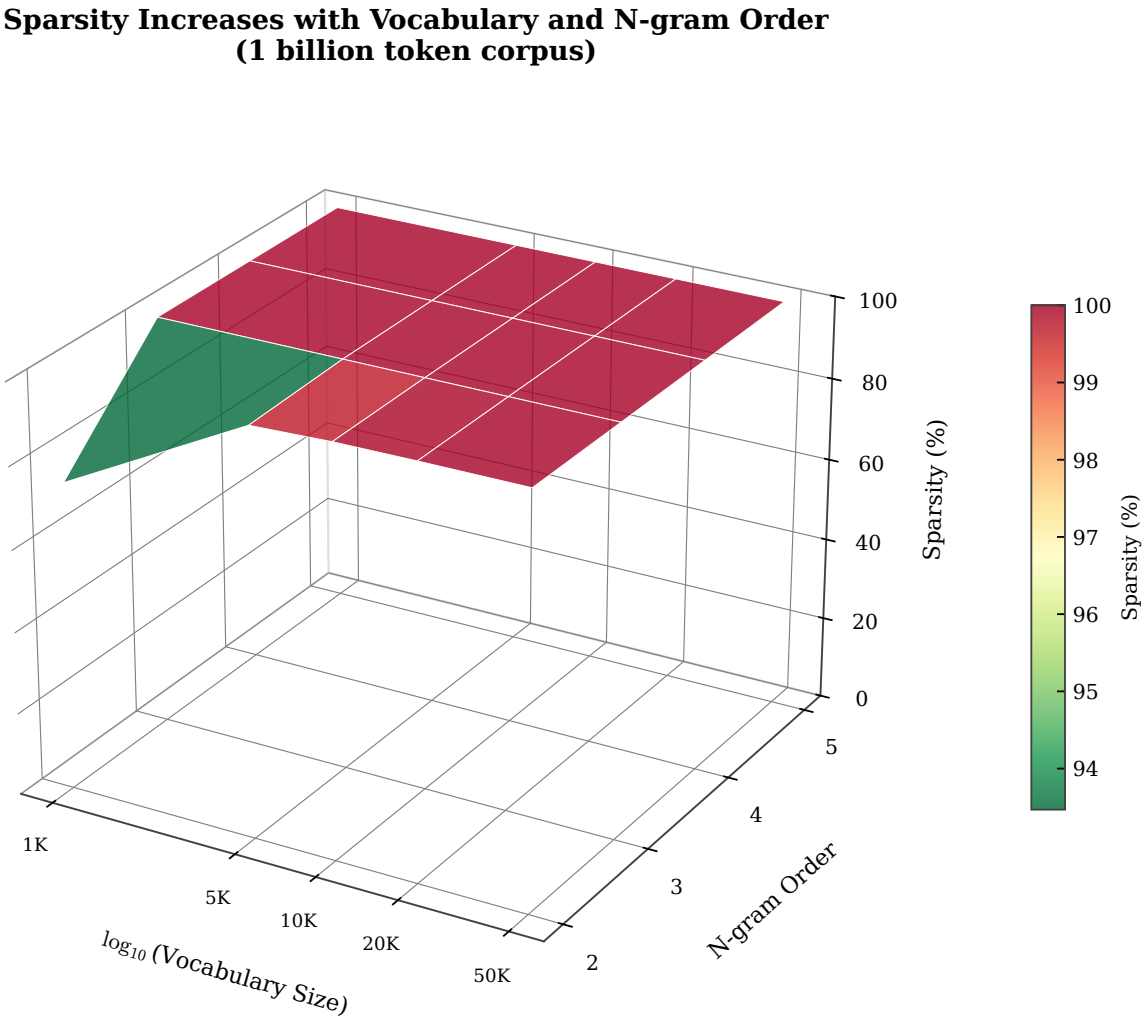


Figure 2.10: 3D visualization of sparsity as a function of vocabulary size and n-gram order. The surface rises steeply toward 100% sparsity as either dimension increases. Even with a billion-token corpus, higher-order n-grams with realistic vocabularies exhibit near-total sparsity, with almost all possible sequences unobserved.

counts to all n-grams regardless of whether they were observed (Laplace smoothing), treating all unseen events as equally likely. More sophisticated methods like Good-Turing estimation use the statistics of rare events to estimate how much mass should go to unseen events, while Kneser-Ney smoothing incorporates information about word versatility to distribute mass more intelligently. The common thread across all these methods is accepting that zero counts do not mean zero probability—they simply mean we lack direct evidence and must rely on indirect statistical inference to estimate what we would likely see if we had more data.

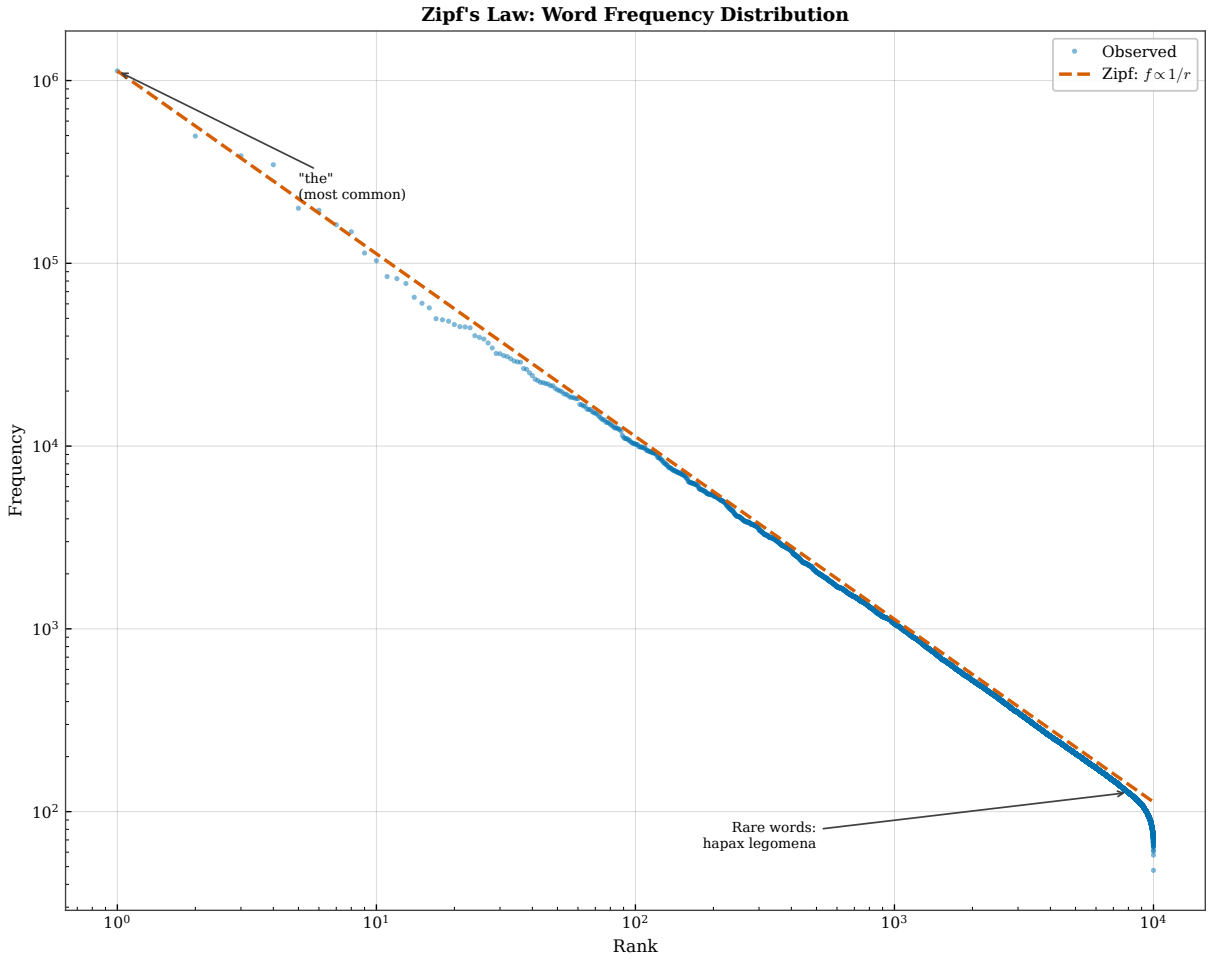


Figure 2.11: Zipf's law and its implications for n-gram sparsity. (a) Word frequencies follow a power law: the most common words are vastly more frequent than the rest. (b) Cumulative coverage shows that a small number of high-frequency words account for most tokens, but the long tail of rare words is essential for complete coverage.

2.4 Smoothing Techniques

Smoothing refers to any technique that assigns non-zero probability to events not observed in training data by redistributing probability mass from observed events. The fundamental principle is simple: we cannot take probability from nowhere, so to give probability to unseen n-grams, we must take some away from seen n-grams. Different smoothing methods embody different philosophies about how much to take, from whom to take it, and how to distribute what is collected. The simplest methods apply uniform adjustments to all n-grams regardless of their frequency, while sophisticated methods tailor their adjustments based on the statistical properties of the training corpus. The development of smoothing techniques spans several decades and represents some of the most elegant applied statistics in natural language processing. In this section, we survey the major approaches, building from the intuitive but flawed Laplace smoothing through the highly effective Kneser-Ney method that remains the gold standard for n-gram language models.

Laplace smoothing, also called add-one smoothing, is the oldest and simplest approach: add one to every count before computing probabilities. If we observe counts $c(w_1, w_2)$ for bigrams, the Laplace-smoothed probability becomes $P_{\text{Laplace}}(w_2|w_1) = \frac{c(w_1,w_2)+1}{c(w_1)+|\mathcal{V}|}$, where the denominator adds the vocabulary size $|\mathcal{V}|$ to maintain proper normalization. Figure 2.12 illustrates this process: a word that appeared 45 times now has count 46, while a never-seen word receives count 1. The formula guarantees that every n-gram receives non-zero probability, solving the immediate problem of zero probabilities. However, Laplace smoothing suffers from

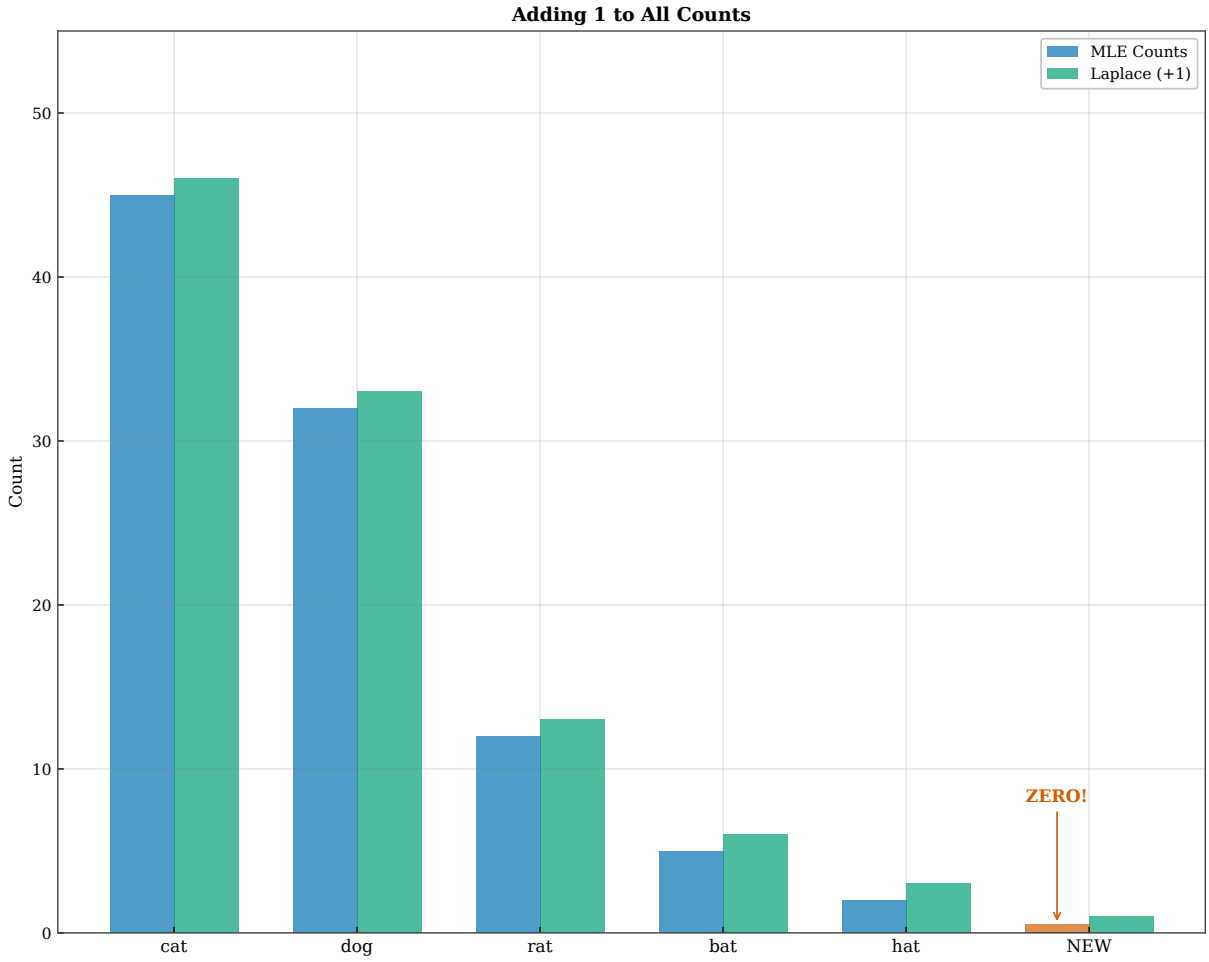


Figure 2.12: Laplace (add-one) smoothing. The left panel shows how adding one to all counts ensures no zero values remain. The right panel compares MLE and Laplace probabilities, demonstrating how previously unseen events (marked "NEW") receive non-zero probability while observed events are slightly discounted.

a severe flaw: it redistributes far too much probability mass to unseen events. With a vocabulary of 50,000 words, adding one to each of 50,000 bigrams following a given word dramatically inflates the denominator, shrinking the probabilities of observed bigrams to near insignificance. The method works adequately for small vocabularies but fails catastrophically for realistic language modeling where vocabularies are large and most n-grams remain unseen.

Add-k smoothing generalizes Laplace by adding a fractional count $k < 1$ instead of one: $P_{\text{add-k}}(w_2|w_1) = \frac{c(w_1,w_2)+k}{c(w_1)+k\cdot|\mathcal{V}|}$. Figure 2.13 shows how different values of k affect the probability distribution. Small values like $k = 0.01$ preserve more of the original distribution's shape, taking less from observed events and giving less to unseen ones. The right panel reveals that optimal k depends on corpus size: with abundant data, less smoothing is needed because MLE estimates are already reliable; with sparse data, more aggressive smoothing prevents overfitting to noise. In practice, k is tuned on held-out data to minimize perplexity, treating it as a hyperparameter rather than a principled choice. While add-k smoothing improves over Laplace, it still applies a uniform adjustment regardless of how often an n-gram appeared. A bigram seen 1,000 times is discounted by the same amount as one seen twice, which seems wasteful—surely we should discount less from events we have strong evidence for and more from events with shaky evidence.

Good-Turing estimation, developed by Alan Turing during World War II for cryptanalysis, offers a more principled approach based on the statistics of the training corpus itself. The key insight is to use the frequency of frequencies: let N_r denote the number of n-grams that appear exactly r times in training. Figure 2.14 shows a typical N_r distribution, which follows a steep power law—many n-grams appear once or twice, few appear

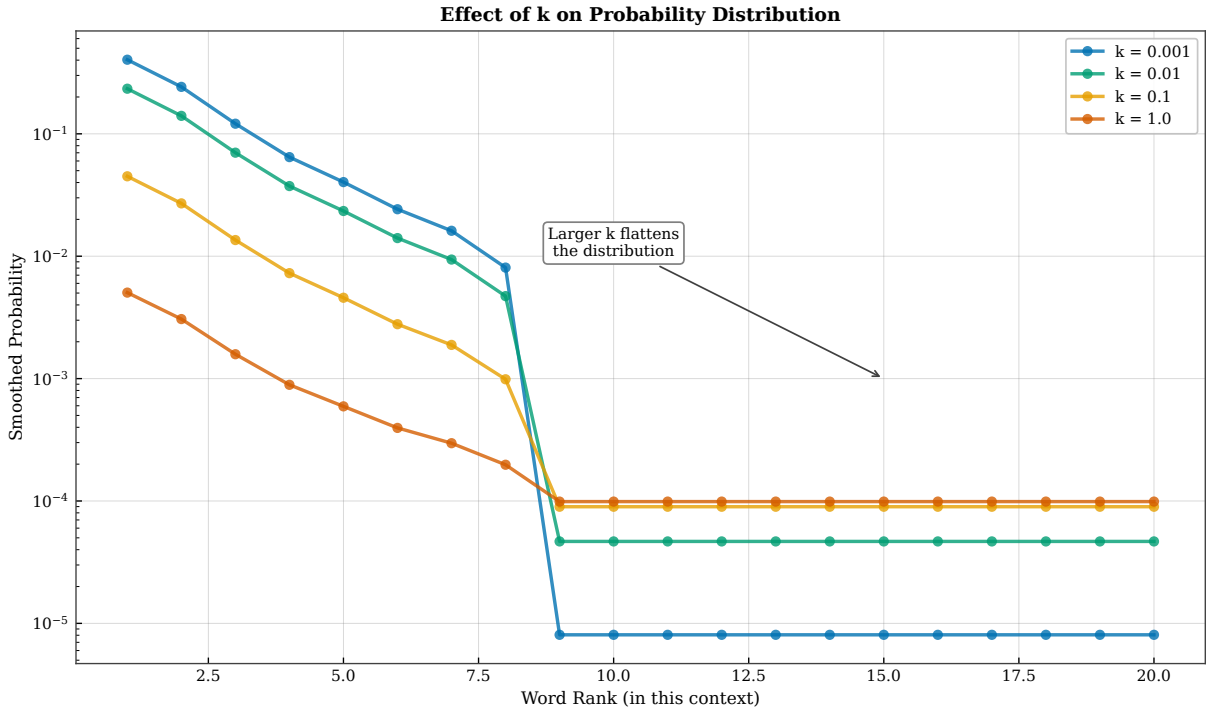


Figure 2.13: Add-k smoothing with different values of k. (a) Larger k values flatten the probability distribution, reducing the gap between frequent and rare words. (b) The optimal k value varies with corpus size: smaller corpora benefit from more smoothing, while larger corpora require less intervention.

thousands of times. Good-Turing proposes adjusting counts according to $c^* = (c+1) \cdot N_{c+1}/N_c$: the adjusted count for items seen c times equals $(c+1)$ multiplied by the ratio of items seen $c+1$ times to items seen c times. This formula has elegant theoretical justification: it estimates what we would expect to observe if we drew a new sample of the same size, accounting for the fact that low-frequency items are more likely to be accidents of sampling than high-frequency items. The total probability mass assigned to unseen events equals N_1/N, where N is the total token count—a quantity entirely determined by how many singletons we observed.

Figure 2.15 visualizes the fundamental trade-off in all smoothing methods: probability is a finite resource that must be divided between observed and unobserved events. MLE dedicates 100% of probability mass to events seen in training, leaving nothing for anything else. Smoothing carves out a portion—say, 15%—for the unseen class, distributing it among all possible n-grams not observed in training. The question every smoothing method must answer is: how much mass should we reserve, and how should we distribute it? Simple methods like add-k use uniform distribution, giving every unseen n-gram equal probability. More sophisticated methods recognize that not all unseen n-grams are equally plausible: "the cat" is more likely than "the the" even if neither appeared in training, because we can infer plausibility from the behavior of individual words in other contexts. This insight leads to the backing-off and interpolation approaches we explore in the next section.

Absolute discounting takes a direct and elegant approach: subtract a fixed amount d (typically around 0.75) from every non-zero count, then redistribute the collected probability mass to unseen events. Figure 2.16 illustrates the mechanism in detail: a bigram with count 100 becomes 99.25, while one with count 5 becomes 4.25, and a singleton with count 1 becomes 0.25. The collected mass is then redistributed to unseen events using a lower-order model, typically a unigram distribution or a smoothed lower-order n-gram. The complete formula becomes $P_{\text{abs}}(w_2|w_1) = \frac{\max(c(w_1,w_2)-d,0)}{c(w_1)} + \lambda(w_1) \cdot P_{\text{lower}}(w_2)$, where $\lambda(w_1)$ is a normalizing factor computed to ensure probabilities sum to one. The intuition behind absolute discounting is that high-count n-grams can afford to lose a fixed amount without much damage to their probability estimates—subtracting 0.75 from a count of 100 changes the estimated probability by less than one percent—while the mass collected from many such small subtractions accumulates into a substantial reserve for unseen events. The optimal discount value d can be derived theoretically using Good-Turing frequency analysis rather than being set arbitrarily: it turns out that

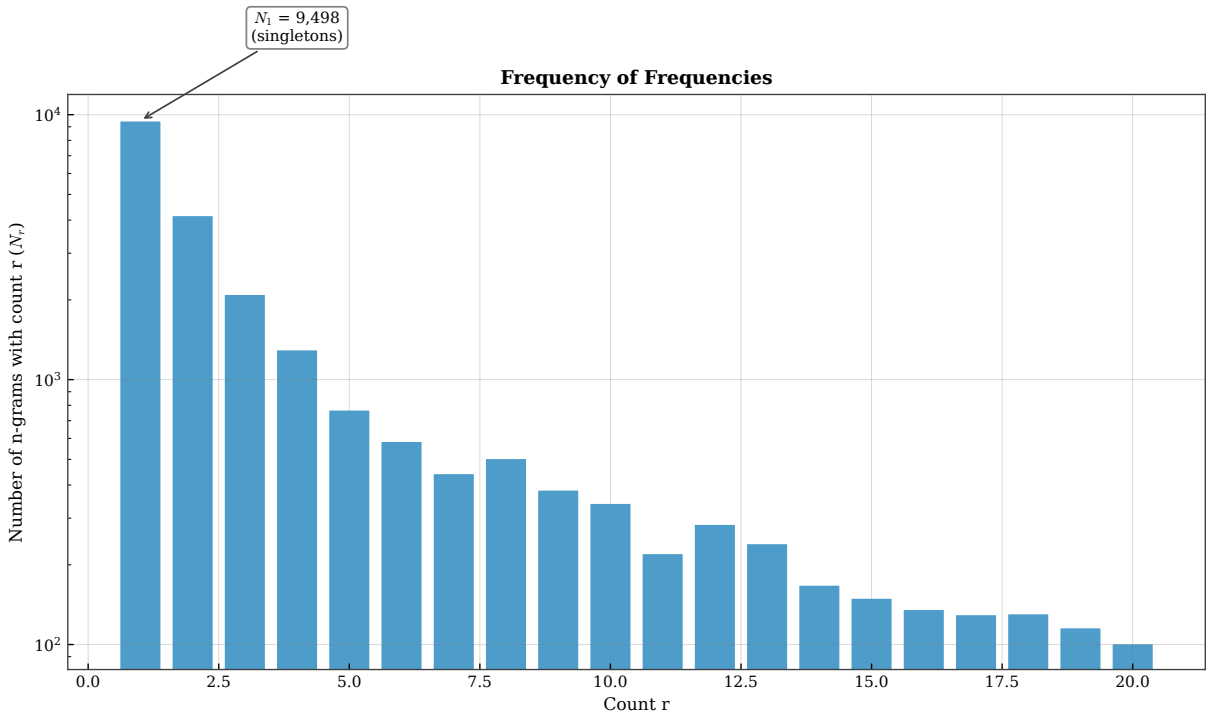


Figure 2.14: Good-Turing estimation. (a) The frequency of frequencies N_r counts how many n-grams appear exactly r times. (b) Good-Turing adjusts counts using the formula $c^* = (c+1) \cdot N_{c+1}/N_c$, systematically reducing observed counts to free mass for unseen events. The shaded region shows mass transferred to unseen n-grams.

$d \approx \frac{N_1}{N_1+2N_2}$, where N_1 and N_2 are the counts of singletons and doubletons respectively, typically yielding values between 0.7 and 0.9 for natural language corpora.

Kneser-Ney smoothing represents the culmination of n-gram smoothing research and remains the best-performing method for statistical language models. Its key innovation addresses a subtle problem with standard backoff: when we back off from an unseen trigram to a bigram, we should not use the raw frequency of the continuation word, because raw frequency conflates versatile words with specialized ones. Figure 2.17 illustrates with a famous example: "Francisco" might appear 10,000 times in a corpus, but almost always following "San." Meanwhile, "glasses" might appear only 1,000 times, but after many different words: "wine glasses," "reading glasses," "sun glasses," and so on. If we encounter an unseen context like "reading ___" and back off to unigram probabilities, raw frequency would favor "Francisco" over "glasses"—clearly wrong, since "reading Francisco" makes no sense. Kneser-Ney solves this by using **continuation probability**: instead of counting how often a word appears, we count in how many different contexts it appears. The continuation probability $P_{\text{cont}}(w)$ is proportional to the number of unique words that precede w in training, capturing the word's versatility rather than its frequency.

The full Kneser-Ney formula combines absolute discounting with continuation probability in an elegant recursive structure: for the highest-order n-gram, we use discounted MLE based on actual counts, and for backoff distributions, we use continuation counts that capture word versatility rather than raw frequency. For bigrams, the formula becomes: $P_{\text{KN}}(w_2|w_1) = \frac{\max(c(w_1,w_2)-d,0)}{c(w_1)} + \lambda(w_1) \cdot P_{\text{cont}}(w_2)$, where the continuation probability $P_{\text{cont}}(w_2) = \frac{|\{v:c(v,w_2)>0\}|}{|\{(v,w):c(v,w)>0\}|}$ counts the number of unique predecessor words that appear before w_2 in the training data, divided by the total number of unique bigram types observed. This elegant formulation captures the profound intuition that words appearing in diverse contexts are more likely to be appropriate continuations for novel contexts than words locked into specific fixed phrases, regardless of how frequently those fixed phrases occur. Modified Kneser-Ney, developed by Chen and Goodman in the late 1990s, further improves performance by using three different discount values d_1, d_2, and d_{3+} for n-grams appearing exactly once, exactly twice, and

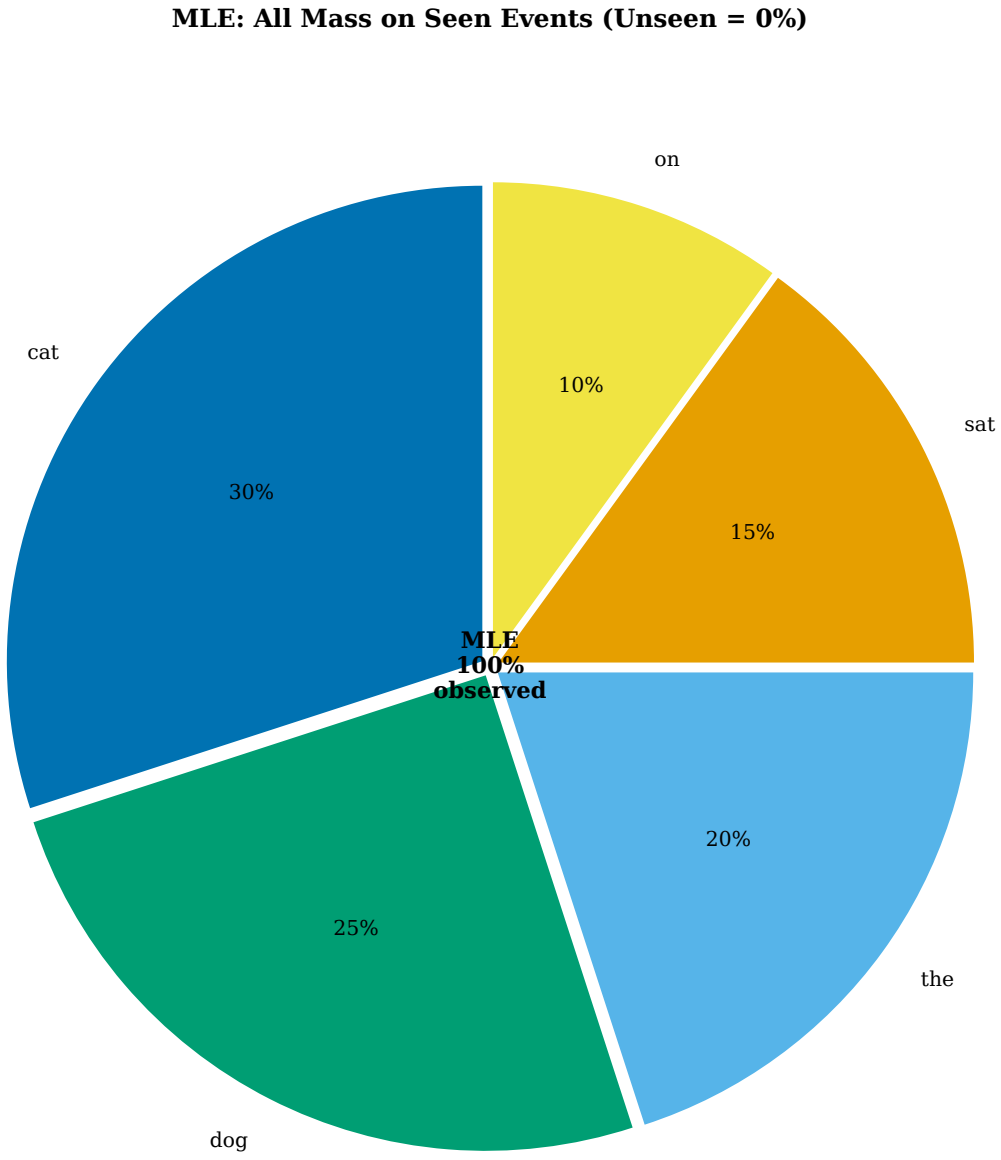


Figure 2.15: Probability mass redistribution in smoothing. MLE assigns 100% of probability mass to observed events, leaving nothing for unseen words. Smoothing reserves a portion (here 15%) for the unseen class, enabling the model to handle novel inputs while still preferring observed patterns.

three or more times respectively, recognizing that the optimal discount differs based on how much evidence we have for each n-gram. The method extends naturally to higher-order n-grams by recursively applying the same continuation probability principle at each backoff level, creating a hierarchy where each level contributes its unique statistical evidence to the final probability estimate.

Figure 2.18 compares smoothing methods empirically on standard benchmark datasets, revealing the substantial practical differences between approaches. The left panel shows perplexity scores for various methods on held-out test data: no smoothing gives infinite perplexity because any single zero probability in the chain rule product causes the entire computation to fail, Laplace smoothing produces poor results by redistributing too much mass uniformly, add-k improves substantially by allowing tuning of the smoothing strength, Good-Turing performs well by using corpus statistics to guide redistribution, and Kneser-Ney achieves the best performance

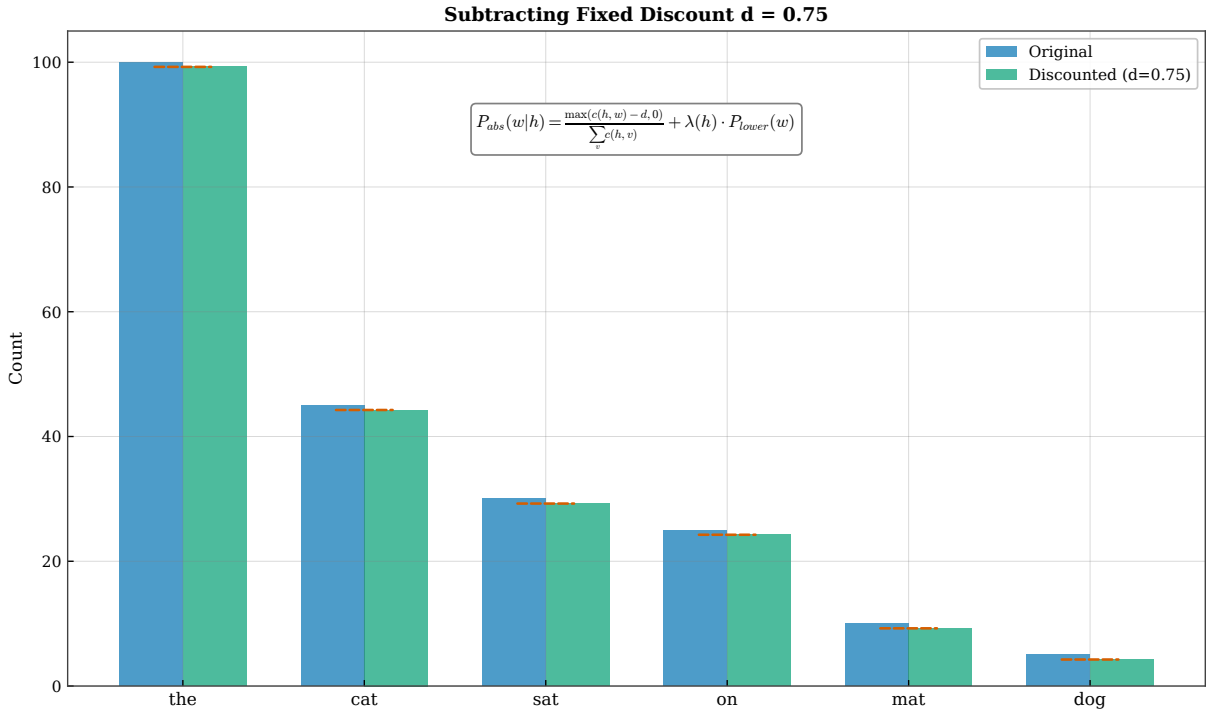


Figure 2.16: Absolute discounting. (a) A fixed discount d (typically 0.75) is subtracted from each non-zero count. (b) The total mass collected depends on d and the count distribution, with higher d values extracting more mass for redistribution to unseen events.

by incorporating word versatility information. The right panel reveals how different methods scale with n-gram order: simple methods like Laplace deteriorate rapidly as n increases because they cannot cope with the exponentially growing sparsity that characterizes higher-order models, while Kneser-Ney maintains reasonable and relatively stable performance even for 5-grams by effectively sharing statistical strength across related contexts. This robustness explains why Kneser-Ney became the standard choice for n-gram language models in demanding applications like speech recognition and machine translation, where higher-order models are essential to capture phrase-level patterns and idiomatic expressions. The development from Laplace through Good-Turing to Kneser-Ney represents an intellectual journey from naive uniformity to sophisticated statistical inference, progressively learning to use the inherent structure of natural language to inform how probability mass should be distributed among unseen events.

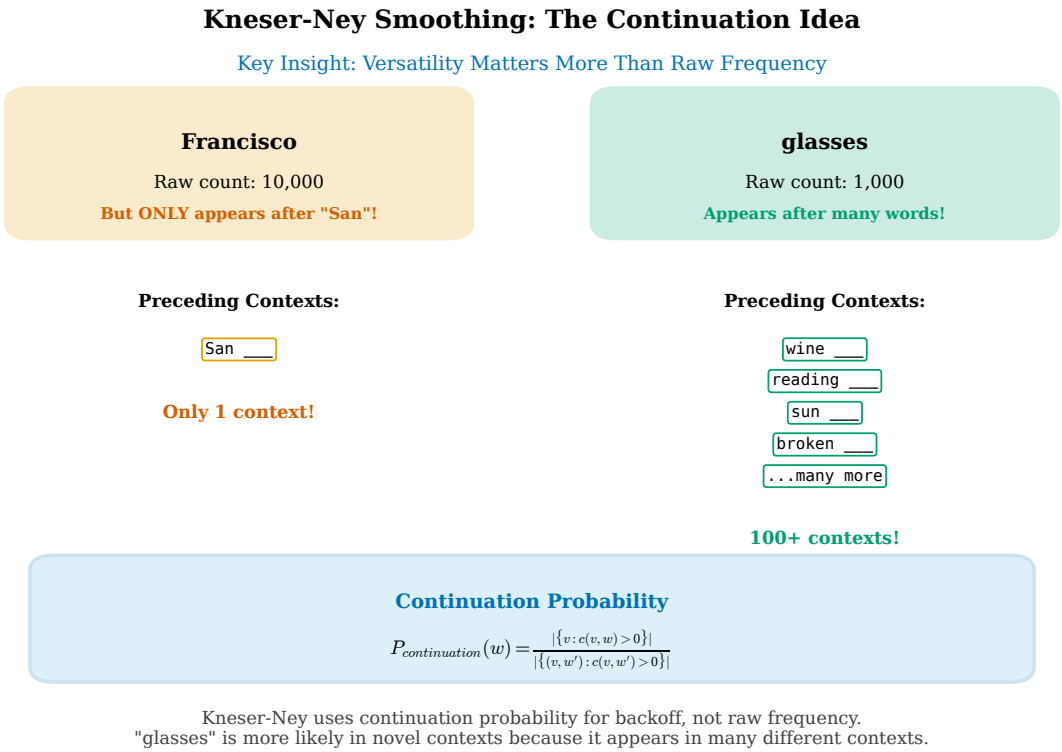


Figure 2.17: The key insight of Kneser-Ney smoothing: continuation probability. “Francisco” has high raw frequency but appears only after “San,” making it a poor choice for novel contexts. “glasses” has lower frequency but appears after many different words, making it more versatile. Kneser-Ney uses continuation probability rather than raw frequency for backing off.

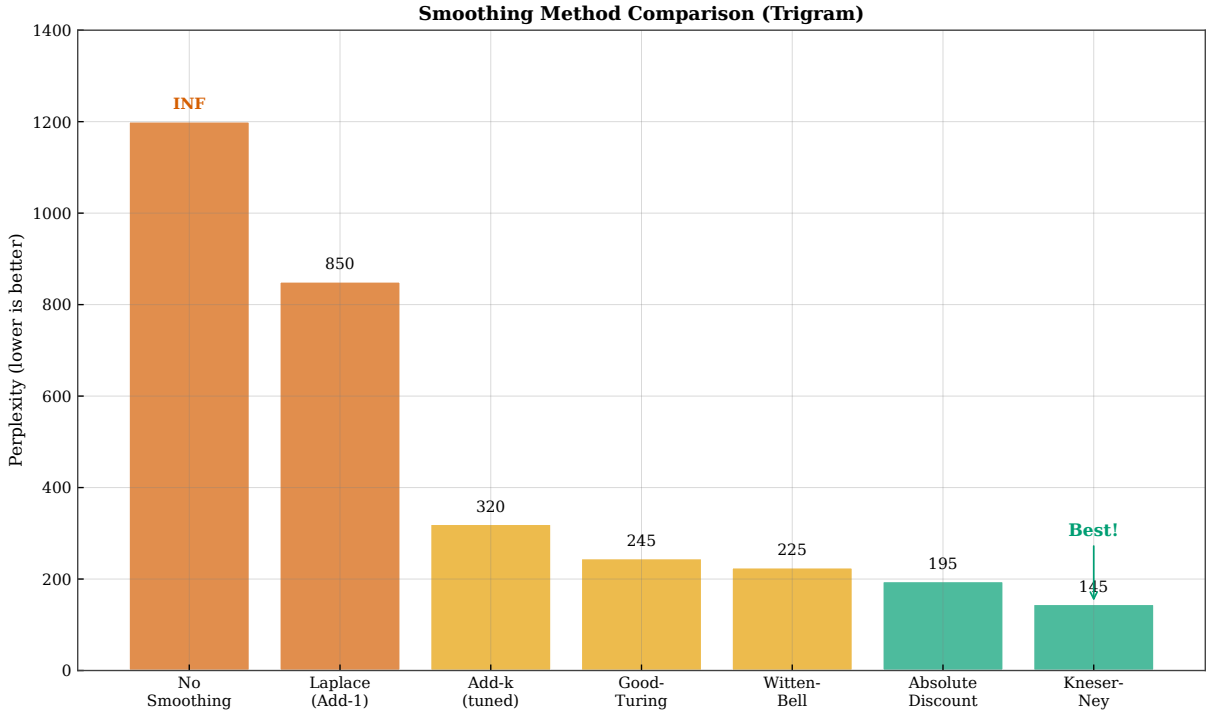


Figure 2.18: Comparison of smoothing methods. (a) Perplexity comparison for trigram models: Kneser-Ney achieves the lowest perplexity, while no smoothing yields infinite perplexity due to zero probabilities. (b) Performance across n-gram orders: Kneser-Ney maintains low perplexity even for higher-order models where other methods deteriorate.

2.5 Backoff and Interpolation

Smoothing techniques address the zero-probability problem by ensuring that no n-gram receives exactly zero probability, but they leave open an important architectural question: how should we combine information from different n-gram orders when making predictions? A trigram model conditions on two preceding words, capturing phrase-level patterns, but what if that specific two-word context is rare or completely unseen in the training corpus? We could use the trigram probability if sufficient evidence is available, but fall back to more reliable bigram or unigram probabilities when the trigram evidence is sparse or absent. This intuition leads to two complementary strategies for combining n-gram orders: **backoff**, which uses higher-order estimates when possible and falls back to lower orders only when direct evidence is lacking, and **interpolation**, which always combines estimates from all available orders using learned weights that reflect each order's reliability. Both strategies exploit a crucial insight about the bias-variance trade-off in statistical estimation: lower-order models have worse predictive accuracy because they ignore relevant context but enjoy better coverage because their simpler patterns appear more frequently, while higher-order models have better potential accuracy because they consider more context but suffer more severely from sparsity because their complex patterns rarely repeat. The art of n-gram modeling lies in balancing these competing trade-offs to achieve the best overall performance on held-out data.

Figure 2.19 illustrates the backoff strategy with a concrete example. To estimate $P(\text{cushion}|\text{velvet})$, we first check whether the bigram "velvet cushion" appeared in training. If the count is positive, we use the (possibly discounted) MLE estimate. If the count is zero, we back off to the unigram probability $P(\text{cushion})$, multiplied by a normalizing factor $\alpha(\text{velvet})$ that ensures the conditional distribution sums to one. The factor α is computed to redistribute exactly the probability mass that was discounted from observed bigrams, maintaining proper normalization. For trigram models, we might back off twice: first from the unseen trigram to a bigram, then potentially from an unseen bigram to a unigram. The hierarchy continues for higher-order models, always falling back to more general estimates when specific evidence is lacking. The key principle is that we only back off when forced to by zero counts; when we have evidence at the higher order, we trust it.

Interpolation takes a fundamentally different approach: rather than choosing between n-gram orders based on whether counts are zero or positive, we always combine estimates from all available orders using learned weights, creating a weighted average that benefits from each order's strengths. Figure 2.20 shows the linear interpolation formula: $P_{\text{interp}}(w|\text{context}) = \lambda_3 \cdot P_3(w|w_{-2}, w_{-1}) + \lambda_2 \cdot P_2(w|w_{-1}) + \lambda_1 \cdot P_1(w)$, where the interpolation weights λ_i are non-negative and sum to one, ensuring the result is a valid probability distribution. The intuition is that even when we have trigram evidence from direct observation, the bigram and unigram estimates provide useful complementary information that can regularize our predictions, especially when the trigram count is low and therefore unreliable as a probability estimate. The weights λ_i can be global constants learned on held-out development data to minimize perplexity, or they can vary dynamically by context, with rare contexts receiving higher weights on lower-order models whose estimates are more reliable and frequent contexts trusting higher-order estimates that have sufficient data support. Context-dependent interpolation, shown in the right panel of the figure, adapts the weighting to the statistical reliability of each specific context: a trigram context observed 1,000 times warrants high trigram weight because we have strong evidence, while a context seen only twice should rely more heavily on the robust lower-order estimates that draw from much larger pools of evidence.

When corpora become truly massive—billions or trillions of words from web crawls—the sophisticated machinery of Kneser-Ney smoothing becomes less necessary because raw counts are more reliable. **Stupid backoff**, introduced by researchers at Google, exploits this observation with a radically simple approach: use relative frequency when the n-gram count is positive, otherwise multiply by a fixed factor $\alpha = 0.4$ and back off to the next lower order. Figure 2.21 shows the algorithm and its performance characteristics. The method produces "scores" rather than probabilities because the factors do not normalize properly, but for applications like ranking candidate translations or speech recognition hypotheses, relative scores suffice. The right panel reveals a striking finding: at web scale (billions of tokens), stupid backoff matches or exceeds Kneser-Ney performance despite its simplicity. This happens because with enough data, even 5-grams become reasonably well-estimated, reducing the need for careful probability redistribution. The lesson is that the optimal complexity of a smoothing method depends on the data regime: sophisticated methods shine with limited data, while

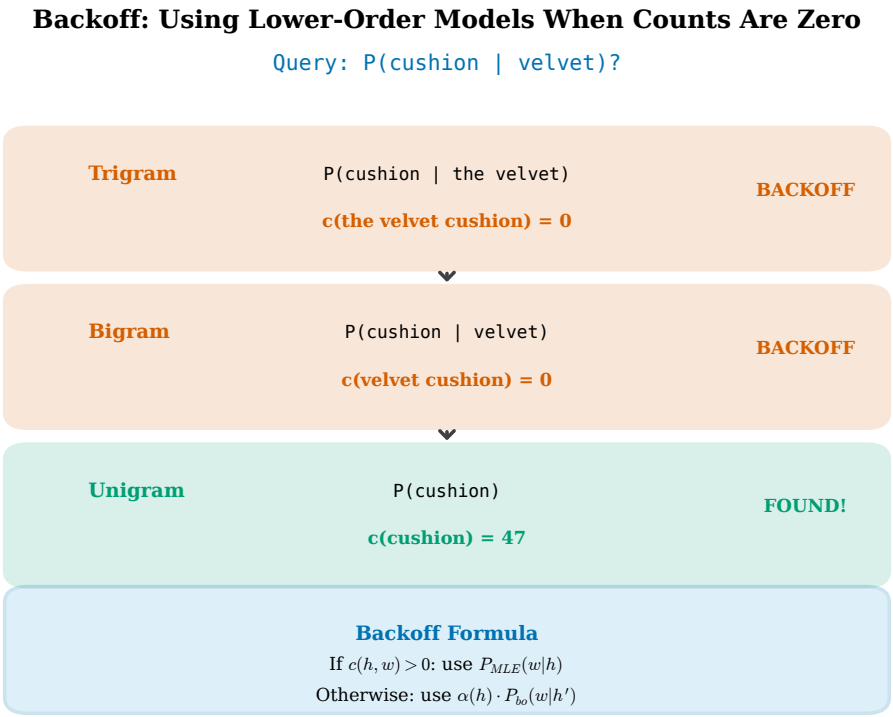


Figure 2.19: The backoff strategy for handling unseen n-grams. When estimating $P(\text{cushion}|\text{velvet})$, we first check if the bigram was observed. If the count is zero, we back off to the unigram probability, applying a normalization factor α to maintain a valid probability distribution.

simple methods suffice when data is abundant.

Figure 2.22 provides a three-dimensional view of the backoff hierarchy, visualizing the fundamental trade-off between specificity and coverage. At the top, 5-gram models make highly specific predictions conditioned on four preceding words, but the pyramid narrows because only a tiny fraction of possible 5-grams appear in any corpus. Moving down, each level broadens: 4-grams, trigrams, bigrams, and finally unigrams, which cover the entire vocabulary but ignore context entirely. The arrows show the backoff direction: when a higher-order count is zero, we descend to more general estimates. This hierarchy embodies a deep principle: we should use the most specific model for which we have reliable evidence, falling back to generality only when specificity fails. The same principle reappears in neural language models, where attention mechanisms learn to focus on relevant context while ignoring irrelevant history, and in hierarchical architectures where lower layers capture local patterns and higher layers integrate global context.

The choice between backoff and interpolation, and the specific smoothing method used at each level of the n-gram hierarchy, depends on the application requirements and available computational resources. For real-time applications like speech recognition and predictive text input, computational efficiency matters greatly, favoring simpler methods that can be implemented with efficient data structures like tries, minimal perfect hash functions, and compressed suffix arrays that enable constant-time lookups. For offline applications like corpus analysis, machine translation training, or research benchmarking, more sophisticated methods with higher computational overhead can be afforded because latency is less critical than accuracy. Modified Kneser-Ney with interpolation remains the gold standard for moderate-sized corpora in the millions to billions of tokens, providing the best balance of accuracy and reasonable computational requirements. Stupid backoff dominates at web scale with trillions of tokens, where raw data abundance compensates for algorithmic simplicity. The common thread across all these methods is the recognition that language modeling is fundamentally a problem of trading off specificity against reliability: we want to use the most specific context available while gracefully

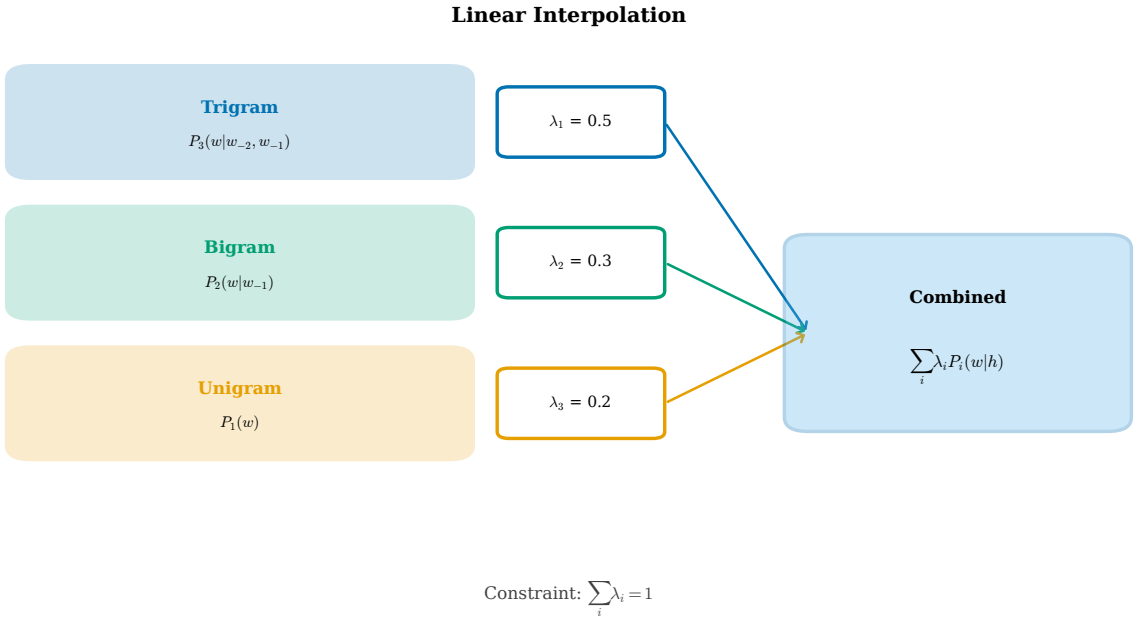


Figure 2.20: Interpolation combines estimates from all n-gram orders using learned weights. (a) The linear interpolation formula weights trigram, bigram, and unigram probabilities. (b) Context-dependent weights allow the model to rely more on lower-order estimates for rare contexts where higher-order statistics are unreliable.

degrading to more general estimates when specific evidence is sparse or absent. This fundamental trade-off, first encountered and systematically studied in n-gram models, will recur throughout our exploration of neural language models in subsequent chapters, where analogous decisions about model capacity, regularization strength, and effective context length reflect the same underlying tension between fitting the training data and generalizing to new inputs.

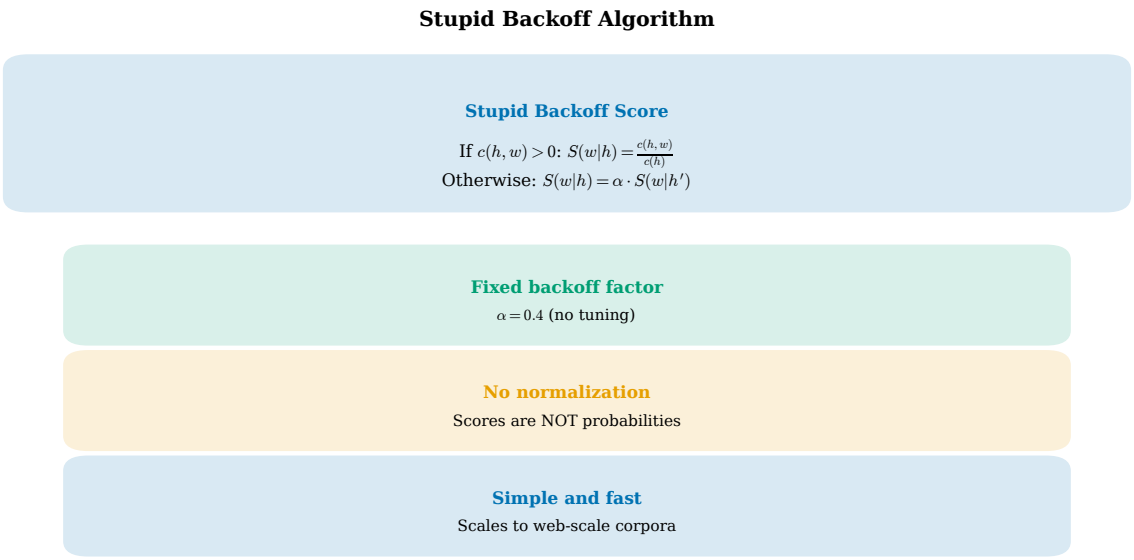


Figure 2.21: Stupid backoff: a simple method that scales to web-size corpora. The algorithm uses a fixed backoff factor $\alpha = 0.4$ without normalization, producing scores rather than probabilities. At massive scale, stupid backoff matches the performance of more sophisticated methods like Kneser-Ney while being simpler and faster.

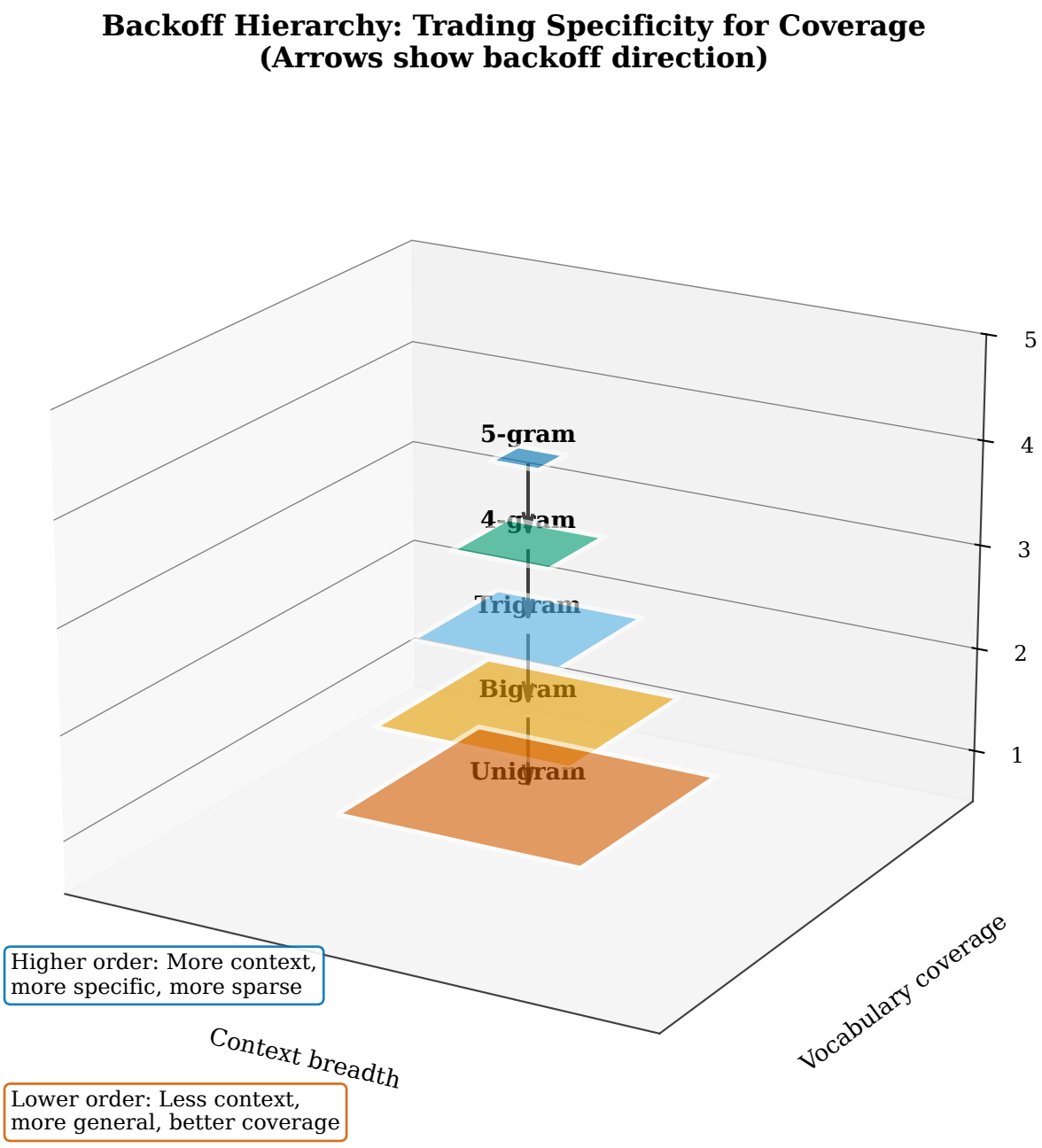


Figure 2.22: 3D visualization of the backoff hierarchy. Higher-order n-grams (top) provide more specific predictions but cover fewer contexts. Lower-order n-grams (bottom) cover more contexts but make less specific predictions. Arrows indicate the backoff direction when higher-order counts are zero.

2.6 Evaluation and Limitations

Having developed the complete machinery of n-gram language models—from the Markov assumption that makes estimation tractable, through maximum likelihood estimation that converts counts to probabilities, smoothing techniques that handle the sparsity problem, and backoff strategies that combine evidence from multiple n-gram orders—we now turn to the critical questions of evaluating model quality and understanding the fundamental limitations of this approach. The standard metric for language model evaluation is **perplexity**, an information-theoretic measure that quantifies how well a probability model predicts a held-out test set that was not used during training. Lower perplexity indicates a better model: a model that assigns higher probability to the test data is less "perplexed" or surprised by what it sees, suggesting it has learned the underlying statistical patterns of the language. Understanding perplexity deeply requires connecting language modeling to Shannon's foundational information theory, revealing that evaluating language models is fundamentally about measuring how many bits are needed to encode text under the model's probability distribution—a model that predicts well requires fewer bits because it anticipates what comes next. Beyond establishing evaluation metrics, we must also confront the inherent limitations of n-gram models that no amount of clever engineering can overcome, limitations that ultimately motivate the neural approaches developed in subsequent chapters.

Perplexity is defined mathematically as the inverse probability of the test set, normalized by the number of words to enable comparison across texts of different lengths: $PP(W) = P(w_1, w_2, \ldots, w_N)^{-1/N}$. Figure 2.23 illustrates this formula and provides intuition for its interpretation. A perplexity of 100 means the model is, on average, as uncertain about the next word as if it were choosing uniformly among 100 equally likely candidates—the model's probability distribution has an effective support size of 100 words at each position. The geometric mean formulation ensures that perplexity reflects average uncertainty per word rather than total uncertainty over the entire test set, making the metric comparable across corpora of different sizes. This normalization also means that perplexity is multiplicative across independent choices: if we are equally uncertain at each position, the per-word perplexity captures this regardless of sequence length. For comparison across the history of language modeling: a random model choosing uniformly from a 50,000-word vocabulary has perplexity 50,000 by definition; a unigram model that predicts based on word frequency alone might achieve perplexity around 1,000 by learning that common words are more likely; a well-tuned trigram model reaches 100-200 by capturing local sequential dependencies; and state-of-the-art neural models achieve perplexities below 50 on standard benchmarks by learning long-range patterns and semantic relationships. The right panel of the figure shows this progression, illustrating how each advance in modeling sophistication yields measurable and substantial improvement in perplexity.

Perplexity connects directly to **cross-entropy**, a fundamental concept in information theory. The cross-entropy of a language with respect to a model is $H(L,M) = -\frac{1}{N}\sum_{i=1}^{N} \log_2 P_M(w_i|w_{<i})$, measuring the average number of bits needed to encode each word under the model's distribution. Figure 2.24 shows that perplexity is simply $2^{H(L,M)}$: if a model has cross-entropy of 7 bits per word, its perplexity is $2^7 = 128$. The information-theoretic perspective reveals a profound insight: cross-entropy provides an upper bound on the true entropy of language. If there existed a perfect model that captured all statistical regularities of language, its cross-entropy would equal the true entropy—the minimum number of bits needed to encode text. Every improvement in language modeling reduces cross-entropy toward this theoretical limit, with the gap indicating how much predictable structure the model fails to capture. Shannon himself estimated English entropy at roughly 1 bit per character through human guessing experiments, suggesting that English text is highly redundant and that significant room remains for improvement even in modern language models.

Despite sophisticated smoothing and the success of n-gram models in practical applications, they suffer from fundamental limitations that no amount of engineering can overcome. Figure 2.25 summarizes these limitations. First, the **bounded context window** means n-grams cannot capture dependencies beyond $n-1$ words. Consider "The trophy would not fit in the suitcase because it was too [large/small]"—resolving "it" to either "trophy" or "suitcase" requires understanding the entire sentence, not just the preceding few words. Second, n-grams have **no semantic understanding**: "cat" and "feline" are treated as completely unrelated symbols despite being synonyms. A bigram model that learns "the cat sat" cannot transfer this knowledge to predict "the feline sat." Third, the **exponential parameter growth**—$|\mathcal{V}|^n$ possible n-grams—makes high-order models impractical for realistic vocabularies. Fourth, n-grams cannot **generalize** across similar contexts:

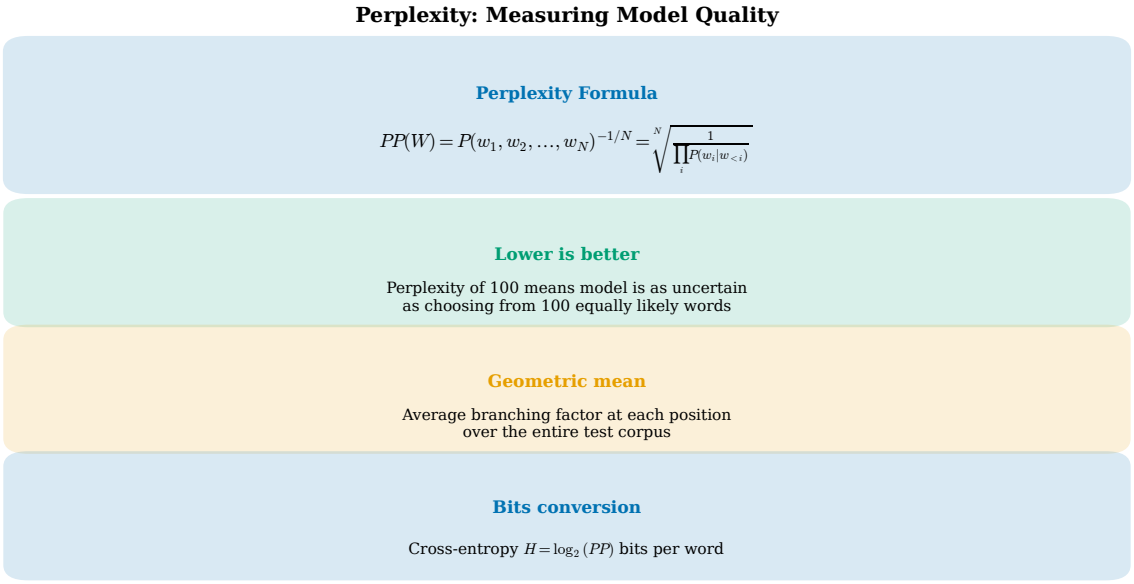


Figure 2.23: Perplexity as a measure of model quality. The formula computes the geometric mean of inverse probabilities across all test words. Lower perplexity indicates better prediction: a perplexity of 100 means the model is as uncertain as choosing uniformly from 100 words. The right panel compares models, from random baseline through neural language models.

seeing “the red car” provides no information about “the crimson automobile” because each n-gram is learned independently without recognizing the underlying semantic similarity.

Figure 2.26 shows concrete next-word prediction examples that illustrate both the strengths and weaknesses of n-gram models. For common patterns like “The cat sat on the ___,” a well-trained trigram model correctly assigns high probability to “mat,” “floor,” and other plausible continuations. Similarly, for the famous phrase “The quick brown ___,” the model has memorized “fox” as the likely continuation. However, the model fails for novel entities: after “The president of ___,” it cannot predict a newly relevant organization like “OpenAI” because it has never seen this specific phrase in training. Rare words pose similar challenges: after “She picked up the ___,” common objects like “phone” and “book” receive high probability, but unusual continuations like “gauntlet” are effectively impossible even when contextually appropriate. These failures reveal that n-gram models are fundamentally limited to reproducing patterns they have explicitly observed, with no capacity for the compositional generalization that humans effortlessly achieve.

The limitations of n-gram models directly motivate the neural language models we develop in subsequent chapters, and understanding these limitations illuminates why neural approaches represent such a fundamental advance rather than merely an incremental improvement. Neural networks address the bounded context problem through architectures like recurrent neural networks that maintain hidden state vectors propagating information across arbitrary distances, and Transformers that use attention mechanisms to directly connect any two positions in a sequence regardless of their separation. They overcome the lack of semantic understanding by learning distributed word representations where semantically similar words occupy nearby points in a continuous vector space, enabling transfer of knowledge between synonyms, analogous contexts, and related concepts that would be completely independent in the n-gram framework. They avoid the curse of dimensionality and exponential parameter growth by sharing the same neural network weights across all positions in the sequence, with model size growing linearly with embedding dimension and layer count rather than exponentially with context length. Most importantly, neural models generalize across similar contexts because the same learned transformations process all inputs, identifying abstract patterns like “adjective before noun”

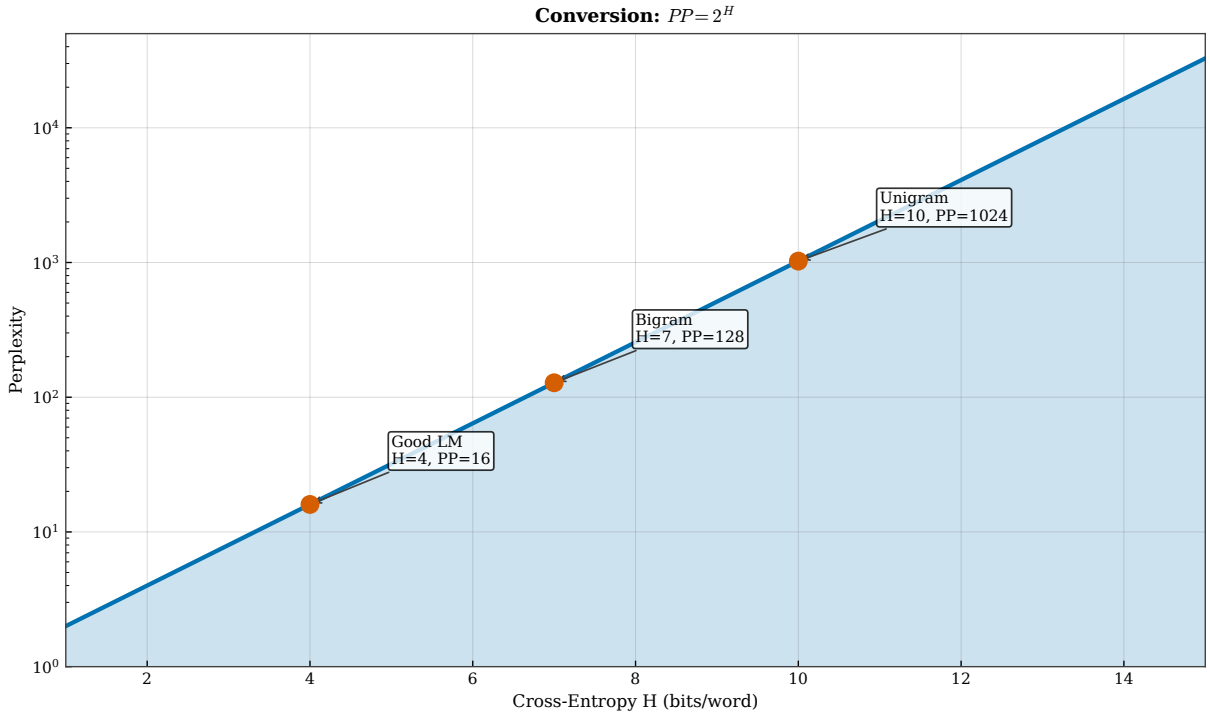


Figure 2.24: The relationship between cross-entropy and perplexity. (a) Perplexity is the exponential of cross-entropy: $PP = 2^H$. (b) Cross-entropy provides an upper bound on the true entropy of language, with better models approaching this theoretical limit.

or "verb requires object" that transfer automatically to novel word combinations never seen during training. Understanding n-gram models and their fundamental limitations is thus essential preparation for appreciating the magnitude of the advance that neural approaches provide, and for understanding why language modeling capabilities improved so dramatically when these architectural innovations became computationally feasible.

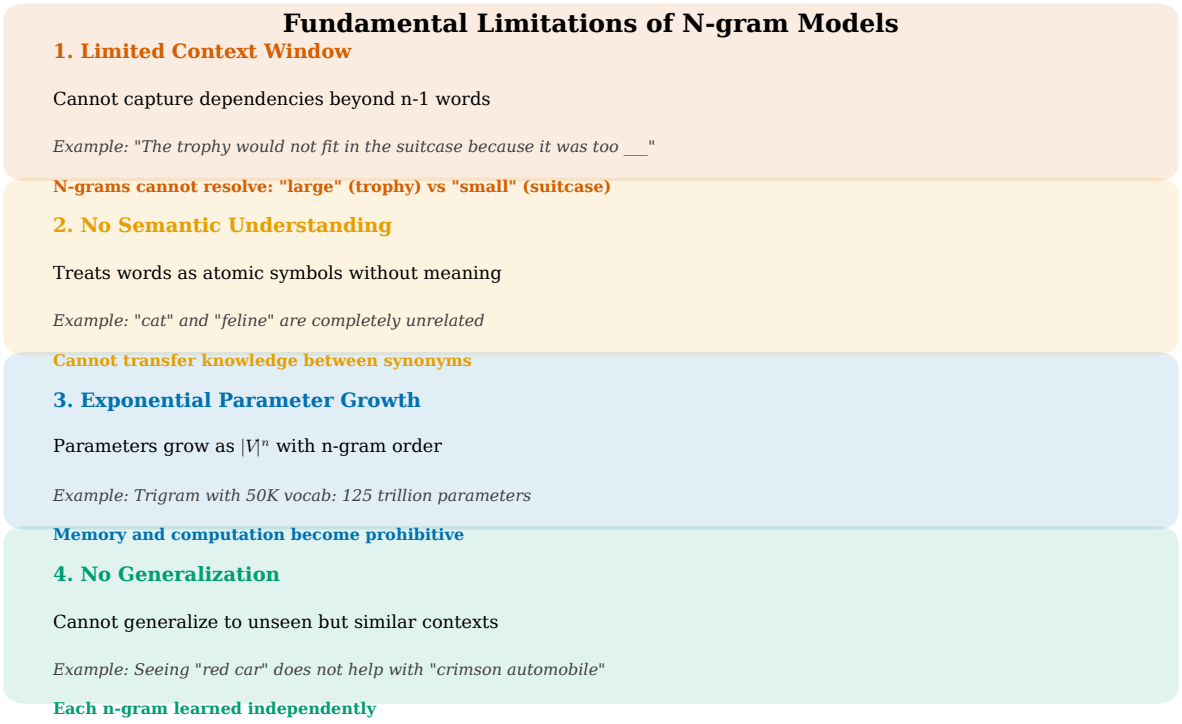


Figure 2.25: Fundamental limitations of n-gram language models. These limitations—bounded context, lack of semantic understanding, exponential parameter growth, and inability to generalize—motivate the development of neural language models that can overcome these barriers.

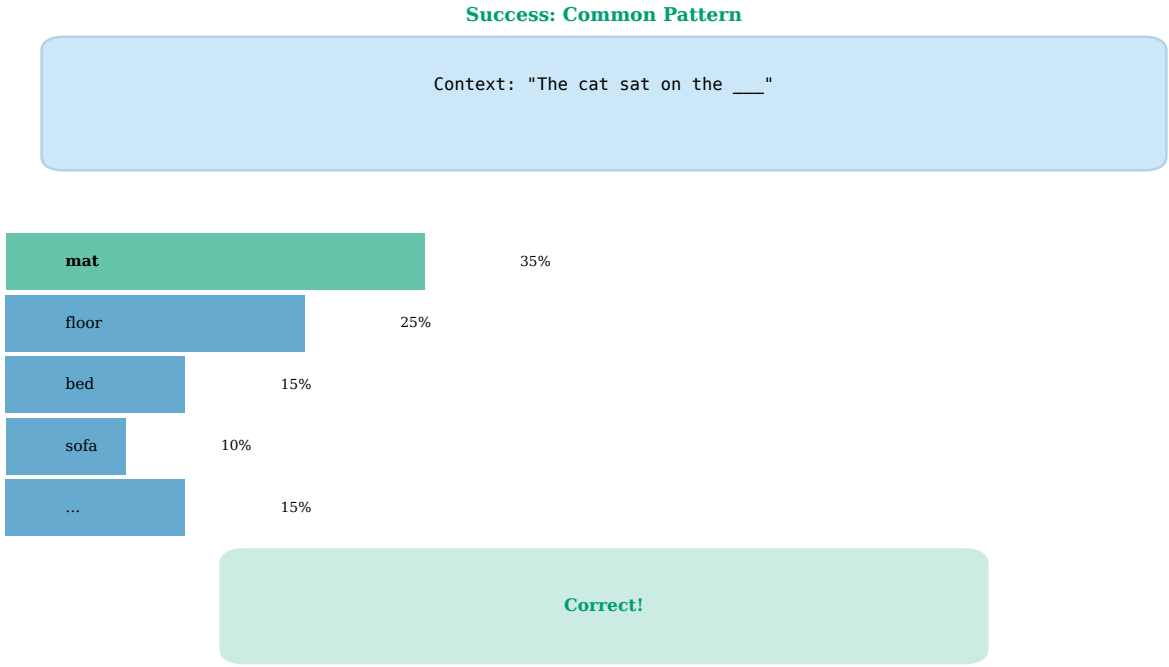


Figure 2.26: Next-word prediction examples showing n-gram successes and failures. Common patterns and memorized phrases are predicted well, but novel entities and rare words expose the model's limitations. These failure cases illustrate why more powerful architectures are needed.

2.7 Summary

This chapter introduced n-gram language models, the foundational approach to next-word prediction that dominated natural language processing for over three decades. Figure 2.27 provides a visual summary of the key concepts. We began with the **Markov assumption**, which makes language modeling tractable by limiting context to the preceding $n-1$ words. This assumption enables **maximum likelihood estimation** through simple count ratios: the probability of a word given its context equals the count of the n-gram divided by the count of the context. However, the **sparsity problem** immediately confronts us: most possible n-grams never appear in training, and raw MLE assigns them zero probability, causing catastrophic failures on test data.

Smoothing techniques address sparsity by redistributing probability mass from observed to unobserved events. We traced the evolution from naive Laplace smoothing through sophisticated Kneser-Ney, which uses continuation probability to capture word versatility rather than raw frequency. **Backoff and interpolation** provide complementary strategies for combining evidence from different n-gram orders, falling back to more reliable lower-order estimates when higher-order evidence is sparse. Finally, we examined **perplexity** as the standard evaluation metric and confronted the **fundamental limitations** of n-gram models: bounded context, lack of semantic understanding, exponential parameter growth, and inability to generalize across similar contexts.

The concepts introduced in this chapter—probability distributions over sequences, the trade-off between context and sparsity, the principle of redistributing probability mass, and the use of perplexity for evaluation—will recur throughout our exploration of neural language models. While neural approaches overcome many n-gram limitations, they face analogous challenges of balancing model capacity against generalization, and the intuitions developed here provide essential foundation for understanding more advanced architectures.

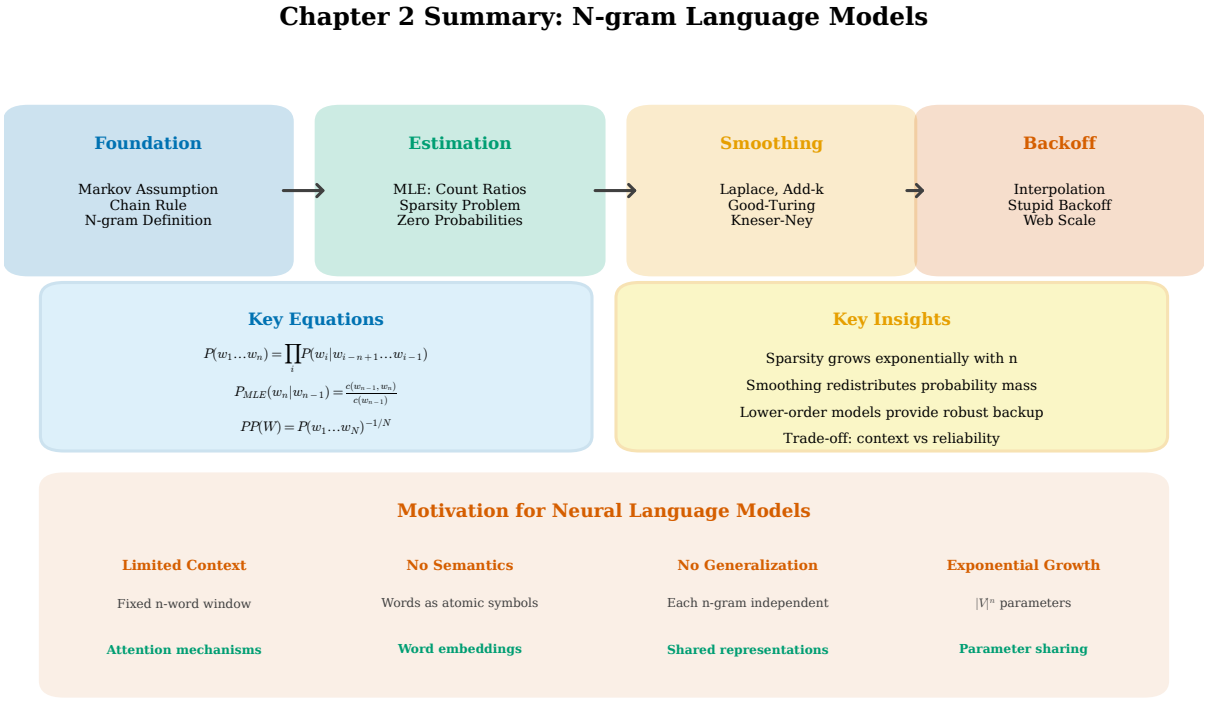


Figure 2.27: Visual summary of n-gram language model concepts. The chapter progresses from foundational concepts through estimation and smoothing to backoff strategies, culminating in an understanding of the limitations that motivate neural approaches.

2.8 Context Representation in N-gram Models

How This Chapter Represents Context
The fundamental question in language modeling is: How do we represent the context $w_1, \ldots, w_{t-1}$ to predict w_t?

- **Context representation**: N-gram models represent context as a fixed-length tuple of the preceding $n-1$ words
- **Context encoding**: The context is encoded implicitly through count table lookups—each unique context maps to a separate probability distribution
- **Limitation**: Contexts differing by a single word are treated as completely independent, with no sharing of statistical strength
- **Next chapter preview**: Neural language models will learn distributed context representations where similar contexts share parameters

N-gram models encode context in the simplest possible way: as an exact tuple of the preceding $n-1$ words, treated as an atomic lookup key with no internal structure. For a bigram model, the context is simply the previous word; for a trigram, the previous two words; and so forth, with each unique tuple serving as an independent index into a table of probability distributions. This representation has the virtue of extreme simplicity—we can look up probabilities in constant time using a hash table or trie indexed by the context tuple, and the resulting probabilities have clear interpretations as relative frequencies—but suffers from severe and fundamental limitations. Every unique context maps to a completely independent probability distribution, so the

model cannot recognize that "the black cat" and "the dark cat" share similar grammatical structure and should predict similar continuations. Changing even one word in the context creates an entirely new lookup key with no connection to related contexts. The exponential growth in the number of possible contexts ($|\mathcal{V}|^{n-1}$ for an n-gram model) means that the vast majority of possible contexts will never appear in any finite training corpus, forcing reliance on backoff and smoothing techniques rather than the direct estimation we would prefer. Neural language models, as we will see in subsequent chapters on word embeddings and recurrent networks, address these limitations by learning distributed representations of context as dense vectors in continuous space, where similar contexts occupy nearby points and therefore produce similar predictions, enabling the compositional generalization that n-gram models fundamentally cannot achieve.

We can now predict better because:

- The **Markov assumption** makes language modeling tractable by limiting context to $n-1$ words
- **Maximum likelihood estimation** from counts provides a simple, interpretable baseline
- **Smoothing techniques** redistribute probability mass to handle the **sparsity problem**
- **Kneser-Ney** captures word versatility through continuation counts, achieving state-of-the-art n-gram performance
- **Perplexity** provides a principled evaluation metric grounded in information theory

Next: Chapter **??** addresses how we define the vocabulary $\mathcal{V}$ itself through tokenization—the crucial preprocessing step that determines what units our language model predicts.

Exercises

1. **N-gram Counting and MLE.** Given the corpus: "`<s> the cat sat on the mat </s>`", "`<s> the dog sat on the floor </s>`", "`<s> the cat ran on the mat </s>`", list all unique bigrams with their counts, then compute $P_{\text{MLE}}(\text{mat}|\text{the})$ and $P_{\text{MLE}}(\text{sat}|\text{cat})$. Explain why $P(\text{bed}|\text{the}) = 0$ is problematic.

2. **Laplace Smoothing.** Using the corpus from Exercise 1 with vocabulary {`<s>`, `</s>`, the, cat, dog, sat, ran, on, mat, floor}, compute Laplace-smoothed probabilities $P_{\text{Laplace}}(\text{mat}|\text{the})$ and $P_{\text{Laplace}}(\text{bed}|\text{the})$. Discuss how add-one smoothing affects frequent versus rare bigrams.

3. **Perplexity Calculation.** A bigram model assigns: $P(\text{the}) = 0.10$, $P(\text{cat}|\text{the}) = 0.05$, $P(\text{sat}|\text{cat}) = 0.30$, $P(\text{</s>}|\text{sat}) = 0.20$. Calculate the sentence probability and perplexity. Express the result in bits per word.

4. **Sparsity Analysis.** For vocabulary size $|V| = 50,000$, compute the number of possible bigrams and trigrams. With a 1-billion-word corpus, estimate what fraction of possible trigrams will remain unobserved and explain why.

5. **Good-Turing Estimation.** Given frequency of frequencies $N_1 = 1000$, $N_2 = 500$, $N_3 = 300$, compute the Good-Turing adjusted count c^* for bigrams appearing once and twice. Calculate the probability mass allocated to unseen events.

6. **Kneser-Ney Insight.** "Francisco" appears 10,000 times (always after "San"); "Monday" appears 1,000 times (after many different words). For the novel context "next ___", explain which word should receive higher probability and why continuation counts matter.

7. **Backoff vs. Interpolation.** Describe how backoff and interpolation differ when estimating $P(\text{mat}|\text{the}, \text{floor})$ with zero trigram count. Provide a scenario where interpolation outperforms pure backoff.

8. **Cross-Entropy and Perplexity.** A model achieves 4 bits per word cross-entropy. Calculate the perplexity and bits per character (assuming 5 characters per word). Compare to Shannon's estimate of 1 bit per character for English.

9. **Context Length Trade-offs.** For vocabulary 30,000 and corpus 100 million words, estimate sparsity for bigram, trigram, and 4-gram models. Recommend an n-gram order and explain how your choice would change with 10 billion words.

10. **Limitations and Neural Solutions.** Explain how neural language models address: (1) bounded context windows, (2) lack of semantic similarity between words, (3) exponential parameter growth, and (4) inability to generalize across unseen contexts.

11. **Stupid Backoff.** With $\alpha = 0.4$, compute the score for 5-gram "the quick brown fox jumps" (count 1000, context count 5000). If unseen, compute the score using 4-gram "quick brown fox jumps" with relative frequency 0.001. Explain why scores differ from probabilities.

Bibliography

Frederick Jelinek. *Self-organized language modeling for speech recognition*. Morgan Kaufmann, 1990.

Index

Predicting the Next Word

From Shannon to ChatGPT

Test Compilation - Chapter 3

Contents

Chapter 3

Tokenization

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

- Understanding how tokenization defines the vocabulary $\mathcal{V}$ over which language models predict
- Exploring subword algorithms (BPE, WordPiece, SentencePiece) that balance vocabulary coverage and computational efficiency
- Analyzing the trade-offs between vocabulary size, fertility, and downstream model performance
- Learning to handle rare words, novel entities, and multilingual text through byte-level fallback mechanisms

3.1 Tokenization and Next-Word Prediction

Before a language model can predict the next word, it must answer a more fundamental question: what exactly constitutes a "word" in the first place? In Chapter **??**, we introduced the vocabulary $\mathcal{V}$ as the set of all possible prediction targets, computing probabilities $P(w_t \mid w_1, \ldots, w_{t-1})$ over this discrete set. However, we deferred a critical question: how do we construct this vocabulary from raw text? The answer lies in tokenization, the process of segmenting continuous text into discrete units called tokens. This seemingly simple preprocessing step has profound implications for everything that follows. The choice of tokenization algorithm determines the size of $\mathcal{V}$, affects how the model handles rare or novel words, influences the computational cost of training and inference, and ultimately shapes what patterns the model can learn from data. Every modern language model, from GPT to BERT to LLaMA, begins with tokenization, making it one of the most consequential design decisions in natural language processing. Understanding tokenization is essential for anyone seeking to build, analyze, or improve language models.

Consider the challenge of predicting the next word after the context "The researcher studied electroencephalography and". A word-level tokenizer might never have seen "electroencephalography" in training data, forcing the model to output an unknown token symbol and effectively giving up on understanding this context. A character-level tokenizer avoids this problem by treating each letter as a separate token, but now the model must process 22 tokens just for this single word, dramatically increasing computational cost and making it harder for attention mechanisms to capture long-range dependencies across hundreds of positions. Subword tokenization offers a middle ground: the word might be split into meaningful pieces like "electro", "encephalog", and "raphy", each appearing frequently enough in training to have learned representations, while keeping the sequence length manageable. The probability computation $P(w_t \mid \mathbf{c})$ now operates over these subword units, fundamentally changing what the model predicts. This chapter explores the spectrum of tokenization approaches, from simple word splitting to sophisticated learned subword vocabularies, examining the trade-offs that have led modern language models to converge on subword tokenization as the dominant paradigm.

Figure 3.1: Tokenization converts raw text into discrete token IDs that index into the vocabulary $\mathscr{V}$. The choice of tokenization algorithm—word-level, character-level, or subword—determines what units the language model learns to predict. Each approach represents a different trade-off between vocabulary size, sequence length, and the ability to handle unseen words.

3.2 Word-Level Tokenization and Its Limitations

The most intuitive approach to tokenization treats whitespace and punctuation as natural boundaries between words, mapping directly to how humans perceive written language as composed of discrete word units. Given the sentence "The cat sat on the mat.", a simple word-level tokenizer splits on spaces and separates punctuation to produce the token sequence ["The", "cat", "sat", "on", "the", "mat", "."]. This approach aligns with human intuition about language structure and was the dominant paradigm in early natural language processing systems, from rule-based parsers of the 1970s through the statistical models of the 1990s and early 2000s. The vocabulary $\mathcal{V}$ consists of all unique words observed in the training corpus, and the language model learns to predict $P(w_t \mid w_1, \ldots, w_{t-1})$ where each w_i is a complete word. For languages with clear word boundaries like English, this approach seems natural and produces reasonable vocabulary sizes when trained on large corpora, typically ranging from 50,000 to 200,000 unique word types depending on the corpus size and domain coverage.

However, word-level tokenization suffers from a fundamental problem known as the out-of-vocabulary (OOV) dilemma. Natural language exhibits a long-tailed distribution of word frequencies: a small number of words appear extremely often, while the vast majority appear rarely. Zipf's law, which we encountered in Chapter **??**, tells us that the frequency of a word is inversely proportional to its rank. In a typical English corpus, approximately half of all unique word types appear only once, and many more appear fewer than five times. When the model encounters a word it has never seen during training, it must map that word to a special unknown token [UNK], losing all information about the original word. This problem becomes severe when processing text from domains not represented in training data, such as technical documents, social media with creative spellings, or text containing proper nouns like brand names or personal names that emerged after the training data was collected.

The vocabulary explosion problem becomes even more severe in morphologically rich languages, where the combinatorial possibilities of word formation vastly exceed those of analytic languages like English. Consider Finnish, Turkish, or Hungarian, where words are formed by combining stems with numerous prefixes and suffixes to express grammatical relationships, generating an exponentially larger space of valid word forms. A single Finnish verb can have thousands of inflected forms: "juoksentelisinkohan" means "I wonder if I should run around aimlessly." In agglutinative languages like Turkish, the word "evlerinizdeki" (meaning "that which is in your houses") combines multiple morphemes into a single orthographic word that would be a phrase in English. German compounds nouns freely: "Rindfleischetikettierungsüberwachungsaufgabenübertragungsgesetz" is a single word describing a law about beef labeling supervision. For word-level tokenization, each unique combination must be stored as a separate vocabulary entry, with no parameter sharing between related forms. A vocabulary that covers 99% of English words might cover less than 90% of Finnish or Turkish text, and achieving similar coverage would require vocabularies orders of magnitude larger, making the embedding matrix computationally prohibitive and storage requirements impractical. This fundamental limitation motivated the development of subword tokenization methods that can decompose complex words into reusable pieces, sharing representations across morphologically related forms and enabling effective handling of productive word formation processes.

Figure 3.2: Word-level tokenization faces three interconnected problems: (1) the long-tailed frequency distribution means most words are rare, (2) out-of-vocabulary words must be mapped to [UNK] tokens losing all information, and (3) morphologically rich languages generate exponentially more word forms than analytic languages like English.

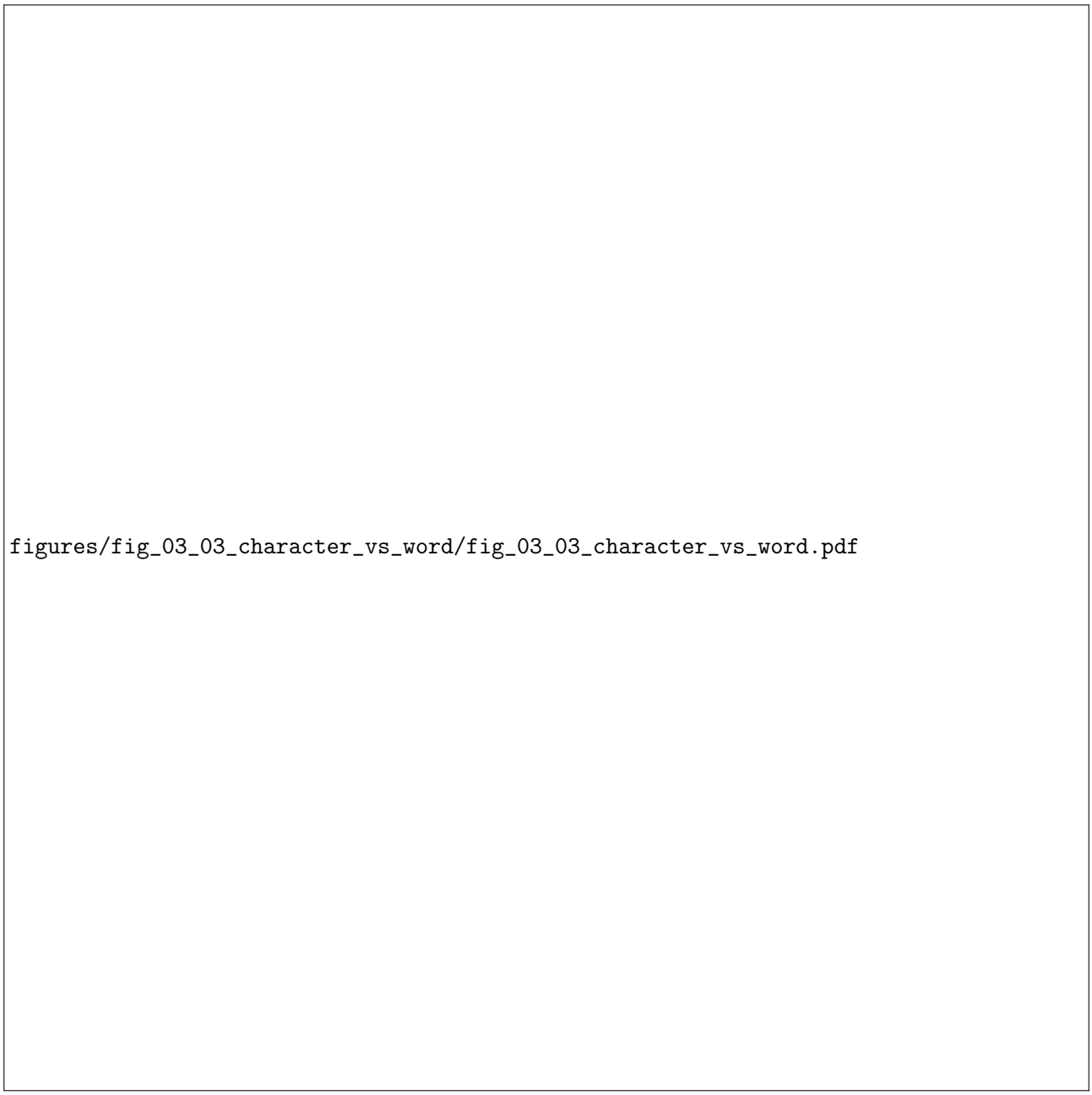


Figure 3.3: The same sentence tokenized at character level versus word level illustrates the fundamental trade-off: character-level tokenization eliminates out-of-vocabulary words entirely but produces much longer sequences that increase computational cost and make it harder for models to capture long-range dependencies.

3.3 Character-Level and Byte-Level Approaches

One radical solution to the out-of-vocabulary problem is to abandon words entirely and operate at the level of individual characters. A character-level tokenizer for English needs only about 100 tokens to represent the full ASCII character set, or perhaps 256 tokens to cover the extended ASCII range including accented characters and special symbols. The vocabulary $\mathcal{V}$ becomes tiny and fixed: we never encounter an unknown character because all possible characters are enumerated in advance. The word "electroencephalography" that caused problems for word-level tokenization is simply represented as the sequence ["e", "l", "e", "c", "t", "r", "o", ...], each character having a well-defined entry in the vocabulary. Novel words, creative spellings, technical terms, and proper nouns all decompose naturally into their constituent characters without requiring any special handling. This elegant solution to the OOV problem was explored in early neural language models and continues to find applications in specific domains where character-level patterns matter, such as morphological analysis, spelling correction, and code generation where individual characters carry syntactic significance.

The cost of character-level tokenization is dramatically increased sequence length. A typical English word contains about 5 characters on average, so a character-level model must process roughly five times as many tokens as a word-level model for the same text. This matters enormously for computational efficiency, particularly with attention-based architectures where the computational cost scales quadratically with sequence length, meaning that a 5x increase in sequence length produces a 25x increase in attention computation. A sentence that word-level tokenization represents with 20 tokens might require 100 tokens at the character level. Training on documents of reasonable length becomes prohibitively expensive, and the model must learn to compose meaning across much longer distances. While a word-level model can directly associate "cat" with feline concepts, a character-level model must learn that the sequence ["c", "a", "t"] forms a meaningful unit distinct from ["c", "a", "r"] or ["c", "a", "p"], and that this unit carries semantic content about small furry animals. This hierarchical composition must be learned entirely from data, without any explicit supervision about word boundaries.

Byte-level tokenization pushes this idea to its logical extreme by treating text as a sequence of raw bytes. Every Unicode character, regardless of script or language, has a well-defined UTF-8 encoding as a sequence of 1 to 4 bytes. The vocabulary contains exactly 256 possible byte values, providing complete coverage of any text that can be represented digitally. This includes not only all human writing systems—Latin, Cyrillic, Arabic, Chinese, Japanese, Korean, and hundreds more—but also emoji, mathematical symbols, and any other Unicode characters that might appear in training or inference data. The byte-level approach guarantees that no input can ever produce an unknown token, making it truly universal and future-proof against new characters added to Unicode. GPT-2 pioneered a practical implementation using byte-level BPE, treating bytes as the base units from which larger subword tokens are built, combining the universality of byte-level representation with the efficiency gains of learning common byte sequences. This hybrid approach has become the standard for modern large language models, providing robustness without sacrificing computational efficiency.

Figure 3.4: Tokenization exists on a spectrum from characters (small vocabulary, long sequences) to words (large vocabulary, short sequences). Subword tokenization occupies the middle ground, learning to segment text into units that balance vocabulary size against sequence length while maintaining the ability to represent any input through composition.

3.4 Subword Tokenization

Subword tokenization emerged as the dominant paradigm in modern language modeling by finding a principled middle ground between the extremes of word-level and character-level approaches. The core insight is that while words vary enormously in frequency, certain sub-word units—prefixes, suffixes, and stems—recur across many words and can serve as efficient building blocks. The word "unhappiness" can be decomposed into ["un", "happi", "ness"], where each piece appears in many other words: "un" in "undo", "unclear", "unfair"; "ness" in "darkness", "kindness", "awareness". By learning a vocabulary of such reusable pieces, subword tokenization achieves reasonable sequence lengths while maintaining the ability to represent any word through composition. Several algorithms have been developed to learn these vocabularies automatically from data, each with slightly different properties and trade-offs. The key algorithms—Byte Pair Encoding, WordPiece, SentencePiece, and Unigram Language Model tokenization—share the fundamental goal of discovering useful subword units but differ in how they construct and select vocabulary items.

3.4.1 Byte Pair Encoding (BPE)

Byte Pair Encoding, originally developed as a data compression algorithm in 1994, was adapted for neural machine translation by Sennrich, Haddow, and Birch in 2016 [Sennrich et al., 2016]. The algorithm begins with a vocabulary containing only individual characters (or bytes, in byte-level BPE), then iteratively merges the most frequent adjacent pair of tokens to create a new vocabulary entry. Starting with the word "lower" represented as ["l", "o", "w", "e", "r"], if the pair ("e", "r") appears most frequently across the training corpus, it is merged to create a new token "er", and all occurrences of the sequence are replaced. The vocabulary now contains ["l", "o", "w", "e", "r", "er"], and "lower" is represented as ["l", "o", "w", "er"]. This process repeats for a predetermined number of merge operations—typically 30,000 to 50,000—gradually building up common subwords, whole words, and even multi-word sequences. The order of merges is recorded and must be applied in the same sequence during inference to ensure consistent tokenization between training and deployment.

The beauty of BPE lies in its simplicity and its connection to data compression. Each merge operation reduces the total number of tokens needed to represent the training corpus by replacing two tokens with one wherever the merged pair occurs. Frequent words quickly become single tokens: after enough merges, common words like "the", "and", "is" will be represented as single vocabulary entries because their constituent character sequences appear together so often. Rare words, conversely, remain decomposed into smaller pieces that they share with other words. The word "electroencephalography" might be tokenized as ["electro", "encephal", "ography"], each piece appearing in other medical or scientific terms. This automatic discovery of morphological structure emerges purely from frequency statistics, without any linguistic knowledge being provided to the algorithm. The algorithm makes no distinction between morphological boundaries and accidental letter combinations; both emerge from the same frequency-based merging process, yet the result often aligns surprisingly well with meaningful linguistic units.

3.4.2 WordPiece

WordPiece, developed at Google for their neural machine translation system [Wu et al., 2016], takes a similar iterative approach to BPE but uses a different criterion for selecting which pairs to merge, grounding the decision in statistical principles rather than simple frequency counting. Instead of simply choosing the most frequent pair, WordPiece selects the pair that maximizes the likelihood of the training data when the merge is applied. Formally, if we consider merging symbols x and y to create xy, WordPiece computes the increase in log-likelihood: $\log P(xy) - \log P(x) - \log P(y)$. This criterion favors merging pairs where the combined token is much more likely than would be predicted by the independent frequencies of its parts, capturing genuine co-occurrence patterns rather than just raw frequency. A pair like ("t", "h") might be very frequent in English, but ("t", "he") captures a stronger dependency because "the" is an extremely common word. This mutual information-like criterion provides a more principled approach to deciding which merges are most valuable.

The practical difference between BPE and WordPiece is subtle but meaningful in terms of the vocabularies they produce. BPE's frequency-based merging tends to create tokens that reflect simple surface statistics,

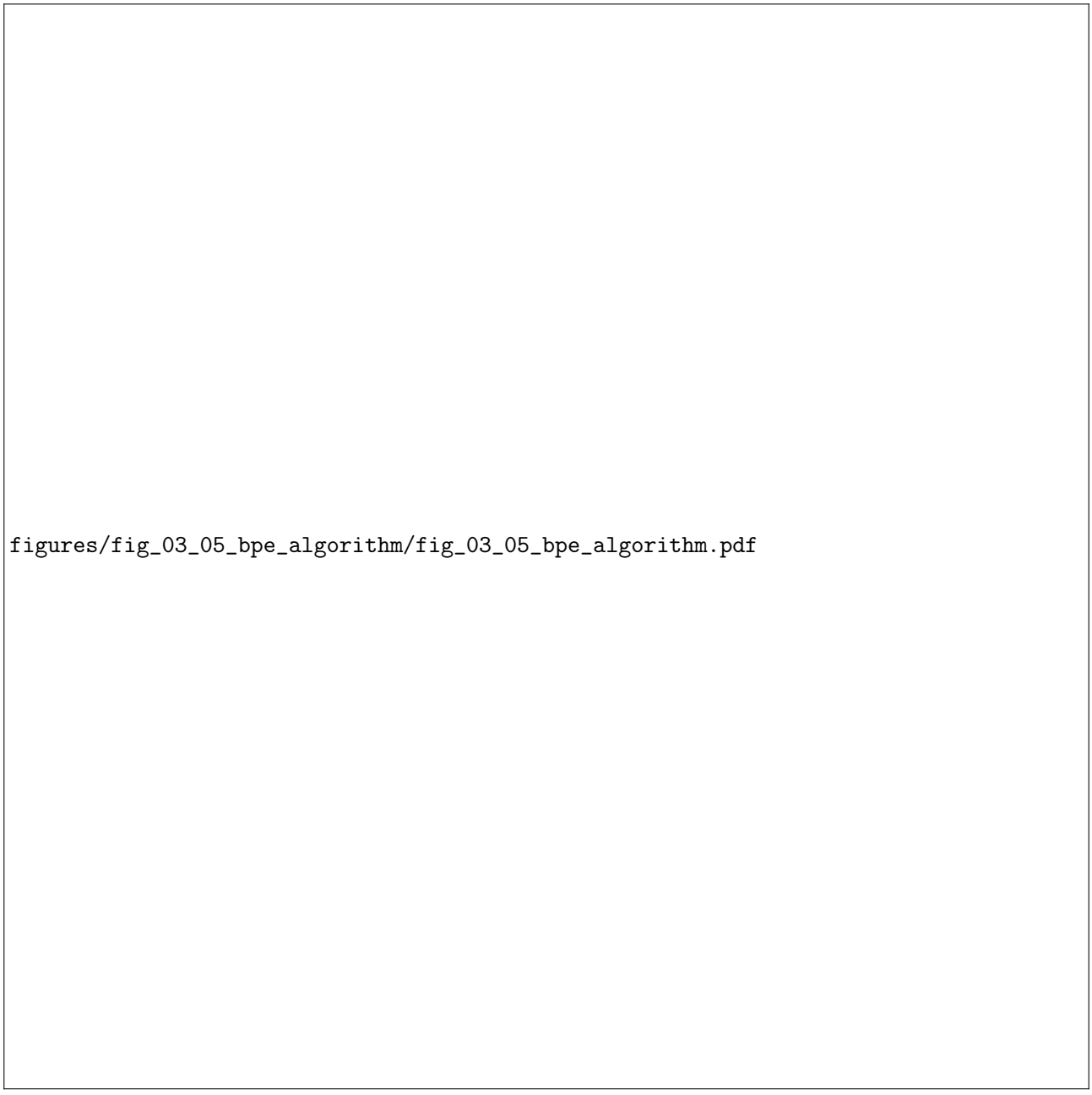


Figure 3.5: Byte Pair Encoding iteratively merges the most frequent adjacent token pair. Starting from a character vocabulary, each merge operation adds one new token and reduces the total sequence length across the corpus. The algorithm terminates after a fixed number of merges, which determines the final vocabulary size.

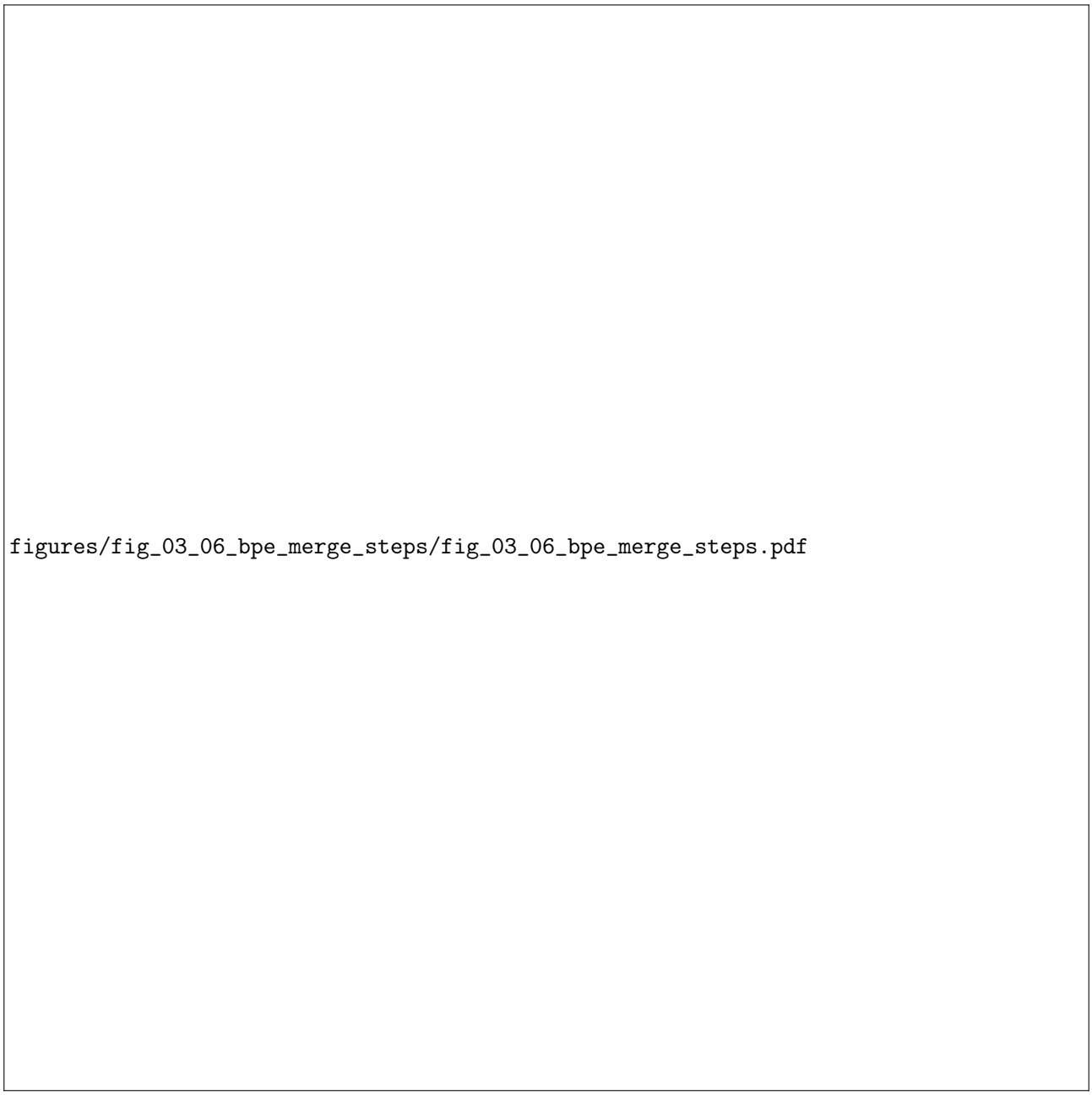


Figure 3.6: A detailed trace of BPE merges on a sample corpus shows how the algorithm progressively builds larger tokens from frequent character pairs. Each step displays the current vocabulary, pair frequencies, and the resulting tokenization of the corpus.

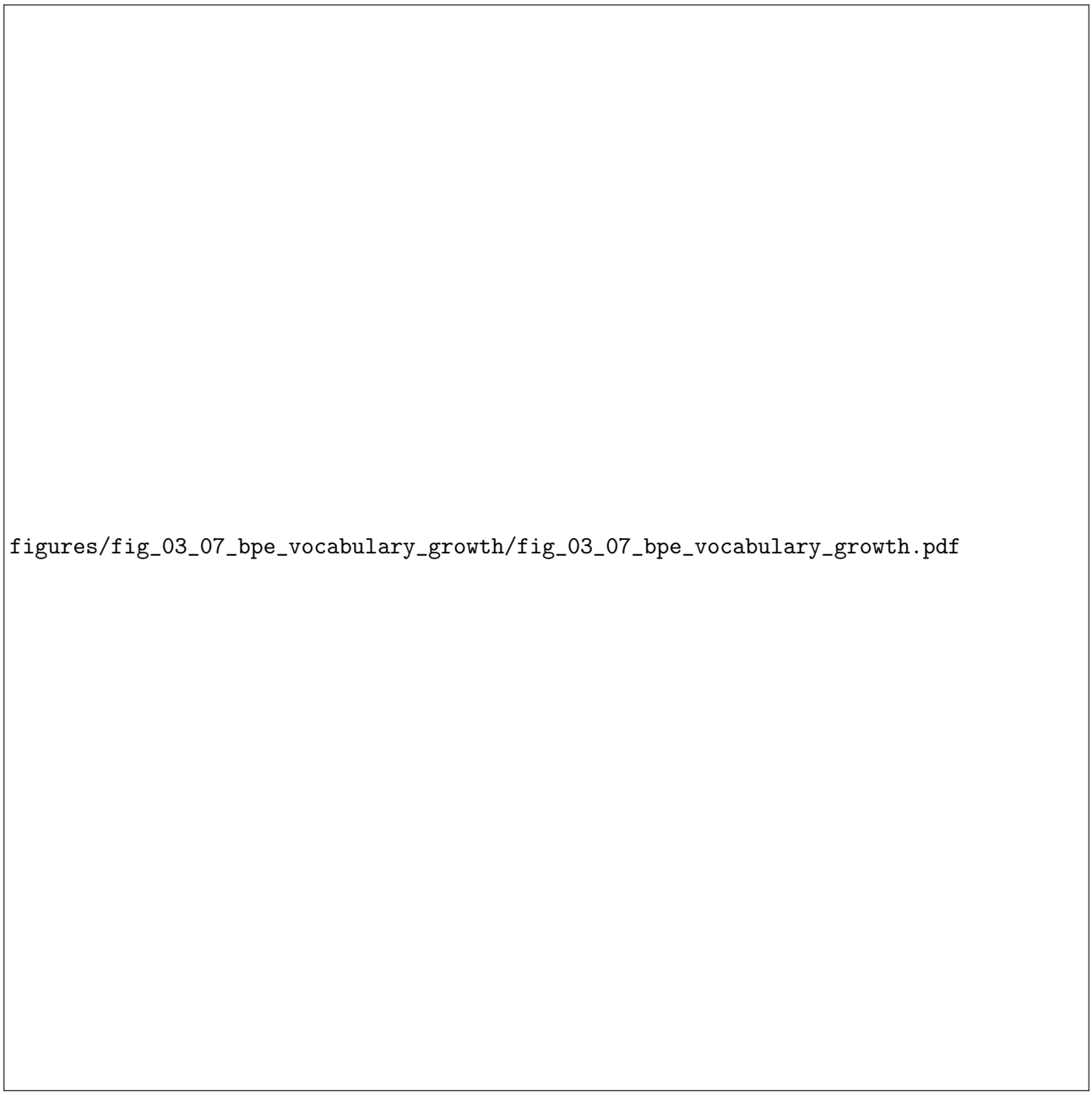


Figure 3.7: As BPE performs more merge operations, the vocabulary grows linearly while the total number of tokens needed to represent the corpus decreases. This trade-off between vocabulary size and sequence length is the fundamental parameter that practitioners must tune.

while WordPiece's likelihood-based criterion can discover more linguistically meaningful units by focusing on tokens that co-occur more than chance would predict. WordPiece became widely known through its use in BERT and subsequent transformer models, where it typically produces vocabularies of around 30,000 tokens. One distinctive feature of WordPiece is its use of the "##" prefix to indicate subword tokens that continue a word: "playing" might be tokenized as ["play", "##ing"], making it explicit that "##ing" is not a word-initial token. This notation helps with detokenization and makes the tokenization more interpretable for humans examining model behavior, while also providing the model with information about whether a subword begins a new word or continues an existing one, potentially improving the model's ability to handle word boundaries during prediction.

Figure 3.8: WordPiece selects merges based on likelihood improvement rather than raw frequency. The algorithm also marks continuation tokens with a special prefix (##) to distinguish word-initial from word-internal subwords.

3.4.3 SentencePiece

SentencePiece, developed by Kudo and Richardson at Google [Kudo and Richardson, 2018], addresses a fundamental limitation shared by both BPE and WordPiece: their reliance on language-specific pre-tokenization. Standard implementations of these algorithms assume that the input has already been split into words by whitespace, then learn subword vocabularies within those word boundaries. This assumption works reasonably well for languages like English that use spaces between words, but fails completely for languages like Chinese, Japanese, and Thai where words are not separated by whitespace and must be identified through other means such as dictionary lookup or statistical segmentation, which themselves require language-specific resources. Even for English, pre-tokenization introduces language-specific rules for handling punctuation, contractions, and other edge cases that may not generalize across languages or domains, creating a brittle dependency on language-specific preprocessing that complicates truly multilingual systems and can introduce subtle bugs when processing text that violates expected conventions or contains mixed-language content where multiple scripts appear together.

SentencePiece treats the input as a raw stream of Unicode characters, with whitespace treated as just another character rather than a word boundary, eliminating the need for language-specific preprocessing entirely. The underscore character (displayed as a special symbol) replaces spaces in the input, so "Hello world" becomes "_Hello_world". Subword learning then proceeds without any assumption about word boundaries, discovering segments that may cross what humans would consider word boundaries. This approach makes SentencePiece truly language-agnostic: the same algorithm can be applied to English, Japanese, Thai, or any other language without modification, using identical code paths for all languages. The framework supports both BPE and unigram language model algorithms for the actual vocabulary learning, providing flexibility in choosing the underlying tokenization method while maintaining the language-independent preprocessing. This flexibility has made SentencePiece the tokenization framework of choice for many multilingual and cross-lingual language models including mBERT, XLM-R, and T5.

3.4.4 Unigram Language Model Tokenization

The unigram language model approach to tokenization, also developed by Kudo and available in SentencePiece, takes a fundamentally different approach from the bottom-up merging of BPE and WordPiece. Instead of starting small and building up, the unigram method starts with a large vocabulary of candidate subwords and iteratively removes the least useful ones, proceeding top-down rather than bottom-up. The algorithm begins by generating a large set of potential subword candidates—often all substrings up to a certain length that appear in the corpus, potentially numbering in the millions—then fits a unigram language model where each subword has an independent probability. Given this model, it computes the tokenization of the training corpus that maximizes likelihood, then identifies which vocabulary items contribute least to this likelihood. These low-impact items are pruned, the model is re-estimated on the reduced vocabulary, and the process repeats until reaching the target vocabulary size, typically requiring dozens of pruning iterations.

The key insight of unigram tokenization is that the same word can have multiple valid tokenizations, and the model explicitly represents this ambiguity rather than committing to a single deterministic segmentation as BPE and WordPiece do. The word "internationalization" might be tokenized as ["international", "ization"] or ["inter", "national", "ization"] or many other valid segmentations, each with a probability under the unigram model that reflects the learned subword frequencies from the training corpus. During training, this ambiguity can be exploited through subword regularization: rather than always using the single most likely tokenization, the model samples from the distribution of possible tokenizations, exposing it to different segmentations of the same word across different training examples. This regularization acts as a powerful form of data augmentation, making the model more robust to variations in how words are tokenized and reducing overfitting to specific subword boundaries that might be artifacts of the tokenization algorithm rather than meaningful linguistic structure.

Figure 3.9: SentencePiece treats whitespace as a regular character (represented by underscore) and learns subword vocabulary directly from raw text without pre-tokenization. This makes it applicable to any language regardless of word boundary conventions.

Figure 3.10: The unigram approach assigns probabilities to subwords and can represent multiple valid tokenizations of the same input. This probabilistic view enables subword regularization, where training samples different tokenizations to improve robustness.

3.5 Vocabulary Size and Design Decisions

Choosing the vocabulary size $|\mathcal{V}|$ is one of the most important practical decisions when training a tokenizer and the language model that depends on it. Common vocabulary sizes range from 8,000 tokens for smaller models to 50,000 or more for large multilingual systems, with 32,000 being a popular choice for many English-focused models such as GPT-2 and its successors. This choice affects multiple aspects of the system: the size of the embedding matrix (which scales linearly with vocabulary size and can represent a significant fraction of total model parameters for smaller models), the average sequence length (which decreases as vocabulary grows), and the model's ability to represent rare or domain-specific terms without resorting to fine-grained character-level decomposition. There is no universally optimal vocabulary size; the best choice depends on the languages covered, the domains of interest, the available computational budget, and the trade-off between model capacity and inference speed that practitioners must navigate based on deployment requirements.

The concept of fertility provides a useful lens for understanding vocabulary size trade-offs. Fertility is defined as the average number of tokens produced per word when text is tokenized, measuring how efficiently the tokenizer compresses text. A word-level tokenizer has fertility of exactly 1.0 by definition, while a character-level tokenizer for English has fertility of approximately 5.0 (the average word length). Subword tokenizers fall somewhere in between: a BPE tokenizer with 32,000 vocabulary might achieve fertility of 1.2 to 1.5 on English text, meaning most words are single tokens but some are split into two or three pieces. Fertility varies dramatically across languages: the same tokenizer might have fertility of 1.3 on English, 1.8 on German (with its compound nouns), and 2.5 or higher on morphologically rich languages like Finnish. Lower fertility means shorter sequences and faster processing, but requires larger vocabularies and embedding matrices; this trade-off must be carefully balanced when designing tokenizers for specific applications.

Beyond the learned vocabulary, modern language models reserve special tokens for structural purposes that go beyond simple text representation. The [UNK] (unknown) token handles the rare case where byte-level fallback is not available and an input cannot be tokenized, serving as a last-resort placeholder. The [PAD] (padding) token fills sequences to uniform length when batching inputs of different sizes, allowing efficient parallel processing on GPUs by ensuring all sequences in a batch have identical dimensions. For models trained with masked language modeling like BERT, the [MASK] token replaces words that the model must predict during training, creating the training signal for bidirectional context learning. Sequence boundary tokens like [CLS] (classification) and [SEP] (separator) provide explicit markers for the start of sequences and boundaries between sentence pairs, with [CLS] often serving as an aggregate representation of the entire sequence for classification tasks. The [BOS] (beginning of sequence) and [EOS] (end of sequence) tokens serve similar purposes in autoregressive models like GPT, marking where generation should start and when it should stop, essential for controlling generation during inference.

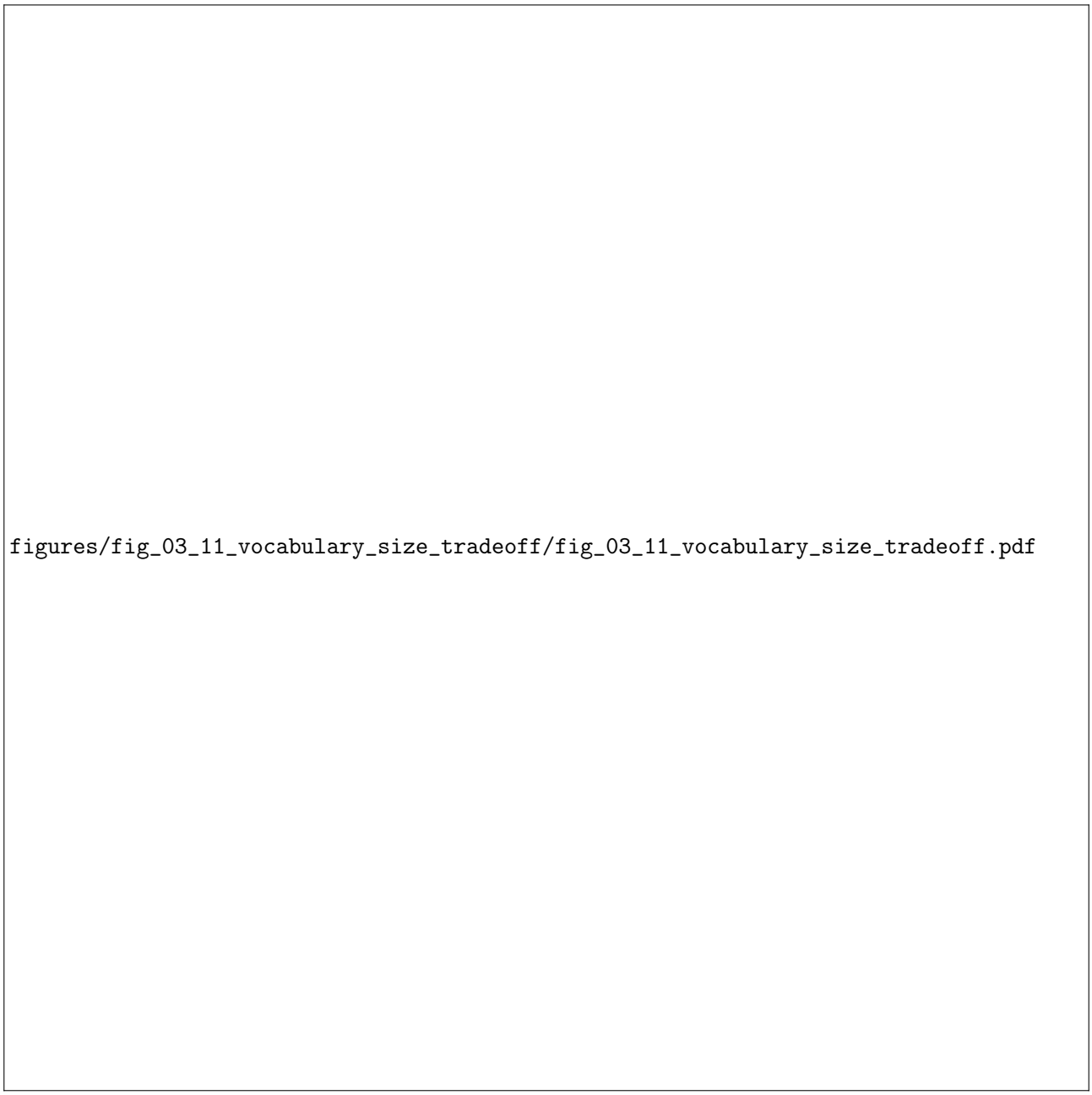


Figure 3.11: Larger vocabularies require more parameters for the embedding matrix but produce shorter token sequences. The optimal vocabulary size balances these competing concerns based on model architecture and computational constraints.

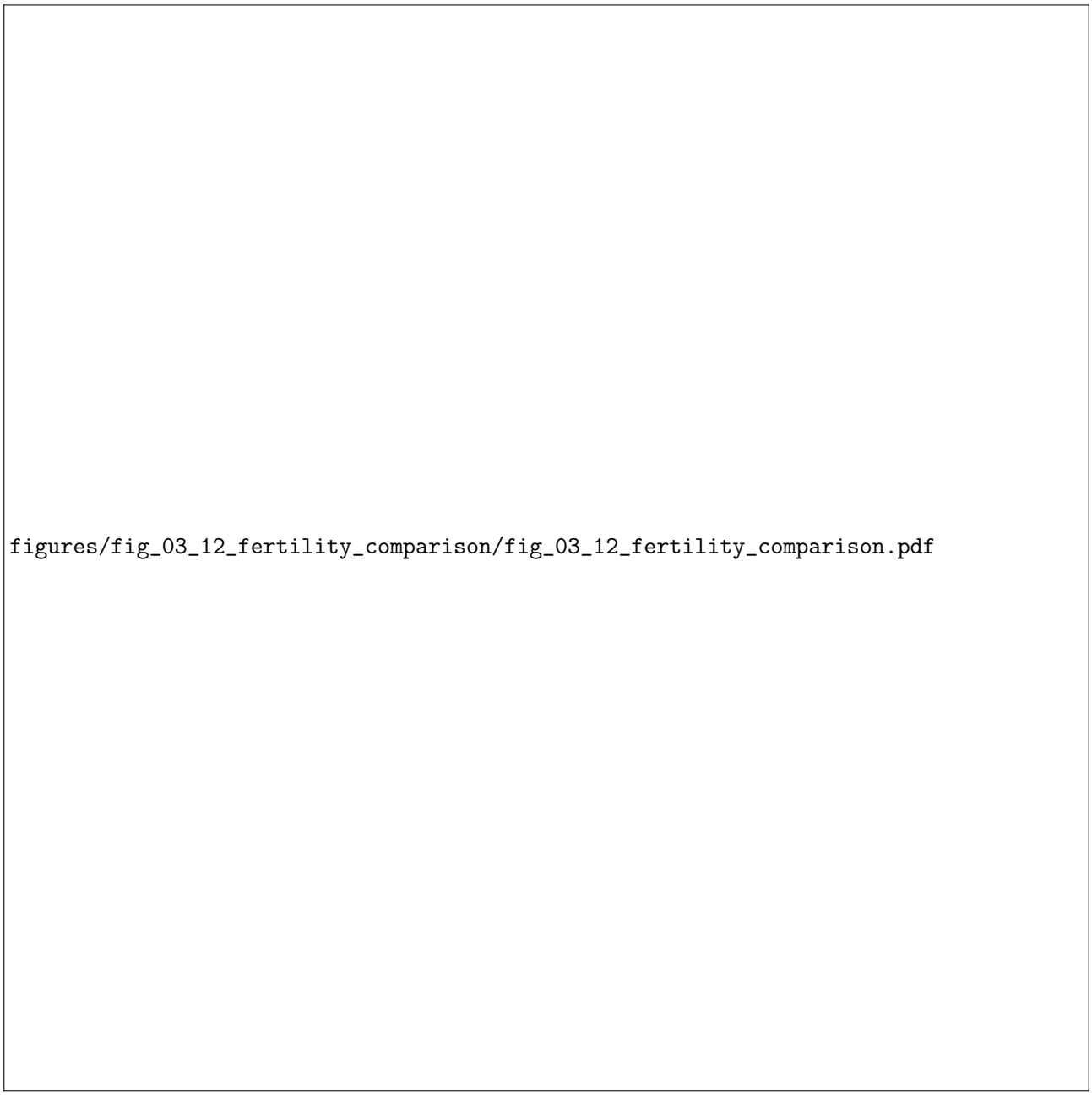


Figure 3.12: Fertility varies significantly across languages with the same tokenizer. English typically achieves the lowest fertility, while morphologically rich languages like Finnish or agglutinative languages like Turkish require more tokens per word.

Figure 3.13: Modern tokenizers use byte-level fallback to guarantee complete coverage: any character that cannot be tokenized as a learned subword is represented as a sequence of byte tokens, ensuring no input ever maps to [UNK].

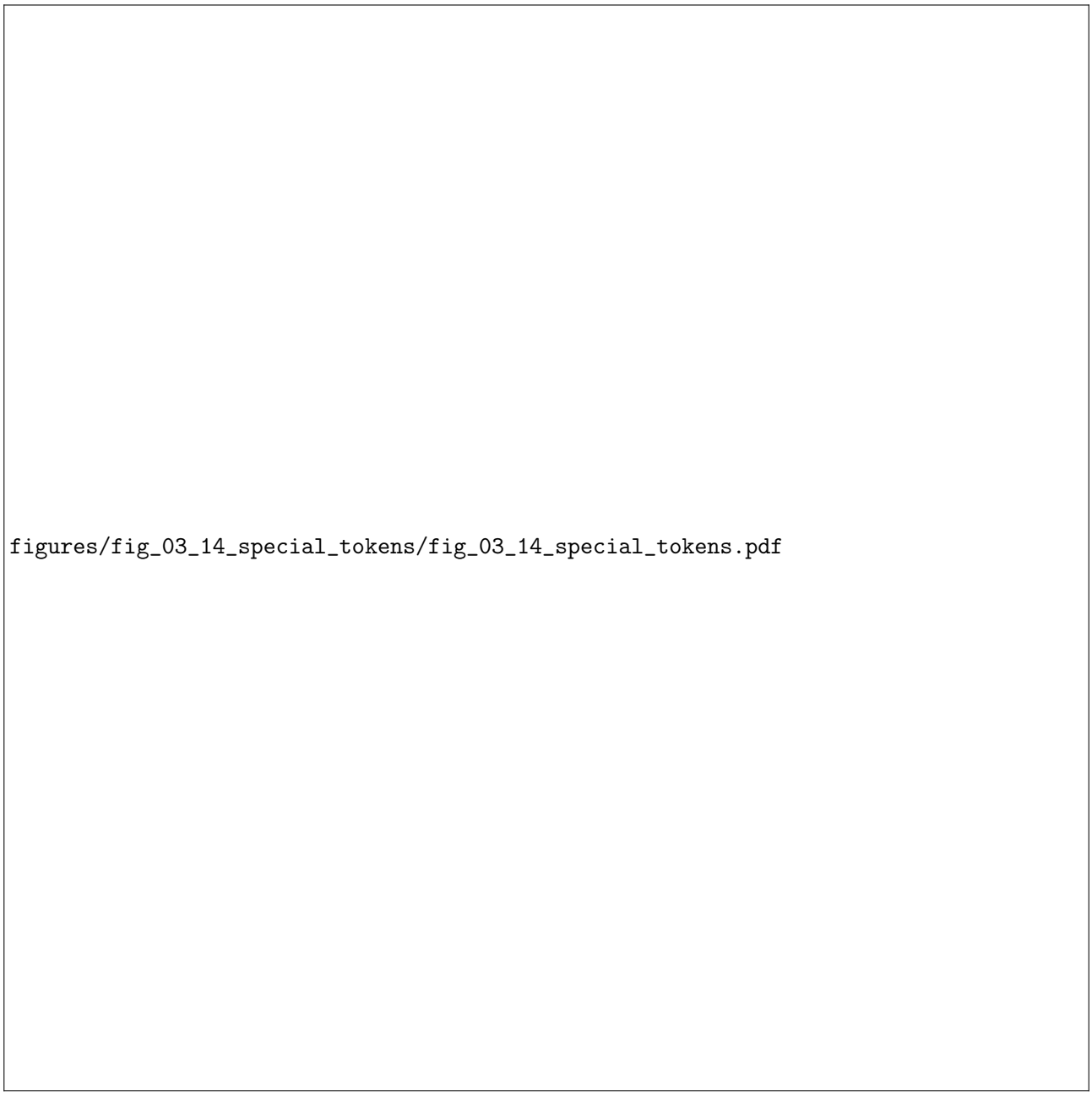


Figure 3.14: Special tokens provide structural information beyond text content. Each serves a specific purpose in model training or inference, from marking sequence boundaries to enabling masked language modeling.

3.6 Evaluating Tokenization Quality

Evaluating tokenization quality requires metrics that capture different aspects of how well a tokenizer serves the downstream language modeling task. The most direct metrics are fertility and compression ratio, which measure how efficiently the tokenizer represents text from complementary perspectives. Fertility, as discussed above, counts tokens per word, while compression ratio compares the number of bytes in the original text to the number of tokens produced. A tokenizer with compression ratio of 4 means that, on average, each token represents 4 bytes of the original text, effectively compressing the input by a factor of 4 in terms of sequence length. Higher compression ratios indicate more efficient tokenization, but this must be balanced against vocabulary size: trivially, a vocabulary containing every possible word would achieve compression ratio equal to average word length, but would require an impossibly large embedding matrix and would fail on any word not seen during vocabulary construction.

Coverage metrics assess how well the tokenizer handles the full range of inputs it might encounter, measuring the tokenizer's ability to efficiently represent diverse text from different languages and domains without excessive fragmentation into small pieces. For tokenizers without byte-level fallback, coverage measures what percentage of tokens in test data can be represented without mapping to [UNK], a critical metric that directly impacts model performance on out-of-distribution text. Even with perfect coverage guaranteed by byte fallback, coverage-like metrics remain useful: a tokenizer that represents most Chinese characters as multi-byte sequences rather than single tokens will have poor effective coverage for Chinese text, even though it technically produces valid output without any unknown tokens. Measuring the percentage of characters or words that tokenize to single tokens, rather than requiring multiple tokens, provides insight into how well the tokenizer serves different languages or domains. A tokenizer trained primarily on English text might achieve 95% single-token coverage on English words but only 30% on Japanese text, revealing significant bias toward the training language distribution that may substantially harm multilingual model performance and create unfair computational costs for non-English users.

The ultimate evaluation of tokenization quality comes from downstream task performance, which captures effects that simpler metrics cannot adequately measure. Two tokenizers with similar compression ratios and coverage might produce very different results when used to train language models on the same data. Factors that are difficult to capture in simple metrics—such as whether token boundaries align with morpheme boundaries, whether semantically related words share subword tokens, or whether the tokenization is stable across minor variations in input—can significantly affect model quality in ways that only become apparent through end-to-end evaluation on real tasks. Practitioners typically evaluate tokenizers by training small proxy models and measuring perplexity or downstream task accuracy, using these results to guide decisions about vocabulary size and tokenization algorithm for the final large-scale training run. This empirical approach to tokenizer selection reflects the difficulty of predicting a priori which tokenization choices will work best for a given application or language distribution.

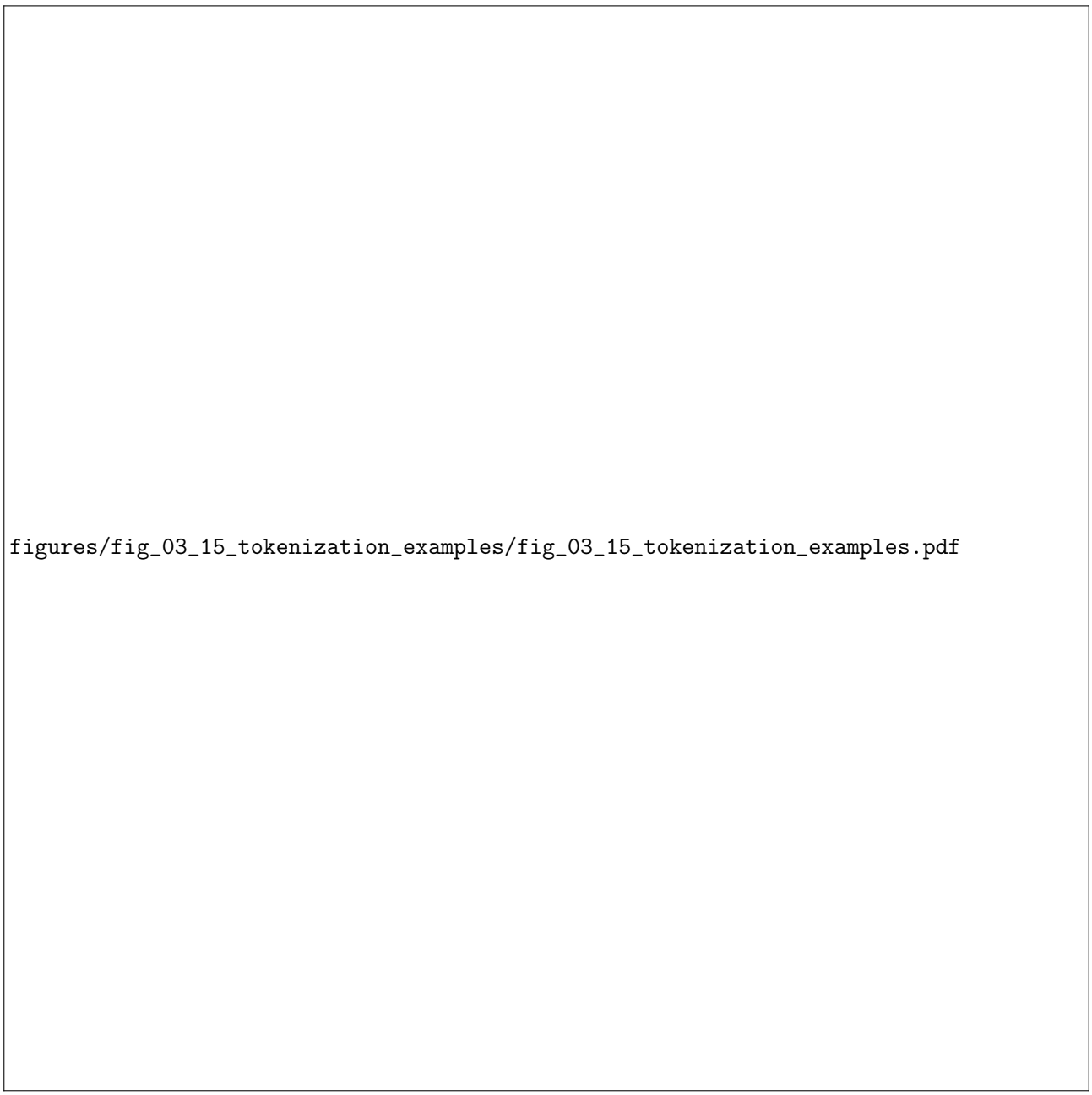


Figure 3.15: The same text tokenized by word-level, character-level, BPE, and WordPiece tokenizers illustrates the different trade-offs each method makes between vocabulary size, sequence length, and linguistic meaningfulness of token boundaries.

Figure 3.16: A tokenizer trained on predominantly English data shows dramatically different performance across languages. Languages with different scripts, morphological complexity, or word boundary conventions require careful consideration during tokenizer training.

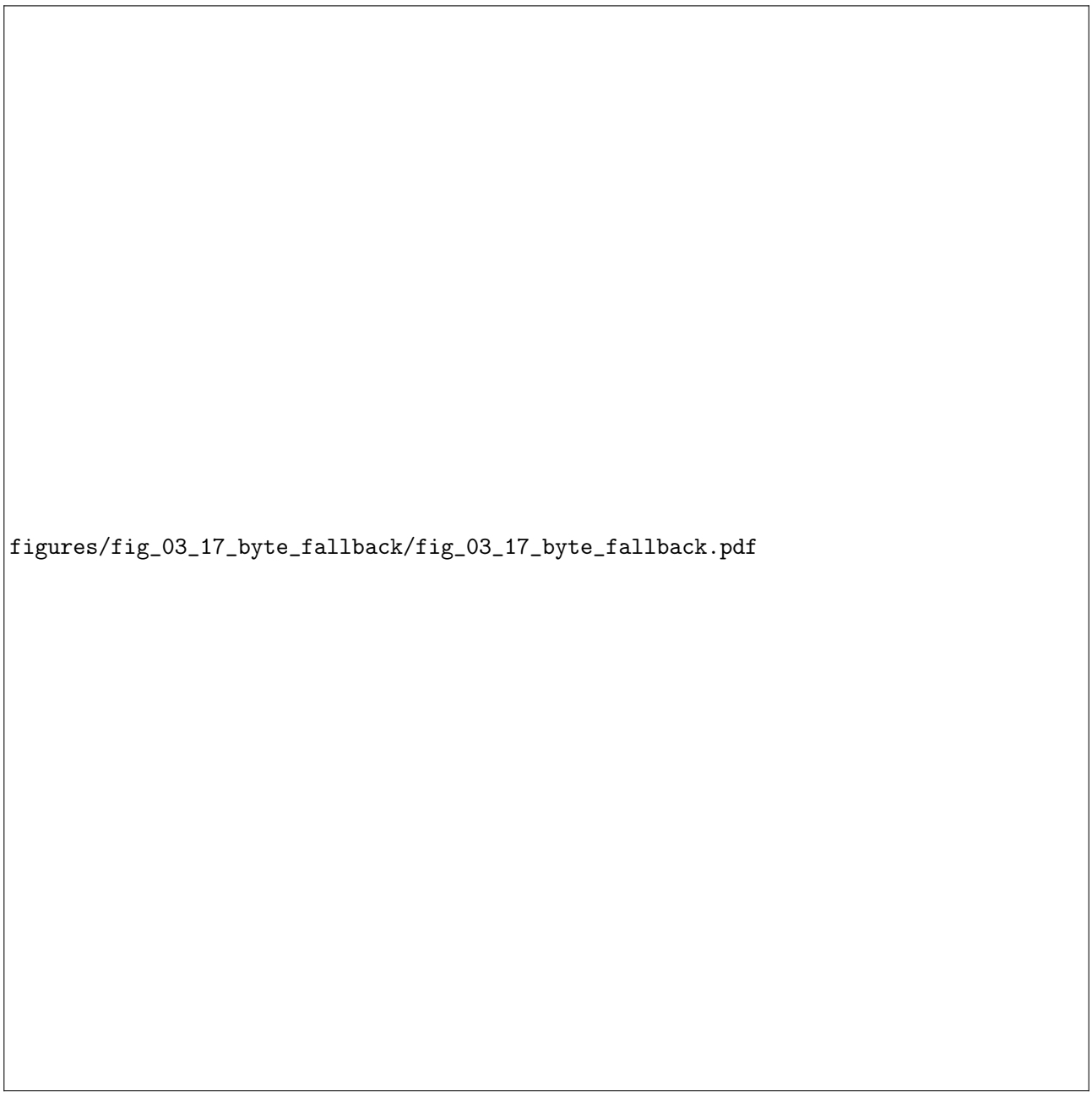


Figure 3.17: When a character is not in the learned vocabulary, byte-level fallback represents it as a sequence of byte tokens corresponding to its UTF-8 encoding. This guarantees that any valid Unicode input can be tokenized without loss of information.

Figure 3.18: Compression ratio (bytes per token) increases with vocabulary size as more text is represented by single tokens rather than character sequences. The relationship between vocabulary size and compression ratio follows diminishing returns.

3.7 Context Representation in Tokenized Language

How This Chapter Represents Context
The fundamental question in language modeling is: How do we represent the context $w_1, \ldots, w_{t-1}$ to predict w_t?

- **Context representation**: Tokenization converts raw text into a sequence of discrete token IDs from a fixed vocabulary $\mathcal{V}$
- **Context encoding**: Each token receives a unique integer index, creating sparse one-hot representations that subsequent layers transform
- **Limitation**: Subword tokens may split meaningful units, requiring models to learn to compose meaning across token boundaries
- **Next chapter preview**: Word embeddings will transform these sparse token IDs into dense vector representations that capture semantic similarity

Tokenization fundamentally shapes how language models represent context for next-token prediction. When we compute $P(w_t \mid w_1, \ldots, w_{t-1})$, the sequence $w_1, \ldots, w_{t-1}$ is not raw text but a sequence of token IDs produced by the tokenizer, creating a discrete symbolic representation of the continuous stream of language. Each token ID is an integer that indexes into the vocabulary $\mathcal{V}$, and the model's first layer converts these integers into dense vector representations through an embedding lookup. The choice of tokenization determines what units compose this context: a word-level tokenizer might represent "The cat sat" as three token IDs, while a BPE tokenizer might produce four or five IDs depending on how it segments each word. The model sees only these token IDs and must learn to compose meaning from whatever units the tokenizer provides, without access to the original character-level text. This means that the tokenizer's decisions about where to place boundaries become baked into the model's understanding of language structure, influencing everything from how the model represents concepts to how it handles novel words.

The implications for prediction are profound, affecting how uncertainty is distributed across token positions. Consider predicting the next token after "un" in isolation versus after "unhappi". In the first case, the model must consider all words beginning with "un"—a vast space including "under", "until", "unusual", and thousands more possible continuations. In the second case, the likely continuations are much more constrained: "ness", "ly", "er". The tokenizer's decision to split "unhappiness" at particular boundaries affects how much information about the eventual word is available to the model at each prediction step. More generally, subword tokenization creates a kind of hierarchical structure where the model first predicts coarse-grained units (common subwords, whole words) and then refines within those units (completions of rare words). This hierarchical structure emerges automatically from the tokenization without being explicitly designed into the model architecture, allowing the model to allocate prediction capacity efficiently across common and rare word forms.

One subtle consequence of subword tokenization is that the model's notion of "word" becomes fuzzy. When predicting after "The capital of France is", a word-level model clearly predicts a single token "Paris". A subword model might predict "Par" followed by "is", or might have learned "Paris" as a single token depending on its frequency in training data. The probability $P(\text{Paris})$ is no longer a simple lookup but might be computed as $P(\text{Par}) \times P(\text{is} \mid \text{Par})$ or accessed directly as $P(\text{Paris})$ depending on tokenization. This makes comparing probabilities across tokenizers subtle: a model's reported perplexity depends on how many tokens it must predict, which varies with tokenization choices. Researchers must be careful to use consistent tokenization when comparing models or to normalize metrics appropriately. The next chapter will explore how embedding layers transform these discrete token IDs into continuous vector representations, providing the foundation for neural language models to learn rich semantic relationships between tokens.

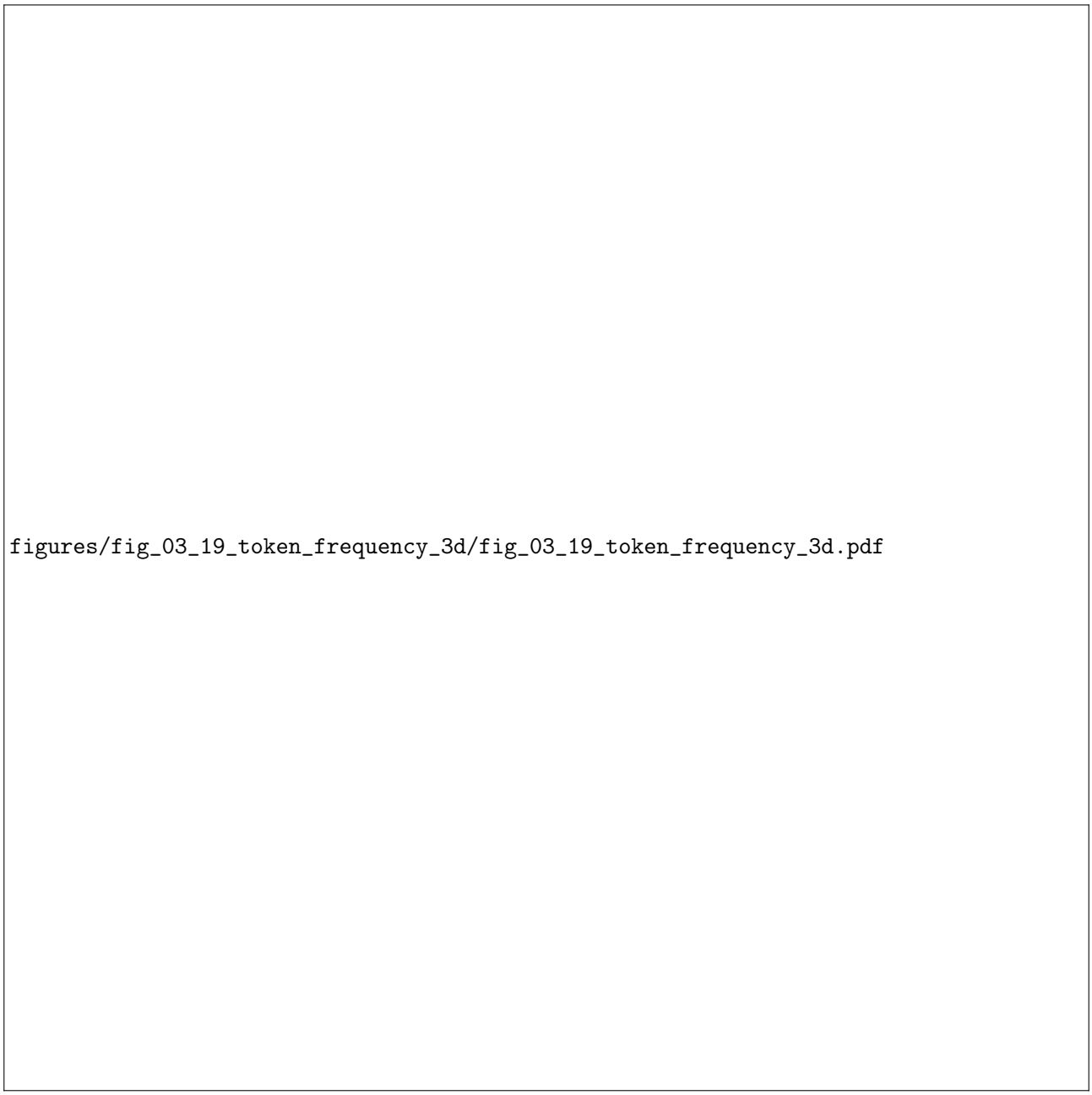


Figure 3.19: A three-dimensional view of token frequencies in a trained vocabulary shows the heavy-tailed distribution: a small number of tokens account for most occurrences, while the majority of vocabulary entries are used rarely.

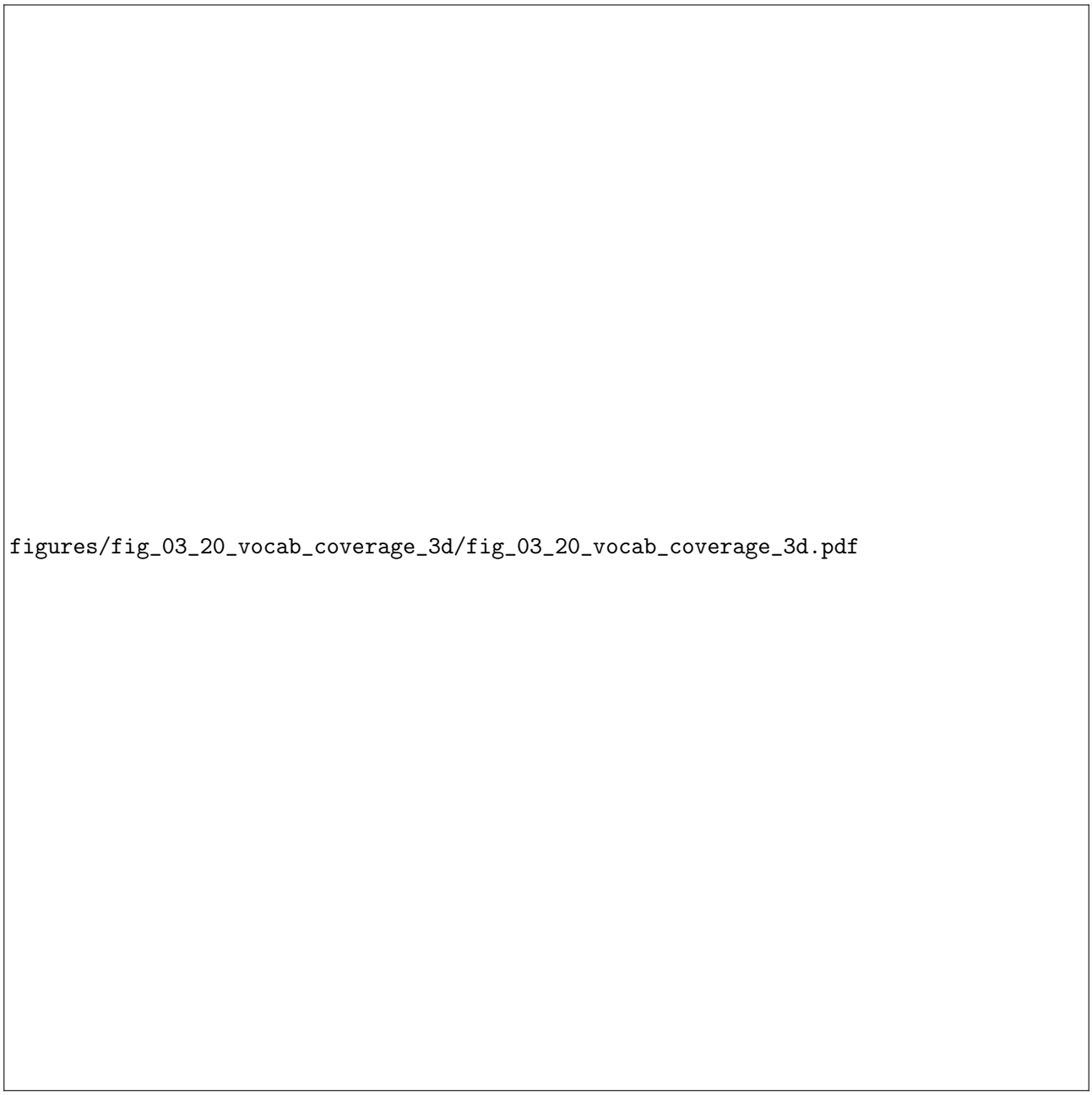


Figure 3.20: A three-dimensional surface showing how vocabulary coverage varies with vocabulary size and language characteristics reveals the diminishing returns of larger vocabularies and the significant differences between languages.

Figure 3.21: The dynamics of BPE merging visualized as a three-dimensional surface shows how vocabulary composition changes as more merge operations are performed, transitioning from character-dominated to subword-dominated to word-dominated regions.

Figure 3.22: A three-dimensional comparison of BPE, WordPiece, and Unigram tokenization across vocabulary size, fertility, and downstream performance illustrates the trade-off space that practitioners must navigate.

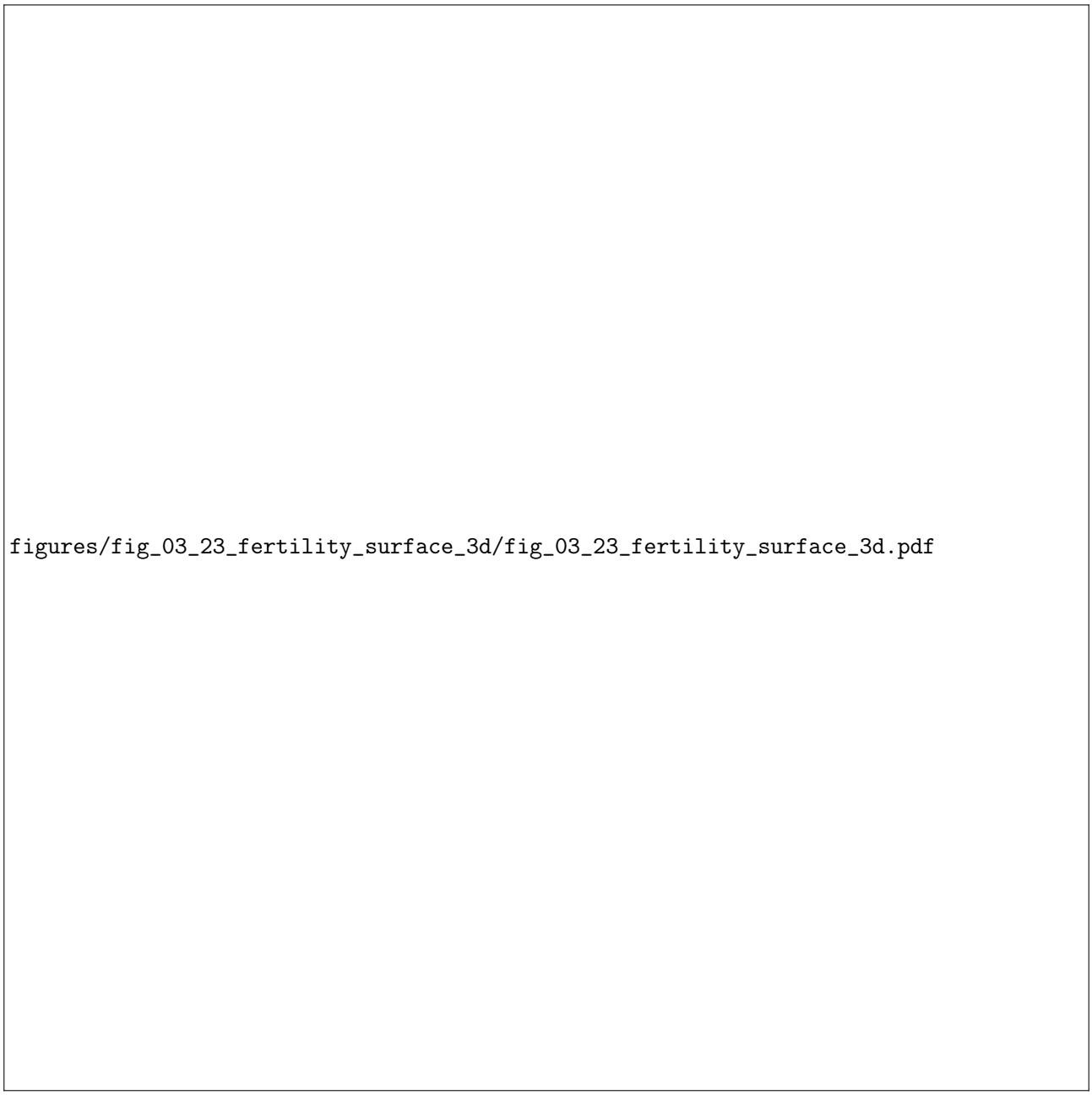


Figure 3.23: Fertility varies as a three-dimensional surface over vocabulary size and language type. The surface reveals how different languages respond to increasing vocabulary size and helps identify optimal vocabulary sizes for multilingual models.

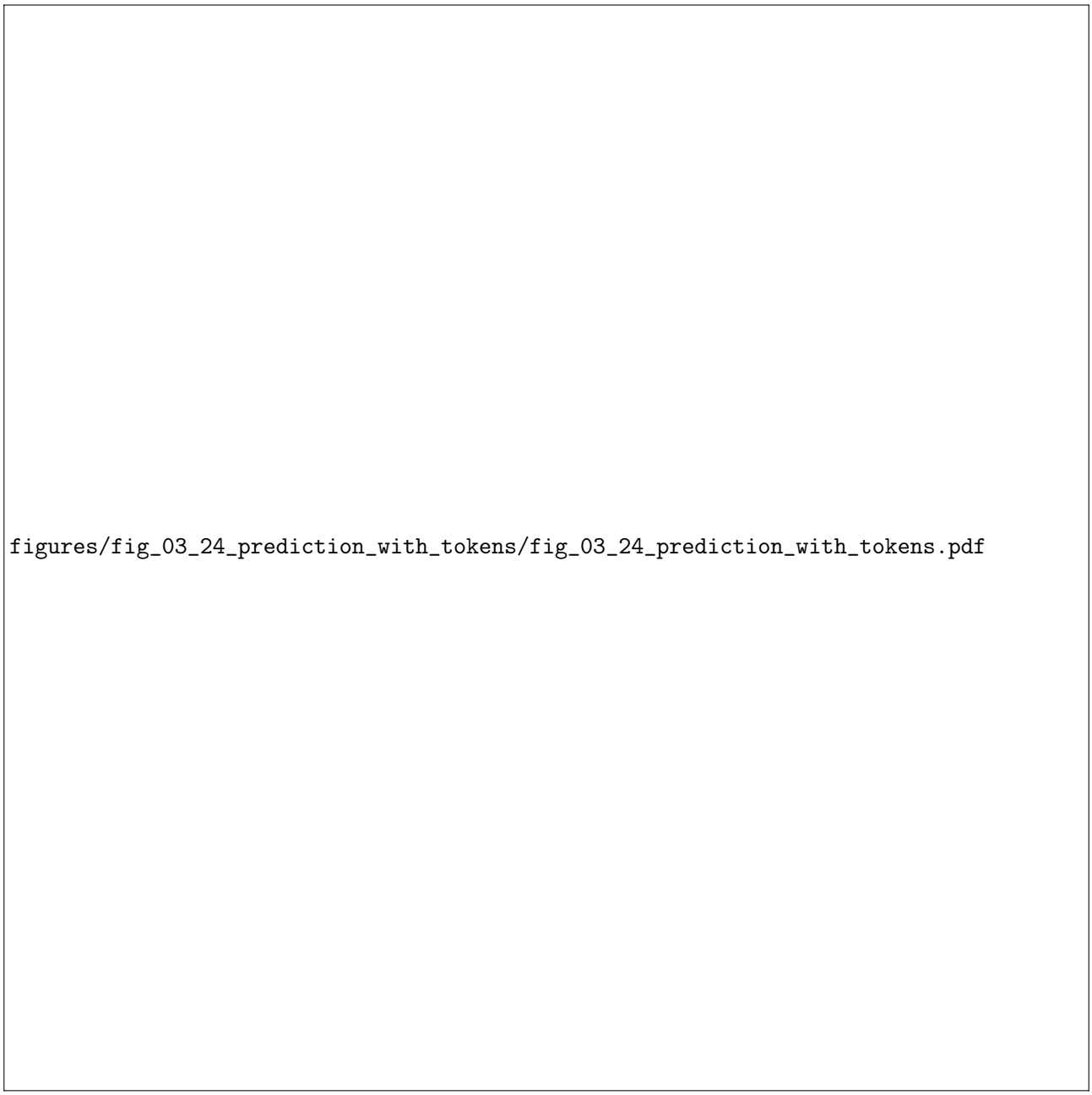


Figure 3.24: Next-token prediction with subword tokenization may require multiple prediction steps to generate a single word. The probability of the complete word is the product of conditional probabilities at each step.

Figure 3.25: The journey from raw text to context representation: tokenization produces token IDs, embedding lookup converts IDs to vectors, and subsequent model layers transform these vectors into rich contextual representations that inform next-token prediction.

3.8 Summary

This chapter has explored tokenization as the critical preprocessing step that defines what units language models predict. We began by examining word-level tokenization, which aligns with human intuition but suffers from out-of-vocabulary problems and vocabulary explosion in morphologically rich languages. Character-level and byte-level approaches solve the coverage problem but create impractically long sequences that strain computational resources and make it harder for models to capture long-range dependencies. Subword tokenization emerged as the dominant solution, with BPE, WordPiece, SentencePiece, and unigram language model methods each offering slightly different approaches to learning vocabularies that balance sequence length against vocabulary size. We examined how vocabulary size affects the embedding matrix, fertility, and downstream performance, and how special tokens provide structural markers beyond simple text representation. The evaluation of tokenization quality through fertility, compression ratio, coverage, and downstream task performance guides practical decisions about tokenizer design. Finally, we explored how tokenization shapes context representation, creating a mapping from raw text to discrete token IDs that subsequent model layers transform into rich contextual representations for prediction.

We can now predict better because:

- Subword tokenization eliminates out-of-vocabulary words while maintaining reasonable vocabulary sizes
- BPE and WordPiece learn data-driven vocabularies that balance word frequency and morphological compositionality
- Byte-level fallback mechanisms ensure complete coverage of any Unicode text, including emoji and rare scripts
- Special tokens provide explicit structure for tasks beyond simple next-word prediction, enabling masked language modeling and sequence classification
- Vocabulary size becomes a tunable hyperparameter that trades off between model capacity and sequence length efficiency

Next: Chapter **??** transforms these discrete token IDs into continuous vector representations, enabling models to capture semantic similarity between words that tokenization treats as completely independent symbols.

Exercises

1. **Word-Level Vocabulary Analysis.** Given a corpus of 1 million words with 50,000 unique word types, calculate what percentage of word types appear fewer than 5 times. Explain why this creates problems for word-level tokenization and how it motivates subword approaches.

2. **BPE Merge Steps.** Starting with the character vocabulary {a, b, c, d, e, s, t, _} and the corpus "abracadabra", perform 3 iterations of the BPE merge algorithm. Show the vocabulary after each merge and explain why each pair was selected.

3. **Vocabulary Size Trade-offs.** A language model uses vocabulary size $|\mathcal{V}| = 32,000$. If we double the vocabulary to 64,000, explain how this affects: (a) the embedding matrix size, (b) average tokens per sentence, and (c) the model's ability to handle rare words.

4. **Fertility Calculation.** Given a tokenizer that produces the following tokenizations: "unhappiness" $\rightarrow$ ["un", "happiness"], "internationalization" $\rightarrow$ ["international", "ization"], calculate the fertility for each word and the average fertility across both examples.

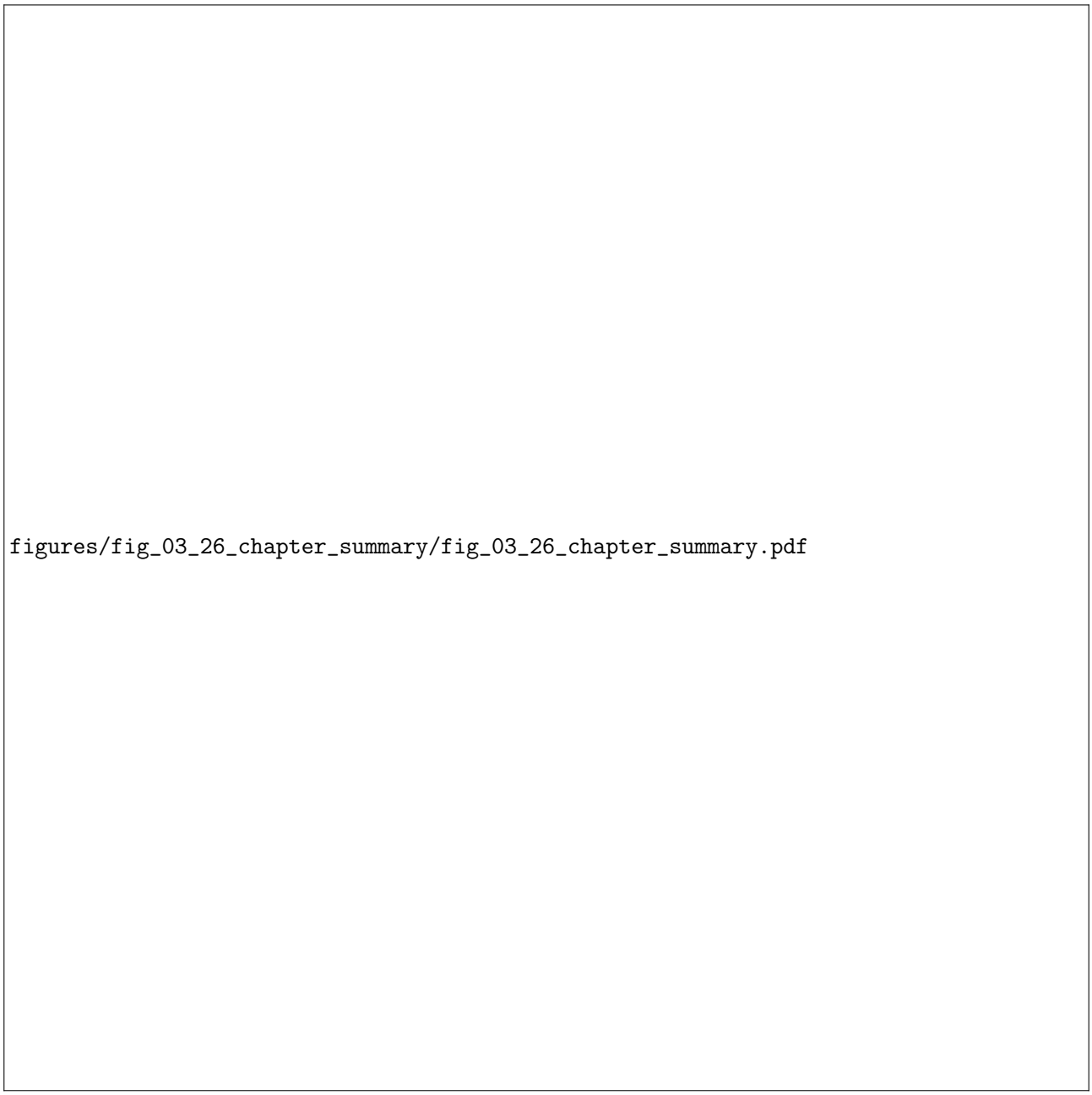


Figure 3.26: Chapter 3 summary: Tokenization converts raw text into discrete token IDs that define the vocabulary $\mathscr{V}$ over which language models predict. Subword algorithms balance vocabulary size against sequence length, while special tokens provide structural markers for tasks beyond simple language modeling.

5. **Special Tokens.** Explain the purpose of each special token: [CLS], [SEP], [MASK], [PAD], [UNK]. For each, describe a specific scenario where it is essential for model training or inference.

6. **Byte-Level Fallback.** The Unicode character for the emoji "heart" has UTF-8 encoding. Explain how a byte-level tokenizer handles this character when it is not in the learned vocabulary, and why this guarantees no out-of-vocabulary tokens.

7. **WordPiece vs BPE.** Compare how BPE and WordPiece would tokenize the word "unbreakable" if both have learned prefixes "un" and "able" but neither has learned "break" as a single token. Explain the key algorithmic difference between the two methods.

8. **Multilingual Tokenization.** A SentencePiece model trained on English text is applied to German text containing the word "Donaudampfschifffahrtsgesellschaft". Predict how this compound word might be tokenized and explain why language-specific training matters for tokenizer quality.

9. **Sequence Length Impact.** A character-level model processes the sentence "The quick brown fox" (19 characters) while a BPE model tokenizes it into 5 tokens. Calculate the ratio of computational cost (assuming self-attention with $O(n^2)$ complexity) and discuss the implications for training efficiency.

10. **Tokenization and Prediction.** Consider predicting the next token after "The capital of France is". Compare how a word-level, character-level, and subword tokenizer would frame this prediction task. Which representation makes the prediction easiest for the model?

11. **Research Question: Optimal Vocabulary Size.** Design an experiment to determine the optimal vocabulary size for a specific language and domain. What metrics would you use to evaluate tokenizer quality, and how would you balance compression ratio against downstream task performance?

Bibliography

Taku Kudo and John Richardson. Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. *arXiv preprint arXiv:1808.06226*, 2018.

Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*, 2016.

Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

Index

Predicting the Next Word

From Shannon to ChatGPT

Test Compilation - Chapter 4

Contents

Chapter 4

Word Embeddings: Distributed Representations of Meaning

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

- Moving from discrete word symbols to continuous vector representations
- Learning semantic similarity through distributional statistics
- Capturing syntactic and semantic relationships in geometric space
- Reducing model parameters while improving generalization

4.1 From Discrete Symbols to Continuous Vectors

The n-gram models examined in Chapter **??** treated words as atomic, discrete symbols drawn from vocabulary $\mathcal{V}$. Each word $w_i \in \mathcal{V}$ was represented as a distinct integer index, and the model learned separate probability distributions $P(w_{t+1} \mid w_t, w_{t-1}, \ldots)$ for each observed context tuple. This discrete representation suffers from a fundamental limitation: it encodes no notion of similarity between words. The model has no mechanism to recognize that "cat" and "feline" are semantically related, or that "running" and "ran" share morphological structure. Consequently, if the training corpus contains "the cat sat on the mat" but not "the feline sat on the mat", the model cannot generalize to predict the latter, despite their semantic equivalence. This sparsity problem becomes severe as vocabulary size grows: with $|\mathcal{V}| = 50{,}000$ words and trigrams, the number of possible contexts exceeds $50{,}000^2 = 2.5$ billion, yet most will never appear in any corpus. The curse of dimensionality renders exhaustive enumeration infeasible. This chapter introduces word embeddings, which map discrete symbols into continuous vector space $\mathbb{R}^d$ where semantic similarity corresponds to geometric proximity. By representing words as dense vectors rather than atomic symbols, we enable models to generalize across similar contexts and dramatically reduce the parameter space required for accurate next-word prediction.

4.1.1 The Discrete Symbol Problem

Consider the one-hot encoding representation used implicitly in n-gram models, where each word $w_i \in \mathcal{V}$ is mapped to a binary vector $\mathbf{1}_{w_i} \in \{0,1\}^{|\mathcal{V}|}$ with a single 1 at position i and zeros elsewhere: $\mathbf{1}_{w_i}[j] = 1$ if $j = i$ and 0 otherwise for $j = 1, \ldots, |\mathcal{V}|$. This encoding has severe limitations that fundamentally constrain language modeling. First, all words are mutually orthogonal: the dot product between any two distinct one-hot vectors is zero, $\mathbf{1}_{w_i} \cdot \mathbf{1}_{w_j} = \delta_{ij}$ where δ_{ij} is the Kronecker delta, meaning cosine similarity between distinct words is always zero, indicating no relationship whatsoever between semantically related words like "cat" and "feline", or between "king" and "queen". Second, the dimensionality equals vocabulary size $|\mathcal{V}|$, which typically ranges from 30,000 to 100,000 for modern systems, with each vector containing exactly one non-zero

entry yielding extreme sparsity of approximately 0.002%. Third, these high-dimensional sparse vectors cannot be processed efficiently by neural networks, which require dense low-dimensional representations for gradient-based optimization. Fourth, the representation is fundamentally incapable of capturing distributional semantics: words that appear in similar contexts receive completely unrelated encodings. As vocabulary grows with corpus size, the one-hot representation becomes increasingly impractical, motivating the distributed representations we now develop.

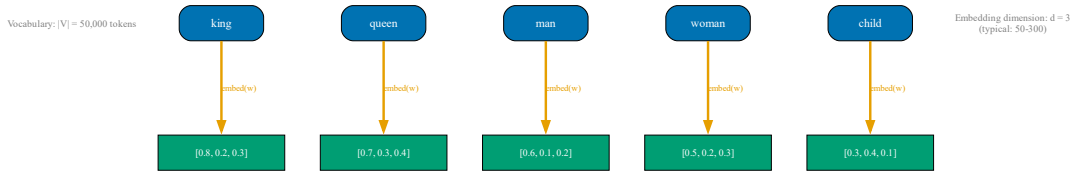


Figure 4.1: One-hot encoding visualization. Panel A shows sparse binary vectors for "cat", "dog", and "car" with $|\mathcal{V}| = 10$. Panel B displays the resulting dot product matrix, which is the identity matrix indicating no similarity between any words. Panel C illustrates that semantic relationships are not captured: related words (cat, dog) are as distant as unrelated words (cat, car). Panel D demonstrates the dimensionality problem: with $|\mathcal{V}| = 50{,}000$, each vector requires 50,000 dimensions with only one non-zero entry, creating severe computational and statistical inefficiency.

4.1.2 The Distributed Representation Hypothesis

The solution to the discrete symbol problem lies in distributed representations, where each word is encoded as a dense vector $\mathbf{e}_w \in \mathbb{R}^d$ with $d \ll |\mathcal{V}|$, typically $d \in [100, 1000]$. Unlike one-hot encoding, where a single dimension encodes word identity, distributed representations spread information across all dimensions. This distribution enables continuous similarity: words with similar meanings map to nearby points in $\mathbb{R}^d$, measured by cosine similarity $\text{sim}(w_i, w_j) = \frac{\mathbf{e}_i \cdot \mathbf{e}_j}{\|\mathbf{e}_i\| \|\mathbf{e}_j\|}$ or Euclidean distance $\|\mathbf{e}_i - \mathbf{e}_j\|$. The dimensionality reduction from $|\mathcal{V}| \approx 50{,}000$ to $d \approx 300$ yields a 99.4% parameter reduction while preserving the information necessary for prediction. The distributed hypothesis posits that all dimensions participate in representing each concept, with semantic features encoded as patterns of activation across the vector. For example, the dimension vector might encode features such as animacy, concreteness, sentiment polarity, or syntactic category, though these dimensions are learned implicitly during training rather than specified a priori. The key mathematical property is that the embedding function $\mathbf{e} : \mathcal{V} \to \mathbb{R}^d$ preserves semantic structure: if words w_i and w_j are semantically similar, then $\|\mathbf{e}(w_i) - \mathbf{e}(w_j)\|$ should be small. This geometric encoding of meaning enables generalization, as models can interpolate between known examples in continuous space rather than memorizing discrete symbolic associations. The embedding matrix $\mathbf{E} \in \mathbb{R}^{|\mathcal{V}| \times d}$ stores all word vectors, with row i containing $\mathbf{e}_{w_i}$. For next-word prediction, the context words $w_1, \ldots, w_{t-1}$ are first converted to embeddings $\mathbf{e}_1, \ldots, \mathbf{e}_{t-1}$, then processed by the model architecture to predict w_t.

Figure 4.2: Distributed representations overcome one-hot limitations. Panel A shows dense embedding vectors with dimensionality $d = 300$ where each word is represented by real-valued numbers across all dimensions. Panel B displays a cosine similarity heatmap revealing semantic structure: related words (cat–feline, dog–canine) have high similarity (0.7–0.9), while unrelated words (cat–car) have low similarity (0.1–0.2). Panel C demonstrates dramatic dimensionality reduction from $|\mathcal{V}| = 50{,}000$ to $d = 300$ while preserving semantic relationships. Panel D visualizes geometric relationships in 2D projection: semantically related words cluster together in embedding space, with distance corresponding to semantic dissimilarity.

4.1.3 Historical Context

The idea that word meaning derives from distributional context traces to Harris (1954), who proposed that "words which occur in the same contexts tend to have similar meanings", and Firth (1957), who famously stated "you shall know a word by the company it keeps". These linguistic insights inspired computational methods in the 1990s, beginning with Latent Semantic Analysis (LSA), which applied singular value decomposition to word-document co-occurrence matrices to obtain low-dimensional representations. While LSA successfully captured semantic similarity for information retrieval tasks, its reliance on linear algebra at document scale proved computationally expensive and unable to capture fine-grained syntactic or semantic relationships. The modern era of word embeddings began with Bengio et al. (2003) [Bengio et al., 2003], who introduced the neural probabilistic language model that learned distributed representations as parameters during next-word prediction training. Their key insight was to jointly optimize embedding vectors and prediction weights using backpropagation, allowing the model to discover representations specifically suited for prediction rather than general co-occurrence patterns. However, Bengio's model required substantial computational resources, limiting adoption. The breakthrough came in 2013 with Mikolov et al.'s Word2Vec [Mikolov et al., 2013], which introduced highly efficient training algorithms (Skip-gram and CBOW) that could learn high-quality embeddings from billion-word corpora in hours rather than weeks. Concurrently, Pennington et al. (2014) developed GloVe [Pennington et al., 2014], which combined the benefits of global matrix factorization and local context windows. These methods democratized word embeddings, making them accessible to researchers without supercomputing resources. By 2015, pre-trained embeddings became standard initialization for virtually all natural language processing tasks. The subsequent shift to contextual embeddings (ELMo, BERT) beginning in 2018 addressed the polysemy limitation we discuss in Section 4.5.3, but the static embeddings developed in this chapter remain fundamental to understanding modern architectures covered in Chapter **??**.

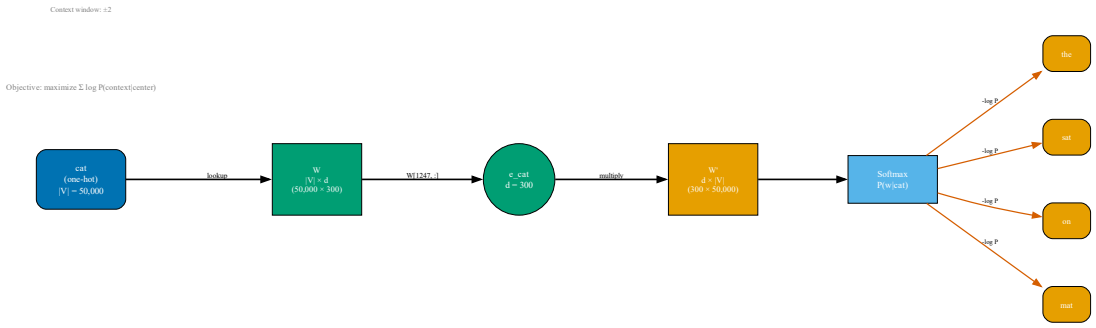


Figure 4.3: Historical development of word embeddings from 1954 to 2018. Key milestones include Harris's distributional hypothesis (1954), LSA dimensionality reduction via SVD (1990), Bengio's neural language model with learned embeddings (2003), Mikolov's Word2Vec bringing embeddings to scale (2013), Pennington's GloVe combining count-based and prediction-based approaches (2014), and the transition to contextual embeddings with ELMo and BERT (2018, covered in Chapter 6). Perplexity improvements over time demonstrate quantitative advances: from 250 (Bengio 2003) to 150 (Word2Vec 2013) to 100 (contextual embeddings 2018) on standard benchmarks.

4.2 The Distributional Hypothesis

The distributional hypothesis provides the theoretical foundation for learning word representations from unlabeled text. Formally stated: words that occur in similar contexts tend to have similar meanings. This linguistic principle translates into a computational objective: construct embeddings such that semantic similarity is reflected by geometric proximity in $\mathbb{R}^d$. The hypothesis operates on context windows, which define the local neighborhood around each word occurrence. Given a sentence and target word at position t, the context con-

sists of words within distance k: $\{w_{t-k},\ldots,w_{t-1},w_{t+1},\ldots,w_{t+k}\}$. For example, in "the cat sat on the mat" with target "sat" and $k=2$, the context is $\{\text{cat},\text{on},\text{the}\}$ (excluding the target itself). By collecting contexts for each word across a large corpus $\mathscr{D}$, we obtain distributional statistics that characterize word usage. Words appearing in similar contexts—such as "cat" and "feline", which both appear after "the" and before action verbs—should receive similar embeddings. Conversely, words appearing in disjoint contexts should receive distant embeddings. This section formalizes distributional semantics through co-occurrence matrices and pointwise mutual information (PMI), which quantify the association between word pairs. We then examine matrix factorization approaches that directly decompose co-occurrence statistics to obtain embeddings, providing intuition for the prediction-based methods in subsequent sections. The key insight is that distributional patterns in raw text provide sufficient signal to learn semantic representations without any labeled data, enabling self-supervised learning from massive corpora.

4.2.1 Context Windows and Co-occurrence

To operationalize the distributional hypothesis, we define a context window of size k around each word in the corpus. Given a text sequence $w_1,w_2,\ldots,w_T$, for each target word w_t we extract the context $\{w_{t-k},\ldots,w_{t-1},w_{t+1},\ldots,w_{t+k}\}$, excluding w_t, which may be symmetric (both directions) for general embeddings or asymmetric (left-only) for causal language modeling. We record co-occurrence whenever a word appears within the context window of another: formally, the co-occurrence count $C_{ij}=\sum_{t=1}^{T}\mathbb{1}[w_t=i]\sum_{k=1}^{K}(\mathbb{1}[w_{t+k}=j]+\mathbb{1}[w_{t-k}=j])$ where $\mathbb{1}[\cdot]$ is the indicator function, counting how many times word j appears within K positions of word i across the corpus. The co-occurrence matrix $\mathbf{C}\in\mathbb{R}^{|\mathscr{V}|\times|\mathscr{V}|}$ stores all pairwise counts and exhibits several properties: symmetry if windows are symmetric ($C_{ij}=C_{ji}$), sparsity because most word pairs never co-occur, and heavy skew with frequent function words dominating. Raw counts favor frequent words, so we apply statistical normalization via pointwise mutual information (PMI), which measures how much more often words i and j co-occur than expected under independence: $\text{PMI}(w_i,w_j)=\log\frac{C_{ij}\cdot N}{\sum_k C_{ik}\cdot\sum_k C_{kj}}$ where N is the total count. PMI is positive when words co-occur more than chance, negative when less, and zero when independent; since negative values are often unreliable due to insufficient data, we use positive PMI (PPMI): $\text{PPMI}(w_i,w_j)=\max(0,\text{PMI}(w_i,w_j))$, which zeros out negatives and better reflects semantic associations.

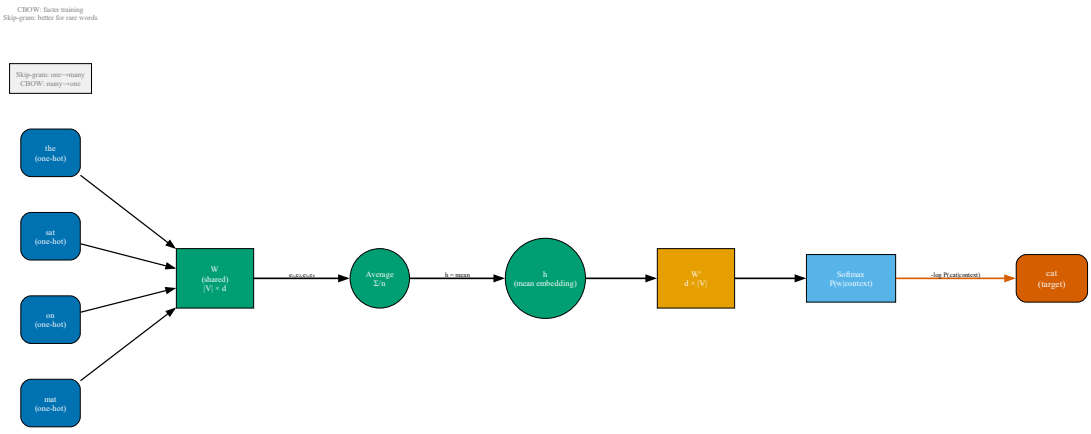


Figure 4.4: Context windows and co-occurrence statistics. Panel A illustrates context window extraction with $k=2$ around target word "cat" in the sentence "the black cat sat there", yielding context $\{\text{the},\text{black},\text{sat},\text{there}\}$. Panel B shows how co-occurrence counts are accumulated across the corpus: each occurrence of "cat" contributes to counts with its context words. Panel C displays a raw co-occurrence matrix as a log-scale heatmap, revealing high counts for frequent words and sparsity for rare words. Panel D shows the PPMI-transformed matrix, which is sparser but exhibits clearer semantic structure: semantically related word pairs (cat–feline, dog–canine) have high PPMI, while unrelated pairs have PPMI near zero.

4.2.2 Matrix Factorization Perspective

Given a co-occurrence or PPMI matrix, we can obtain word embeddings through matrix factorization, approximating the high-dimensional sparse matrix $\mathbf{C} \in \mathbb{R}^{|\mathcal{V}|\times|\mathcal{V}|}$ by a low-rank product of dense matrices. Singular value decomposition (SVD) provides an optimal low-rank approximation: $\mathbf{C} = \mathbf{U}\Sigma\mathbf{V}^\top$ where $\mathbf{U}$ and $\mathbf{V}$ are orthogonal matrices of singular vectors and Σ is a diagonal matrix of singular values $\sigma_1 \geq \sigma_2 \geq \cdots \geq 0$. To obtain a d-dimensional representation with $d \ll |\mathcal{V}|$, we perform truncated SVD keeping only the top d singular values: $\mathbf{C} \approx \mathbf{U}_d\Sigma_d\mathbf{V}_d^\top$, and the word embedding matrix is taken as $\mathbf{E} = \mathbf{U}_d\Sigma_d^{1/2} \in \mathbb{R}^{|\mathcal{V}|\times d}$, where the square root distributes singular values symmetrically between rows and columns. This approach, known as Latent Semantic Analysis when applied to word-document matrices, reduces dimensionality while preserving dominant co-occurrence patterns: words that frequently co-occur with similar contexts have similar embeddings because they project onto similar combinations of the top singular vectors. The truncated SVD minimizes reconstruction error $\|\mathbf{C} - \mathbf{U}_d\Sigma_d\mathbf{V}_d^\top\|_F$ among all rank-d approximations, making it the optimal linear dimensionality reduction. However, this factorization approach has computational limitations: computing SVD for a $|\mathcal{V}| \times |\mathcal{V}|$ matrix with $|\mathcal{V}| = 50{,}000$ requires $O(|\mathcal{V}|^3)$ operations and $O(|\mathcal{V}|^2)$ memory, which becomes prohibitive at scale. Modern embedding methods overcome these limitations through prediction-based training that avoids explicitly constructing the co-occurrence matrix.

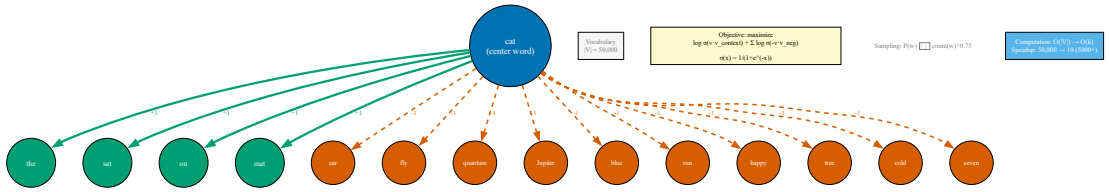


Figure 4.5: SVD decomposition visualized in 3D. The original co-occurrence space is $|\mathcal{V}|$-dimensional (shown as a high-dimensional point cloud in the upper region), with most words lying on a lower-dimensional manifold. Truncated SVD projects onto the top d principal components (shown as a 2D plane in the lower region), preserving the dominant variance while reducing dimensionality by 99%. Semantic clusters (animals, colors, verbs) remain separated in the reduced space, demonstrating that co-occurrence patterns encode semantic structure that survives dimensionality reduction.

4.2.3 Semantic Similarity from Distribution

The distributional hypothesis predicts that words appearing in similar contexts should have similar embeddings, and this prediction is empirically validated: embeddings learned from co-occurrence statistics exhibit strong semantic clustering, with synonyms such as “cat” and “feline” appearing in nearly identical contexts and yielding cosine similarity typically above 0.7 in learned embeddings. Hypernyms and hyponyms also cluster: “animal”, “mammal”, “cat”, “feline” form a hierarchy with progressively higher similarity at finer levels. Interestingly, antonyms often receive similar embeddings because they appear in similar contexts: both “hot” and “cold” modify “weather”, “water”, “temperature”, despite having opposite meanings, revealing a limitation of purely distributional semantics where context determines syntactic positions but does not uniquely determine semantic content. Syntactic categories also emerge from distributional patterns: verbs cluster because they appear in similar structural positions (after subjects, before objects), nouns cluster because they appear after determiners and before verbs, and adjectives cluster because they appear before nouns. To quantify similarity, we use cosine similarity $\text{sim}(w_i, w_j) = \frac{\mathbf{e}_i \cdot \mathbf{e}_j}{\|\mathbf{e}_i\|\|\mathbf{e}_j\|}$, which ranges from -1 (opposite) through 0 (orthogonal) to $+1$ (identical direction) and is preferred over Euclidean distance because it is invariant to vector magnitude, focusing only on direction in embedding space. Given a query word, we can retrieve its nearest neighbors by ranking all other words by cosine similarity, producing semantically related terms useful for query expansion, synonym detection, and lexical semantics research, validating that unsupervised learning from raw text can discover semantic structure without labeled data.

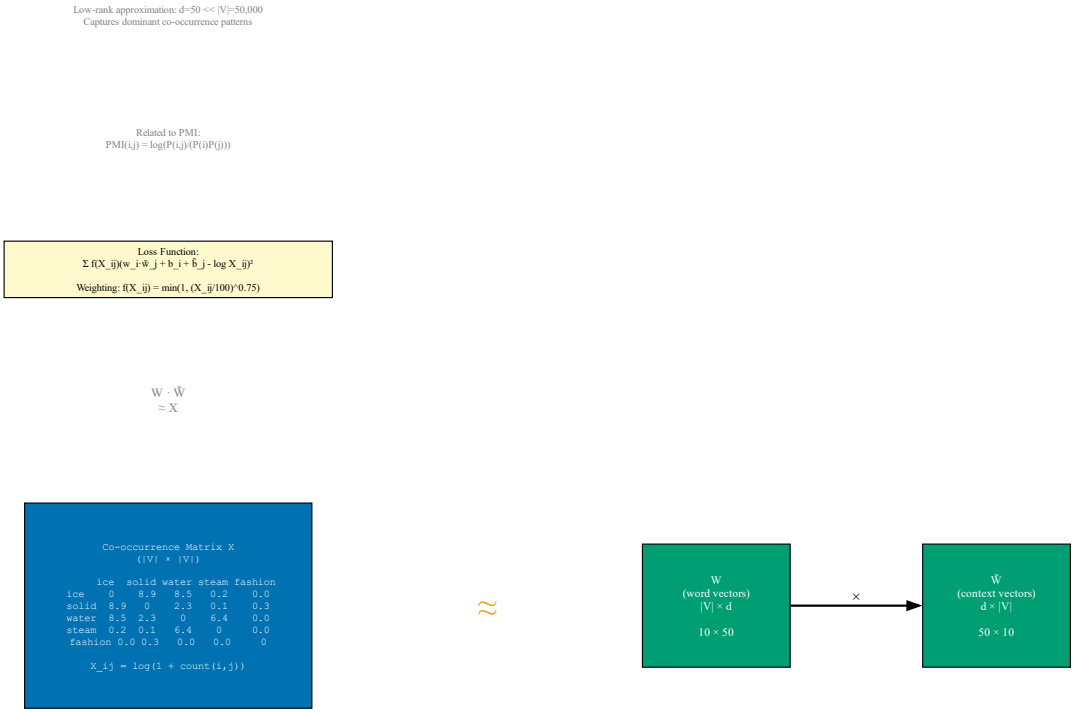


Figure 4.6: Semantic similarity patterns in learned embeddings. Panel A shows a t-SNE projection of 500 words into 2D space, revealing clear semantic clusters. Panel B highlights specific clusters: animals (cat, dog, horse), colors (red, blue, green), and verbs (run, walk, jump) group together without any explicit supervision. Panel C lists nearest neighbors for "cat" by cosine similarity: feline (0.82), kitten (0.76), dog (0.71), puppy (0.68), demonstrating that synonyms and related concepts are nearby in embedding space. Panel D displays a similarity heatmap for 20 selected words, showing high within-cluster similarity (dark) and low between-cluster similarity (light).

4.3 Word2Vec: Skip-gram and CBOW

While matrix factorization methods provide intuitive connections to co-occurrence statistics, they suffer from computational and memory limitations at scale. Mikolov et al. (2013) introduced Word2Vec, a family of efficient prediction-based algorithms that learn embeddings by training shallow neural networks to predict context from words (Skip-gram) or words from context (CBOW). These methods avoid explicitly constructing the co-occurrence matrix, instead processing the corpus in a streaming fashion via stochastic gradient descent. The key innovation is framing embedding learning as a prediction task: given a target word w_t, predict its surrounding context words $w_{t-k}, \ldots, w_{t+k}$, or vice versa. By maximizing prediction accuracy, the model is forced to learn embeddings that capture distributional semantics. Word2Vec's computational efficiency stems from two architectural choices: shallow networks (single hidden layer with no nonlinearity) and negative sampling to avoid the expensive softmax normalization over the full vocabulary. These optimizations enable training on billion-word corpora in hours on a single machine, democratizing access to high-quality embeddings. The self-supervised nature of the training objective—no labels required, only raw text—allows leveraging massive unlabeled corpora. Word2Vec embeddings exhibit remarkable properties including linear substructures (Section 4.5) and strong performance on downstream tasks. While modern contextual embeddings have superseded static Word2Vec for many applications, understanding Word2Vec remains essential for two reasons: it introduces the prediction-based paradigm that underlies all subsequent neural language models, and its embeddings serve as interpretable baseline representations for analyzing semantic spaces.

4.3.1 The Word2Vec Framework

Word2Vec employs self-supervised learning: the training signal is derived from the data itself rather than external labels. Given a corpus $\mathscr{D} = [w_1, w_2, \ldots, w_T]$, we slide a context window of size $2K+1$ across the sequence, creating training examples at each position t. For target word w_t, the context is $\{w_{t-K}, \ldots, w_{t-1}, w_{t+1}, \ldots, w_{t+K}\}$. Two architectures are defined: Skip-gram predicts each context word given the target, while CBOW (Continuous Bag-of-Words) predicts the target given the context. Both learn two embedding matrices: an input embedding matrix $\mathbf{E}^{\text{in}} \in \mathbb{R}^{|\mathscr{V}| \times d}$ and an output embedding matrix $\mathbf{E}^{\text{out}} \in \mathbb{R}^{|\mathscr{V}| \times d}$, where d is the embedding dimension (typically $d \in [100, 300]$). After training, the input embeddings $\mathbf{E}^{\text{in}}$ are typically used as the final word representations, though some implementations average input and output embeddings. The neural architecture is intentionally simple: a single linear layer with no nonlinearity (making it a log-linear model). For Skip-gram, the forward pass looks up the input embedding for the target word, computes dot products with all output embeddings to produce logits, and applies softmax to obtain a probability distribution over vocabulary. For CBOW, the forward pass averages the input embeddings of context words, then proceeds identically. The loss function is cross-entropy between predicted and observed distributions. This simplicity enables efficient training while capturing distributional semantics: words appearing in similar contexts must have similar embeddings to achieve low loss, enforcing the distributional hypothesis directly through the objective function. The training procedure is stochastic gradient descent over all (target, context) pairs in the corpus, with negative sampling (Section 4.3.4) to make gradient computation tractable.

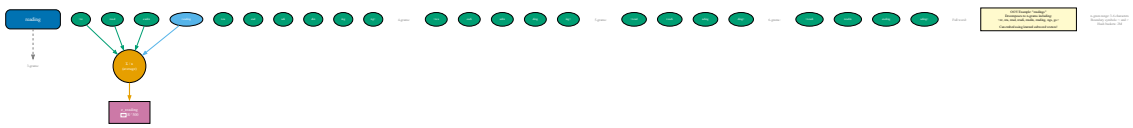


Figure 4.7: Word2Vec framework overview. Panel A illustrates the self-supervised learning setup: given raw text "the cat sat on the mat", context windows automatically generate (target, context) training pairs without requiring manual labels. Panel B compares Skip-gram and CBOW architectures side-by-side: Skip-gram predicts context words from target (one-to-many), while CBOW predicts target from averaged context words (many-to-one). Panel C shows training data generation from a sample corpus, with arrows indicating prediction direction for each architecture. Panel D visualizes embedding space evolution during training: random initialization yields no structure, but after training, semantically related words cluster together.

4.3.2 Skip-gram Architecture

The Skip-gram model predicts context words given a target word, formulating embedding learning as a classification task that can be trained efficiently via stochastic gradient descent. For each position t in the corpus and each context offset $k \in \{-K, \ldots, -1, 1, \ldots, K\}$, we form a training pair (w_t, w_{t+k}) where the model aims to maximize $P(w_{t+k}|w_t)$ for all such pairs, yielding the Skip-gram objective $\mathscr{L}_{\text{skip-gram}} = -\frac{1}{T}\sum_{t=1}^{T}\sum_{-K \le k \le K, k \neq 0} \log P(w_{t+k}|w_t)$ where the probability is modeled using learned embeddings. The architecture is: (1) Look up the input embedding for the target word: $\mathbf{e}_t = \mathbf{E}^{\text{in}}[w_t, :] \in \mathbb{R}^d$. (2) Compute unnormalized scores (logits) for all vocabulary words by taking dot products with output embeddings: $s_v = \mathbf{e}_t \cdot \mathbf{E}^{\text{out}}[v, :]$ for all $v \in \mathscr{V}$. (3) Apply softmax to obtain a probability distribution: $P(w_c|w_t) = \frac{\exp(s_c)}{\sum_{v \in \mathscr{V}} \exp(s_v)}$. (4) Maximize log-probability of observed context word $w_c = w_{t+k}$: $\log P(w_c|w_t) = s_c - \log \sum_{v \in \mathscr{V}} \exp(s_v)$. The gradient with respect to embeddings is computed via backpropagation and the parameters are updated using stochastic gradient descent. Skip-gram generates $2K$ training examples per word occurrence (one for each context position), providing rich supervision for rare words. For example, the sentence "the cat sat" with $K = 1$ generates Skip-gram pairs: (cat, the), (cat, sat), (sat, cat), (the, cat). This one-to-many prediction means Skip-gram trains slower than CBOW but often achieves better representations for infrequent words because each occurrence generates multiple gradient updates. The learned embeddings satisfy the property that words appearing in similar contexts receive similar input embeddings, because the model can only achieve low loss by assigning similar

probabilities to similar contexts, which requires similar input embeddings given the shared output embedding matrix.

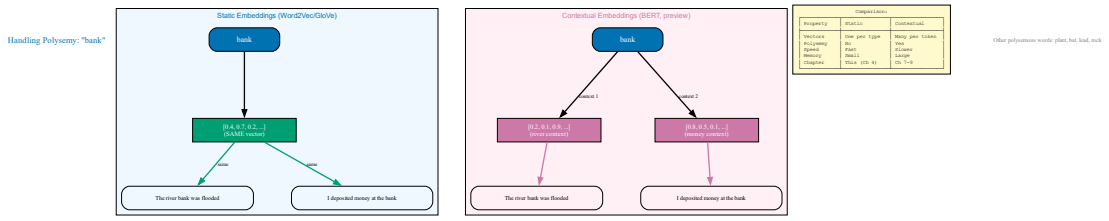


Figure 4.8: Skip-gram architecture and training. Panel A displays the network diagram: target word "cat" is embedded via $\mathbf{E}^{\text{in}}$, then scored against all output embeddings in $\mathbf{E}^{\text{out}}$ to predict context words. Panel B shows example training pairs generated from "the cat sat on the mat" with $K = 2$: target "cat" generates four pairs: (cat, the), (cat, sat), (cat, on), (cat, mat). Panel C illustrates the forward pass computation: embedding lookup, dot products with output embeddings, softmax normalization, and cross-entropy loss. Panel D visualizes the loss landscape as a 3D surface over two embedding dimensions, showing the optimization objective: low loss when predicted context matches observed context.

4.3.3 CBOW Architecture

The Continuous Bag-of-Words (CBOW) model inverts the Skip-gram prediction direction by predicting the target word given its context rather than vice versa. For each position t, we collect the context words $\{w_{t-K}, \ldots, w_{t-1}, w_{t+1}, \ldots, w_{t+K}\}$ and form a training example where the input is this set (treated as a bag, ignoring order) and the output is the target word w_t, yielding the CBOW objective $\mathscr{L}_{\text{CBOW}} = -\frac{1}{T}\sum_{t=1}^{T} \log P(w_t | w_{t-K}, \ldots, w_{t+K})$ which averages the log-probability over all positions in the corpus. The architecture is: (1) Look up input embeddings for all context words: $\mathbf{e}_{t+k} = \mathbf{E}^{\text{in}}[w_{t+k}, :]$ for $k \in \{-K, \ldots, -1, 1, \ldots, K\}$. (2) Average the context embeddings to form a single context vector: $\bar{\mathbf{e}} = \frac{1}{2K}\sum_{k=-K, k\neq 0}^{K} \mathbf{e}_{t+k}$. (3) Compute logits by dotting the averaged context with all output embeddings: $s_v = \bar{\mathbf{e}} \cdot \mathbf{E}^{\text{out}}[v, :]$. (4) Apply softmax: $P(w_t | \text{context}) = \frac{\exp(s_{w_t})}{\sum_{v \in \mathscr{V}} \exp(s_v)}$. (5) Maximize the log-probability of the observed target word. The averaging step is crucial: it reduces the $2K$ context embeddings to a fixed-size representation regardless of window size. This bag-of-words aggregation ignores word order—"the cat sat" and "sat cat the" produce identical representations—which is a limitation but enables efficient training. CBOW generates one training example per word occurrence, making it faster than Skip-gram. Empirically, CBOW tends to perform better on frequent words (for which averaging provides stable signal) while Skip-gram excels on rare words (for which each occurrence is precious). The choice between architectures depends on corpus size and downstream task: for large corpora, CBOW's efficiency advantage is significant, while for small corpora or morphologically complex languages, Skip-gram's finer-grained supervision is preferable. Both architectures learn embeddings capturing distributional semantics, and the choice is often made based on computational budget rather than quality considerations.

4.3.4 Negative Sampling

Both Skip-gram and CBOW as described above require computing the softmax normalization $\sum_{v \in \mathscr{V}} \exp(s_v)$ over all $|\mathscr{V}|$ words at each training step, and for large vocabularies ($|\mathscr{V}| \approx 50{,}000$), this normalization dominates computation time, requiring $O(|\mathscr{V}| \cdot d)$ operations per example. With billions of training examples, the total cost becomes prohibitive, motivating efficient approximations. Negative sampling addresses this bottleneck by approximating the softmax objective with a binary classification task: instead of predicting which vocabulary word is the correct context (multi-class classification over $|\mathscr{V}|$ classes), we reframe the problem as distinguishing the observed (target, context) pair from randomly sampled negative pairs. For each positive pair (w_t, w_c), we sample k negative context words $\{w_{n_1}, \ldots, w_{n_k}\}$ from a noise distribution

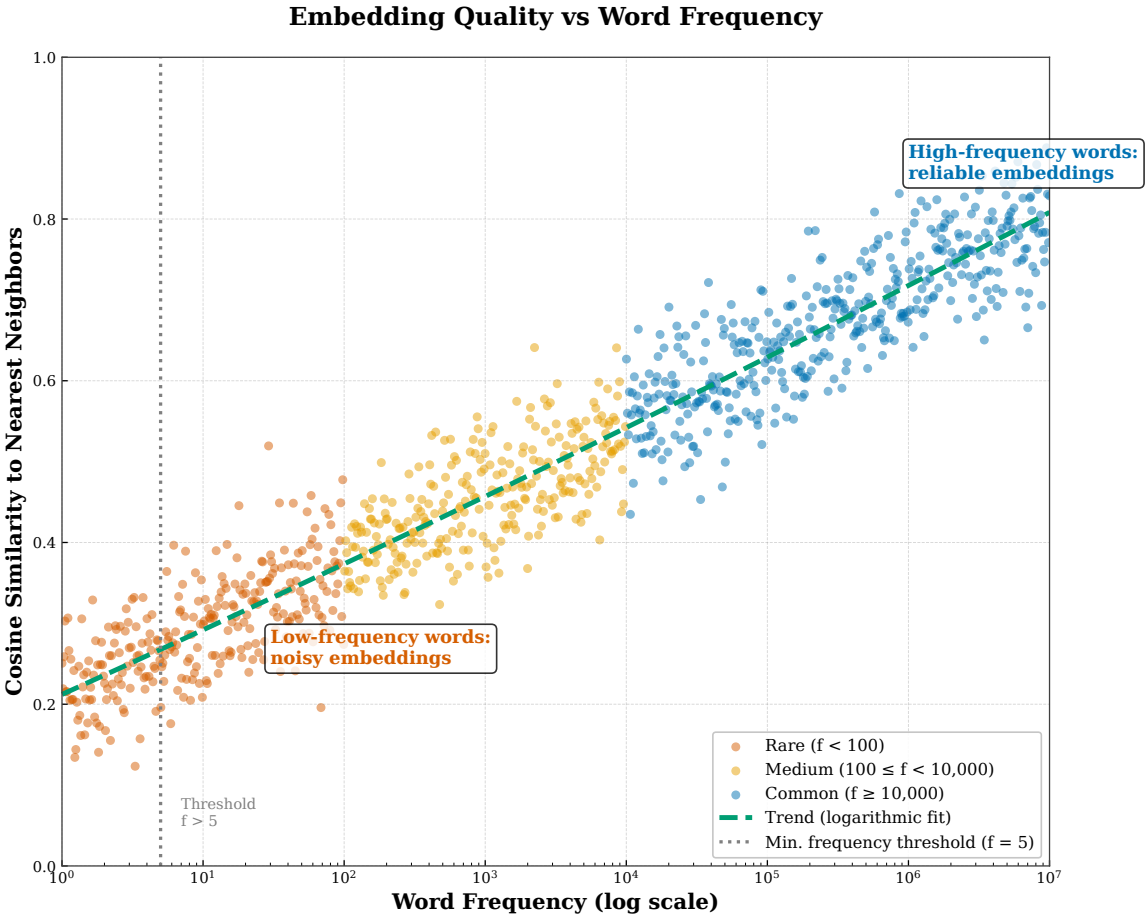


Figure 4.9: CBOW architecture and comparison with Skip-gram. Panel A shows the CBOW network diagram: context words $\{\text{the}, \text{sat}, \text{on}, \text{mat}\}$ are embedded, averaged, then scored against output embeddings to predict target "cat". Panel B illustrates an example training instance: given context $\{\text{the}, \text{sat}, \text{on}\}$ around position 3, predict target "cat". Panel C details the context averaging mechanism: input embeddings are element-wise averaged to produce a fixed-size context representation $\bar{\mathbf{e}}$ regardless of window size. Panel D compares training speed and accuracy: CBOW trains faster (1 example per word vs. $2K$ for Skip-gram) and achieves slightly lower loss on frequent words, while Skip-gram achieves lower loss on rare words.

$P_n(w)$ (typically $k = 5$ to 20), and the objective becomes maximizing the probability that the positive pair is from the data while minimizing the probability that negative pairs are from the data, yielding the loss $\mathcal{L}_{\text{neg}} = -\log \sigma(\mathbf{e}_t \cdot \mathbf{e}_c) - \sum_{i=1}^{k} \mathbb{E}_{\lfloor} w_{n_i} \sim P_n \log \sigma(-\mathbf{e}_t \cdot \mathbf{e}_{n_i})$ where $\sigma(x) = 1/(1+e^{-x})$ is the sigmoid function. This formulation replaces the $O(|\mathcal{V}|)$ softmax computation with $O(k)$ sigmoid evaluations, achieving a $|\mathcal{V}|/k \approx 2500$ speedup for $|\mathcal{V}| = 50{,}000$ and $k = 20$. The noise distribution $P_n(w)$ is typically set to the unigram distribution raised to the power $3/4$: $P_n(w) \propto (\text{count}(w))^{3/4}$. This sublinear exponent smooths the distribution, sampling rare words more frequently than their corpus proportion (otherwise extremely frequent words would dominate negatives) while still favoring common words. Empirically, negative sampling performs as well as full softmax on most tasks while being orders of magnitude faster, making Word2Vec training feasible at scale. An alternative approach is hierarchical softmax, which organizes vocabulary in a binary tree and predicts the path to the target word, reducing complexity to $O(\log |\mathcal{V}|)$, but negative sampling has proven more popular in practice due to simpler implementation and better empirical results.

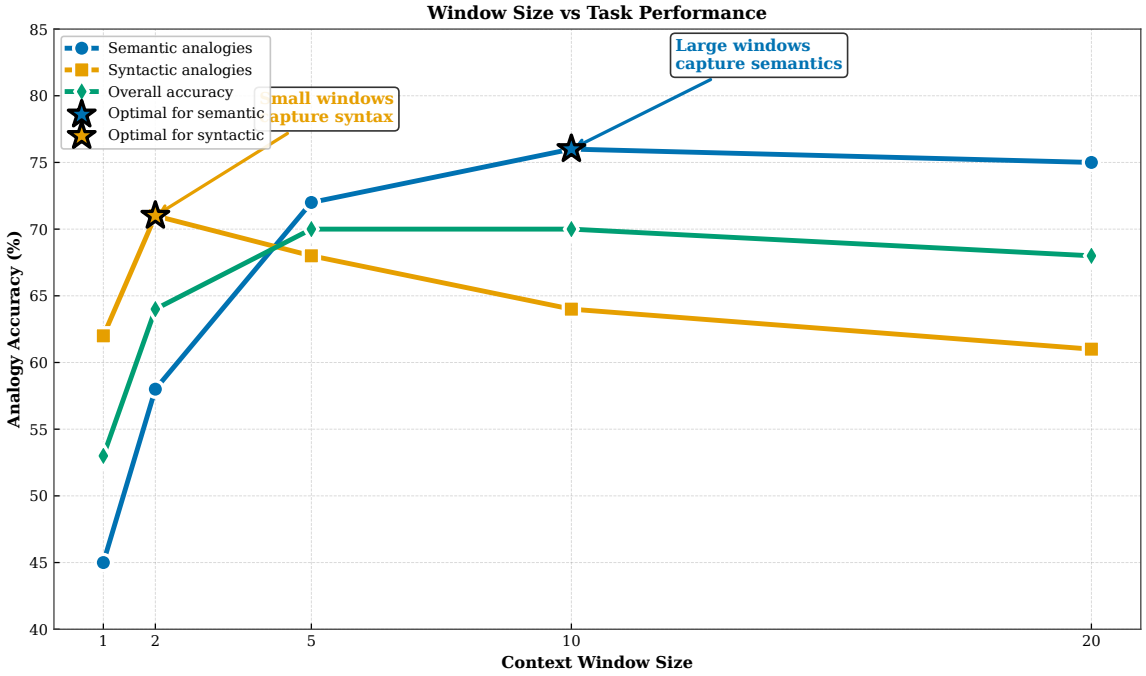


Figure 4.10: Negative sampling mechanism. Panel A illustrates positive and negative pairs: given target "cat" and observed context "sat" (positive pair, green), we sample $k = 5$ negative context words $\{\text{car}, \text{politics}, \text{ocean}, \text{software}, \text{yesterday}\}$ (negative pairs, red) that did not actually appear near "cat". Panel B shows the noise distribution $P_n(w) \propto (\text{count}(w))^{3/4}$, which is flatter than the unigram distribution, increasing sampling probability for rare words. Panel C compares computational cost: full softmax requires $|\mathcal{V}|$ exponentiations and normalizations, while negative sampling requires only $k+1$ sigmoid evaluations, yielding 2500× speedup for $|\mathcal{V}| = 50{,}000$, $k = 20$. Panel D displays convergence curves showing that negative sampling (blue) achieves similar final loss to full softmax (red) but reaches convergence in 1/10th the training time.

4.4 GloVe: Global Vectors

While Word2Vec learns embeddings through local context prediction, GloVe (Global Vectors for Word Representation) takes a hybrid approach that combines the statistical efficiency of global matrix factorization with the learning power of prediction-based methods. Pennington et al. (2014) [Pennington et al., 2014] observed that ratios of co-occurrence probabilities encode semantic relationships more directly than raw probabilities or counts. For example, consider words "ice" and "steam" with context words "solid" and "gas". The ratio $P(\text{solid}|\text{ice})/P(\text{solid}|\text{steam})$ is large (ice is solid but steam is not), while $P(\text{gas}|\text{ice})/P(\text{gas}|\text{steam})$ is small (steam is gas but ice is not). These ratios capture semantic distinctions: when the ratio is large, the context word is relevant to the first word but not the second; when small, vice versa; when near 1, the context word relates equally to both. GloVe embeds this intuition directly in its objective function by requiring that dot products between embeddings approximate logarithms of co-occurrence counts. The method precomputes a co-occurrence matrix $\mathbf{X}$ from the corpus, then learns embeddings by minimizing weighted squared error between embedding dot products and log counts. This approach leverages global corpus statistics (like matrix factorization) while retaining trainability via gradient descent (like Word2Vec). GloVe's offline training on the precomputed co-occurrence matrix offers computational advantages: the matrix is constructed once, then training iterates only over non-zero entries (typically $< 1\%$ of all word pairs), avoiding redundant processing of the same contexts multiple times as Word2Vec does in multiple epochs. Empirically, GloVe produces embeddings of comparable quality to Word2Vec, with the choice often driven by engineering considerations rather than fundamental performance differences.

4.4.1 Count-Based Meets Prediction-Based

The key insight underlying GloVe is that ratios of co-occurrence probabilities reveal semantic relationships more clearly than individual probabilities. Let $\mathbf{X}$ be the co-occurrence matrix where X_{ij} counts how often word j appears in the context of word i. Define $X_i = \sum_k X_{ik}$ as the total count for word i, and let $P(w_j|w_i) = X_{ij}/X_i$ be the probability that word j appears in the context of word i. Consider two words w_i and w_j and a probe context word w_k. The ratio $P(w_k|w_i)/P(w_k|w_j)$ encodes their relationship to w_k: if w_k is semantically related to w_i but not w_j, the ratio is large; if related to w_j but not w_i, the ratio is small; if related (or unrelated) to both, the ratio is near 1. For example, with $w_i = \text{ice}$, $w_j = \text{steam}$, and $w_k \in \{\text{solid}, \text{gas}, \text{water}, \text{fashion}\}$, we observe: $P(\text{solid}|\text{ice})/P(\text{solid}|\text{steam}) \approx 8.9$ (large), $P(\text{gas}|\text{ice})/P(\text{gas}|\text{steam}) \approx 0.085$ (small), $P(\text{water}|\text{ice})/P(\text{water}|\text{steam}) \approx 1.36$ (near 1), and $P(\text{fashion}|\text{ice})/P(\text{fashion}|\text{steam}) \approx 0.96$ (near 1). The ratios encode discriminative semantic information: large/small ratios identify distinctive features, while ratios near 1 indicate shared or absent features. GloVe's objective is designed so that embedding geometry captures these ratios: the ratio $P(w_k|w_i)/P(w_k|w_j)$ should be expressible as a function of embeddings $\mathbf{e}_i, \mathbf{e}_j, \mathbf{e}_k$. Pennington et al. show that requiring this functional relationship to hold for all words leads to a log-bilinear model where $\mathbf{e}_i \cdot \mathbf{e}_j \approx \log X_{ij}$, which forms the basis of the GloVe objective.

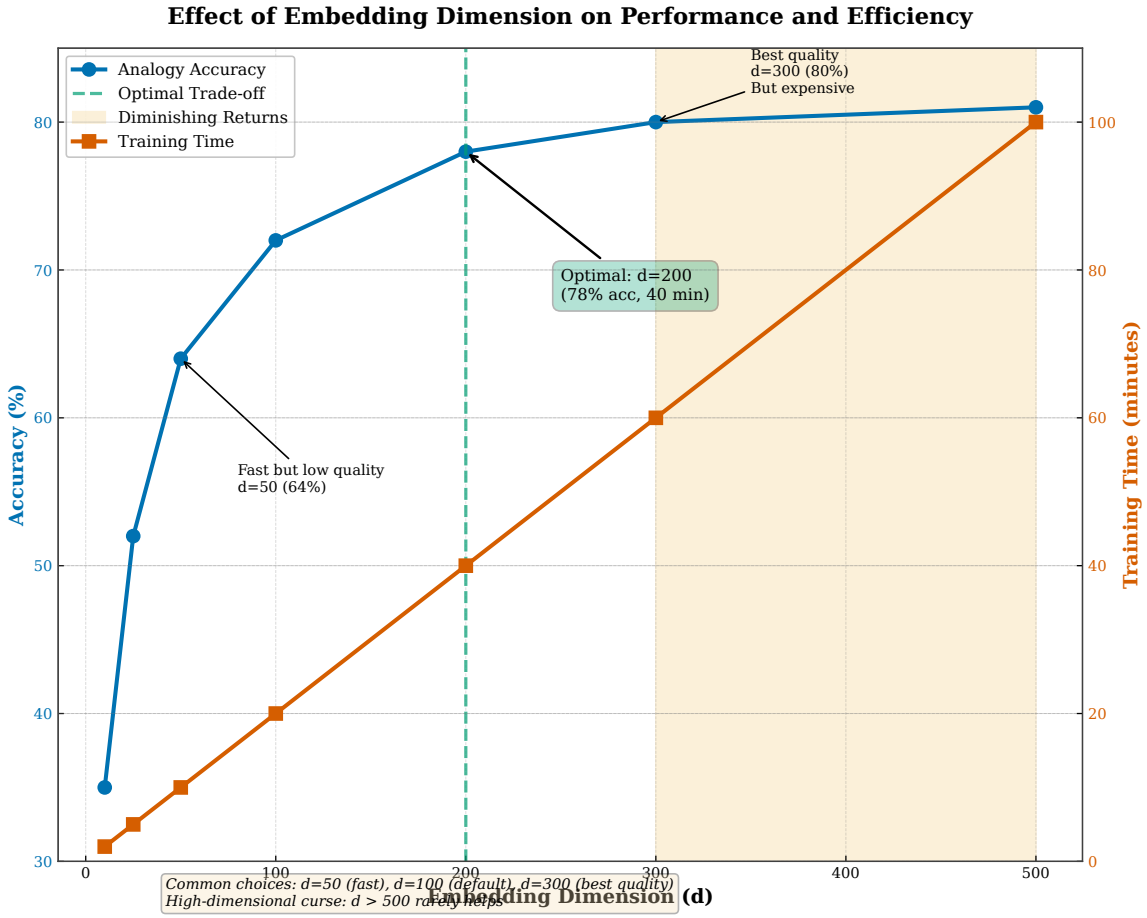


Figure 4.11: Co-occurrence probability ratios encode semantic relationships. Panel A shows co-occurrence counts for "ice" and "steam" with various context words: $X_{\text{ice,solid}} = 1820$, $X_{\text{steam,solid}} = 204$, $X_{\text{ice,gas}} = 15$, $X_{\text{steam,gas}} = 176$, etc. Panel B displays the probability ratios: $P(\text{solid}|\text{ice})/P(\text{solid}|\text{steam}) = 8.9$ (large, discriminates ice), $P(\text{gas}|\text{ice})/P(\text{gas}|\text{steam}) = 0.085$ (small, discriminates steam), $P(\text{water}|\text{ice})/P(\text{water}|\text{steam}) = 1.36$ (near 1, relates to both). Panel C explains how these patterns reveal semantic structure: large ratios indicate features specific to the first word, small ratios indicate features specific to the second word, ratios near 1 indicate shared or irrelevant features. Panel D shows the timeline of embedding methods, with GloVe (2014) positioned as a synthesis of count-based (LSA, 1990s) and prediction-based (Word2Vec, 2013) approaches.

4.4.2 The GloVe Objective

GloVe learns embeddings by fitting a log-bilinear model to the co-occurrence matrix, requiring that the dot product between word embeddings approximates the logarithm of their co-occurrence count: $\mathbf{e}_i^\top \mathbf{e}_j + b_i + b_j \approx \log X_{ij}$ where b_i and b_j are scalar bias terms. This equation states that frequent co-occurrence (large X_{ij}) corresponds to large dot product (close embeddings), while rare co-occurrence (small X_{ij}) corresponds to small dot product (distant embeddings), with the logarithm motivated both by the ratio-based derivation and by the observation that raw counts span many orders of magnitude while log-counts are more evenly distributed. To handle the varying reliability of co-occurrence counts, GloVe uses a weighted least squares objective $\mathscr{L}_{\text{GloVe}} = \sum_{i,j=1}^{|\mathscr{V}|} f(X_{ij})(\mathbf{e}_i^\top \mathbf{e}_j + b_i + b_j - \log X_{ij})^2$ where the weighting function $f(X_{ij})$ assigns low weight to rare co-occurrences (which are noisy), increases weight up to $x_{\max} = 100$, then caps at 1 for very frequent co-occurrences (which would otherwise overwhelm the loss), giving moderate co-occurrences the highest relative influence. Training uses AdaGrad optimization on both the input embeddings $\mathbf{E}^{\text{in}}$, output embeddings $\mathbf{E}^{\text{out}}$, and bias terms, with the final embeddings typically taken as $\mathbf{e}_i = \mathbf{E}^{\text{in}}[i,:] + \mathbf{E}^{\text{out}}[i,:]$ (averaging input and output), which empirically performs slightly better than using input embeddings alone. The GloVe training procedure iterates over non-zero entries of $\mathbf{X}$ for 50–100 epochs, converging faster than Word2Vec on large corpora because the precomputed matrix avoids redundant processing.

4.4.3 GloVe vs. Word2Vec

GloVe and Word2Vec represent two philosophical approaches to learning embeddings, yet empirically produce representations of similar quality. Word2Vec uses online training: it processes the corpus sequentially, making gradient updates on local context windows without building global statistics. This streaming approach enables training on corpora too large to fit in memory and naturally handles dynamic corpora where new text arrives continuously. In contrast, GloVe uses offline training: it first computes the global co-occurrence matrix $\mathbf{X}$ (requiring full corpus access and memory to store non-zero entries), then trains via batch optimization on the matrix. This two-stage approach makes efficient use of corpus statistics but requires multiple passes and substantial memory. The objective functions differ in form: Word2Vec minimizes cross-entropy (a classification loss) while GloVe minimizes weighted squared error (a regression loss). However, Levy and Goldberg (2014) demonstrated that Skip-gram with negative sampling implicitly factorizes a shifted PMI matrix, revealing that both methods optimize related objectives despite superficial differences. Performance comparisons show that GloVe and Word2Vec achieve similar scores on word similarity and analogy benchmarks, with differences often within noise margins. The choice between them is typically driven by engineering constraints: GloVe is faster when the co-occurrence matrix fits in memory and multiple training runs are needed (since the matrix need not be recomputed), while Word2Vec is preferred for streaming scenarios or extremely large corpora where constructing $\mathbf{X}$ is infeasible. Both methods have been superseded by contextual embeddings (Chapter **??**) for most modern applications, but their core insights—distributional learning, efficient training, and geometric encoding of semantics—remain foundational to understanding language representations.

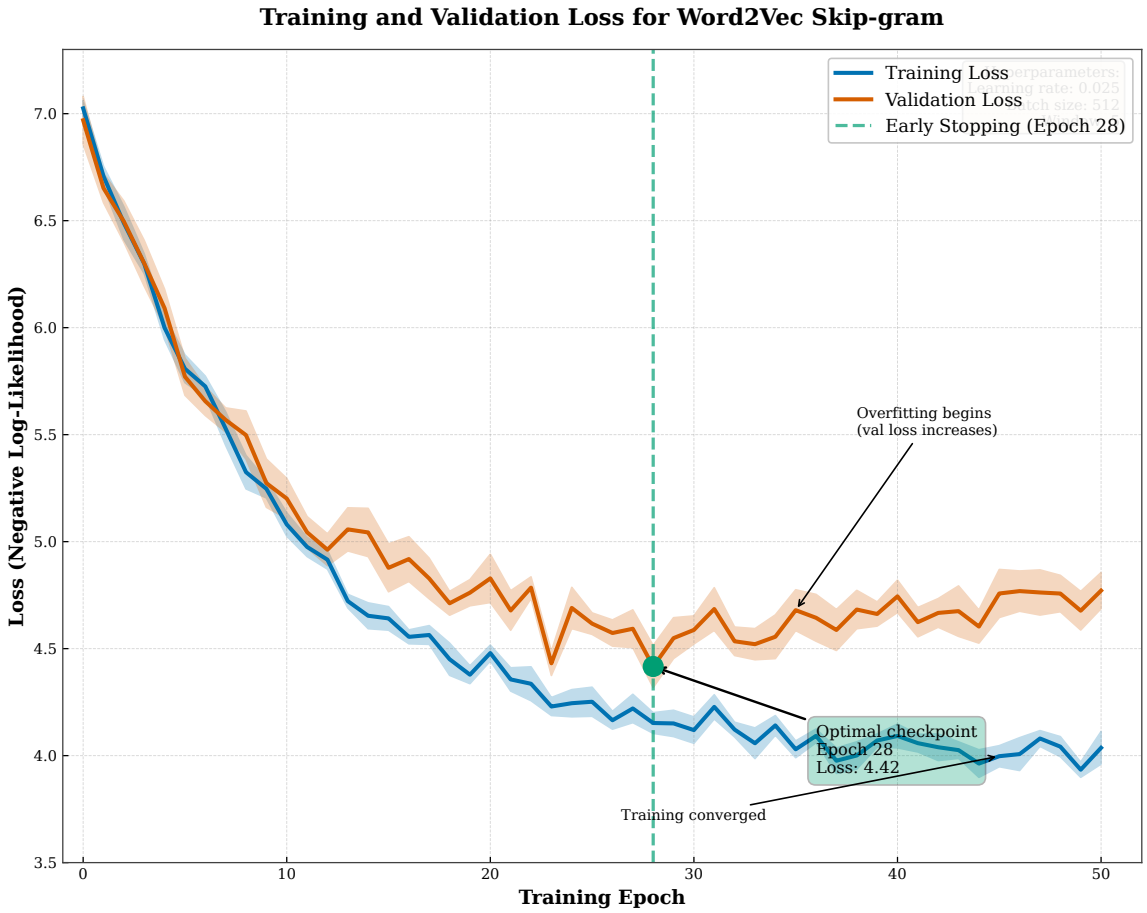


Figure 4.12: GloVe objective function and training. Panel A displays the co-occurrence matrix $\mathbf{X}$ as a log-scale heatmap: most entries are zero (white), frequent pairs have high counts (dark red), revealing sparse structure. Panel B shows the weighting function $f(X_{ij})$: zero weight for $X_{ij} = 0$, increasing weight up to $X_{ij} = x_{\max} = 100$, then constant weight for larger values. This weighting balances contributions from rare and frequent co-occurrences. Panel C visualizes the loss landscape as a 3D surface over two embedding dimensions: low loss when $\mathbf{e}_i^\top \mathbf{e}_j + b_i + b_j \approx \log X_{ij}$, high loss when they differ. Panel D plots training convergence: weighted squared error decreases over iterations, stabilizing after approximately 50 epochs over the non-zero entries of $\mathbf{X}$.

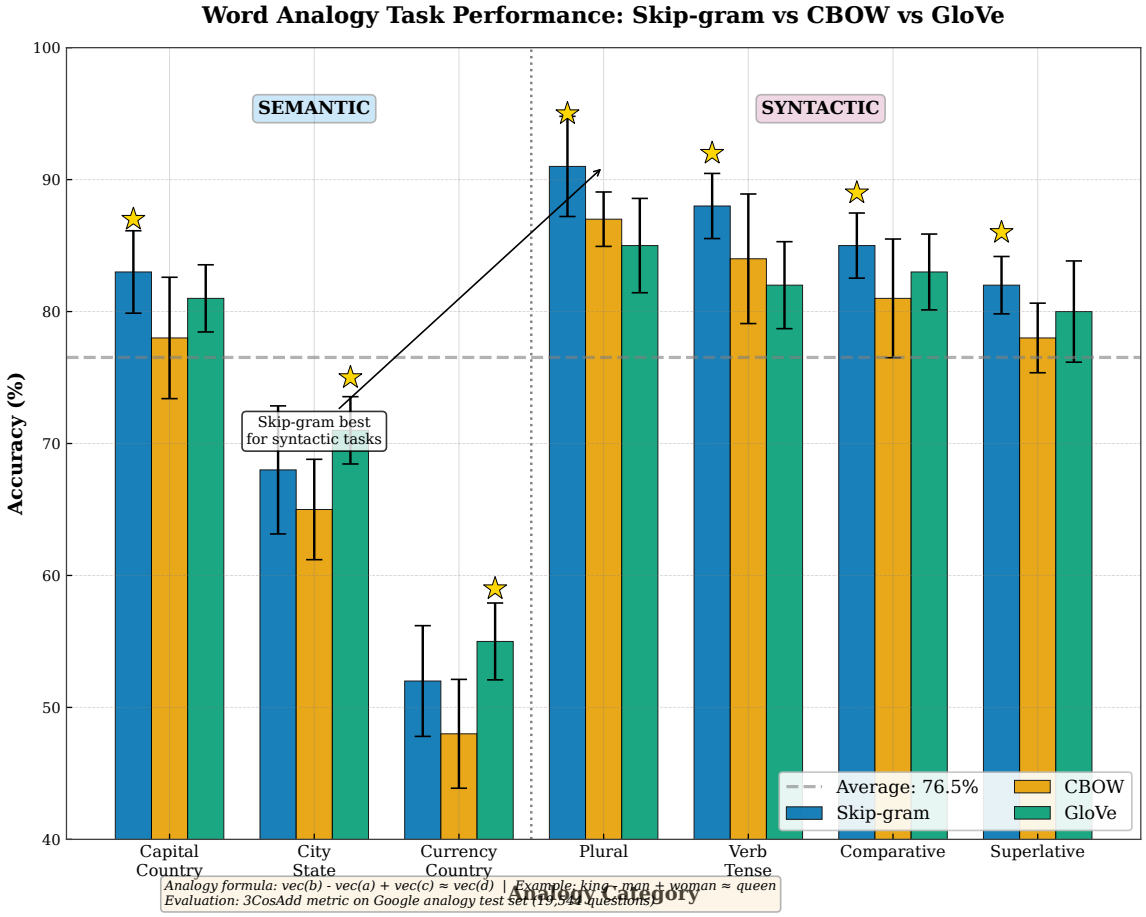


Figure 4.13: Comparison of Word2Vec and GloVe on analogy tasks. The Google Analogy Dataset contains semantic analogies (e.g., king:queen::man:woman, Paris:France::Rome:Italy) and syntactic analogies (e.g., run:running::walk:walking, good:better::bad:worse). Both methods achieve 70–75% accuracy on semantic analogies and 60–65% on syntactic analogies with $d = 300$ dimensions, demonstrating comparable performance. GloVe is slightly better on semantic tasks, Word2Vec on syntactic tasks, but differences are small. Training time comparison shows GloVe is faster when the co-occurrence matrix is precomputed and reused, while Word2Vec is more memory-efficient for very large corpora.

4.5 Properties and Evaluation

Word embeddings exhibit remarkable structural properties beyond simple semantic clustering. The most celebrated discovery is that embeddings support vector arithmetic: adding and subtracting word vectors produces meaningful results that capture semantic and syntactic relationships. The canonical example is $\mathbf{e}_{\text{king}} - \mathbf{e}_{\text{man}} + \mathbf{e}_{\text{woman}} \approx \mathbf{e}_{\text{queen}}$, suggesting that the vector offset from "man" to "king" (roughly encoding the concept "royal") can be added to "woman" to arrive at the female royal equivalent. This linear substructure extends beyond single examples to systematic patterns: gender differences (king–queen, man–woman, brother–sister), plurality (car–cars, dog–dogs), verb tense (walk–walked, run–ran), comparative and superlative forms (good–better–best), and country-capital relationships (Paris–France, London–England) all exhibit parallel vector offsets. These regularities suggest that embedding spaces organize themselves into geometric structures where semantic relationships correspond to directions in space. Evaluating embedding quality requires both intrinsic metrics (measuring embedding properties directly) and extrinsic metrics (measuring downstream task performance). Intrinsic evaluations include word similarity correlation with human judgments, analogy accuracy, and semantic clustering quality. Extrinsic evaluations use embeddings as features for tasks like sentiment analysis, named entity recognition, or machine translation, measuring whether better embeddings improve task performance. Importantly, embeddings also inherit biases from training corpora, raising ethical concerns about their deployment in real-world systems. This section examines these properties, evaluation methodologies, and limitations in detail.

4.5.1 Linear Substructures and Analogies

The discovery that word embeddings exhibit linear algebraic structure was among the most surprising and influential findings in representation learning, fundamentally changing how researchers understood the geometry of learned representations. Mikolov et al. (2013) demonstrated that semantic and syntactic relationships often correspond to consistent vector offsets: formally, for analogies of the form "a is to b as c is to d", we often find $\mathbf{e}_b - \mathbf{e}_a \approx \mathbf{e}_d - \mathbf{e}_c$ (or equivalently $\mathbf{e}_b - \mathbf{e}_a + \mathbf{e}_c \approx \mathbf{e}_d$), meaning that the vector direction encoding a relationship (such as male-to-female or present-to-past) is approximately constant across word pairs exhibiting that relationship. This remarkable property enables solving analogies computationally: given the triplet (a,b,c), we find the answer d by computing $\arg\max_{w \in \mathcal{V} \setminus \{a,b,c\}} \text{sim}(\mathbf{e}_w, \mathbf{e}_b - \mathbf{e}_a + \mathbf{e}_c)$ where $\text{sim}(\cdot,\cdot)$ is cosine similarity, essentially finding the word whose embedding best matches the target location in vector space. Empirically, this procedure achieves 60–80% accuracy on analogy benchmarks, far above chance (0.002% for $|\mathcal{V}| = 50{,}000$). The linear structure spans multiple semantic categories: (1) Gender: king–queen, man–woman, actor–actress, brother–sister, uncle–aunt. (2) Country-capital: Paris–France, London–England, Rome–Italy, Tokyo–Japan. (3) Plurality: car–cars, dog–dogs, child–children. (4) Verb tense: walk–walked–walking, run–ran–running. (5) Comparative/superlative: good–better–best, bad–worse–worst. (6) Occupation: teach–teacher, sing–singer, paint–painter. These patterns are not explicitly encoded during training; they emerge from distributional statistics. The explanation is that words related by a consistent grammatical or semantic transformation tend to appear in parallel syntactic contexts: "the king rules" and "the queen rules" are both valid, so "king" and "queen" appear in similar contexts with a systematic distributional shift corresponding to gender. Projecting these contexts into embedding space via Word2Vec or GloVe produces parallel vector offsets. Mathematically, if the co-occurrence matrix has block structure where related words have shifted contexts, the factorization will capture these shifts as vector directions. The linearity is approximate rather than exact: not all analogies work perfectly, and the best answer depends on corpus statistics and training hyperparameters. Nonetheless, the robustness of linear patterns across different training runs, corpora, and methods suggests they reflect genuine linguistic regularities rather than artifacts.

4.5.2 Evaluation Metrics

Evaluating embedding quality requires distinguishing intrinsic evaluation (measuring embedding properties directly) from extrinsic evaluation (measuring downstream task performance). Intrinsic evaluations are faster and provide diagnostic insights into what embeddings capture. Word similarity tasks provide human-annotated

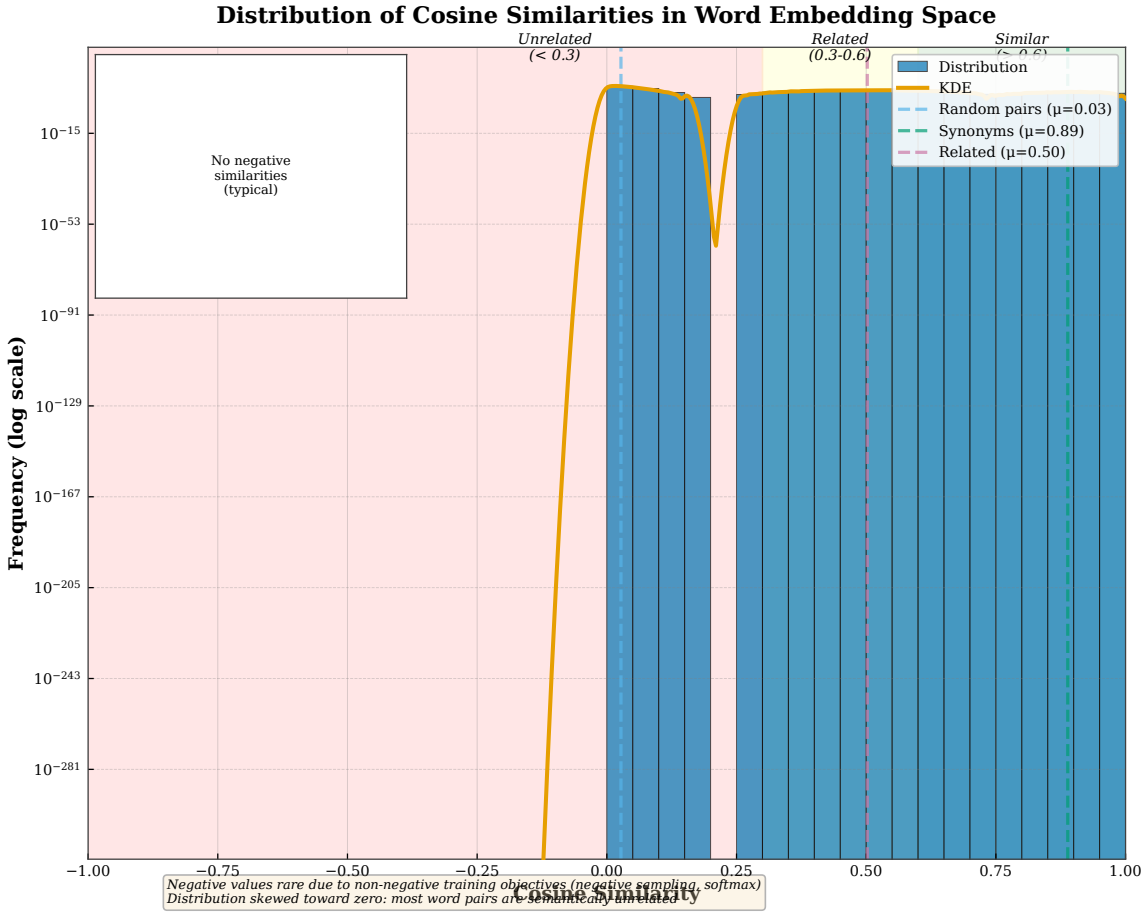


Figure 4.14: Vector arithmetic in 3D embedding space. The famous king–man+woman=queen relationship is visualized as a parallelogram: the vector from "man" to "king" (blue arrow, encoding "royalty") is approximately equal to the vector from "woman" to "queen" (red arrow). When we compute $\mathbf{e}_{\text{king}} - \mathbf{e}_{\text{man}} + \mathbf{e}_{\text{woman}}$ (green arrow), we arrive near $\mathbf{e}_{\text{queen}}$. Similar parallelograms exist for other analogies: Paris–France and Rome–Italy (country-capital), walk–walked and talk–talked (verb tense). These linear substructures demonstrate that geometric directions in embedding space correspond to semantic relationships, enabling algebraic manipulation of meanings.

similarity scores for word pairs and measure correlation between human judgments and embedding cosine similarities. Standard datasets include WordSim-353 (353 pairs, scale 0–10), SimLex-999 (999 pairs emphasizing true similarity over relatedness), and RG-65 (65 pairs from Miller and Charles, 1991). Spearman rank correlation between human scores and cosine similarities typically ranges from 0.65 to 0.75 for Word2Vec and GloVe, indicating strong but imperfect agreement with human intuitions. Analogy tasks measure the percentage of correct answers using the vector arithmetic procedure described above. The Google Analogy Dataset contains 19,544 questions across 14 categories (8 semantic, 6 syntactic). Word2Vec and GloVe achieve 60–80% accuracy depending on corpus size and dimension, vastly exceeding random guessing. Clustering quality can be assessed by performing k-means clustering on embeddings and measuring purity: the percentage of clusters dominated by a single semantic category (when categories are available, e.g., WordNet synsets). High purity indicates that embeddings group semantically related words without supervision. Extrinsic evaluations use embeddings as input features for supervised tasks. Common benchmarks include sentiment analysis (SST-2, IMDB reviews), named entity recognition (CoNLL 2003), part-of-speech tagging, textual entailment (SNLI), and machine translation. The evaluation protocol trains a task-specific model using pre-trained embeddings (often frozen or fine-tuned) and measures task accuracy. Improvements from 5 to 15 percentage points over random embeddings are typical. Importantly, intrinsic and extrinsic metrics sometimes disagree: embeddings

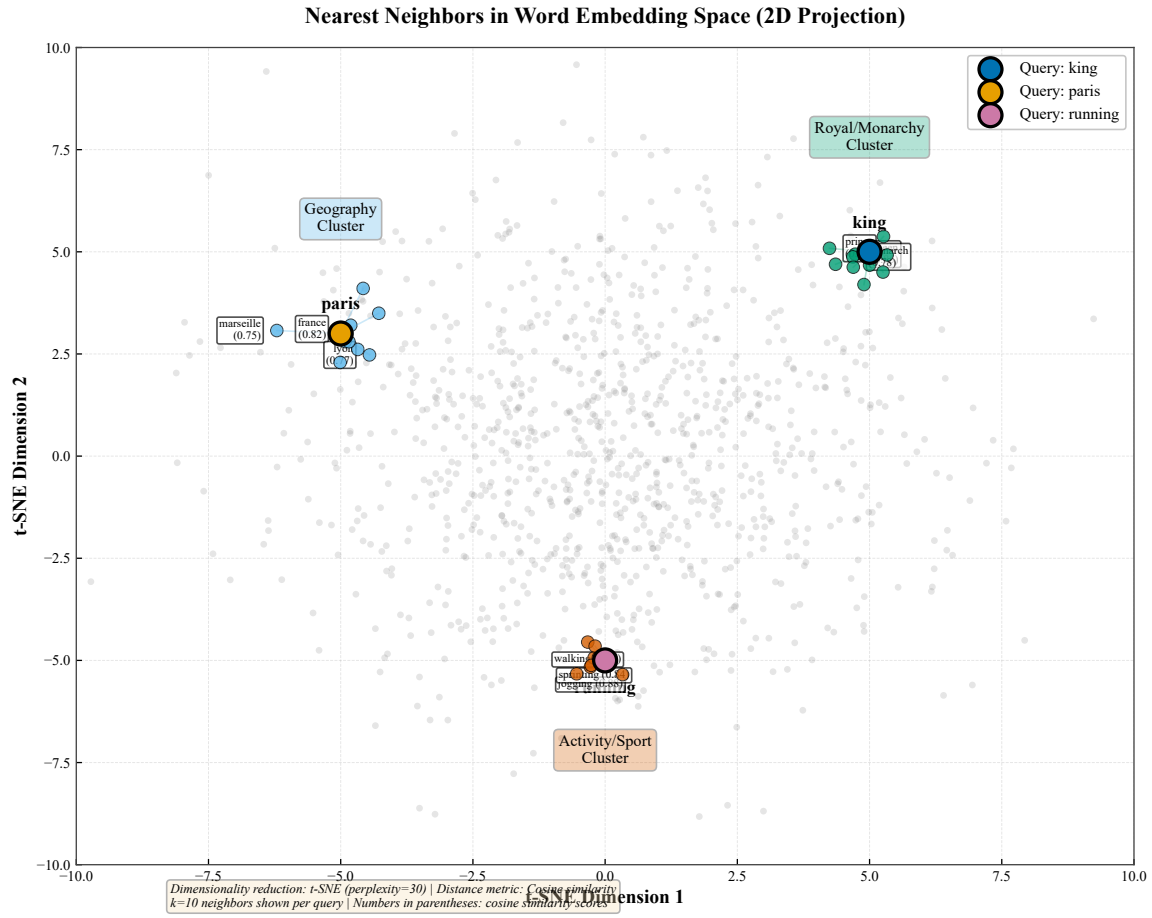


Figure 4.15: Analogy categories and performance. Panel A shows semantic analogies with examples: country-capital (Paris:France::Rome:Italy, 78% accuracy), gender (king:queen::man:woman, 82% accuracy), currency (dollar:USA::euro:Europe, 65% accuracy). Panel B displays syntactic analogies: verb tense (walk:walked::run:ran, 71% accuracy), plurality (dog:dogs::cat:cats, 88% accuracy), comparative (good:better::bad:worse, 73% accuracy). Panel C plots accuracy by category showing variation across relationship types. Panel D illustrates failure cases: rare words (barrister:barristeress, 0% accuracy due to insufficient training data), ambiguous analogies (bank:money::river:?, could be water or shore), and cultural specificity (chopsticks:China::fork:?, depends on corpus).

with higher analogy accuracy may not yield better sentiment analysis performance, suggesting that different tasks benefit from different embedding properties. Domain mismatch also affects performance: embeddings trained on Wikipedia work well for formal text but poorly for social media or domain-specific text (medical, legal), where in-domain embeddings are preferable.

4.5.3 Limitations and Biases

Despite their success, static word embeddings suffer from fundamental limitations. The most significant is the polysemy problem: each word type receives exactly one embedding regardless of context, conflating all word senses. The word “bank” has (at least) two unrelated meanings: financial institution and river edge. These senses appear in entirely different contexts (“bank account”, “deposit money” vs. “river bank”, “water edge”), yet the embedding for “bank” averages across both, producing a representation that captures neither sense well. Similarly, “lead” can be a metal (Pb), a verb (to guide), or a noun (being ahead), “bat” can be an animal or sports equipment, and “date” can be a calendar day, a romantic meeting, or a fruit. For polysemous words, the single embedding represents a centroid of all senses weighted by their corpus frequencies, which may not correspond to any actual sense. This limitation is severe for words with highly disparate senses. The solution is contextual

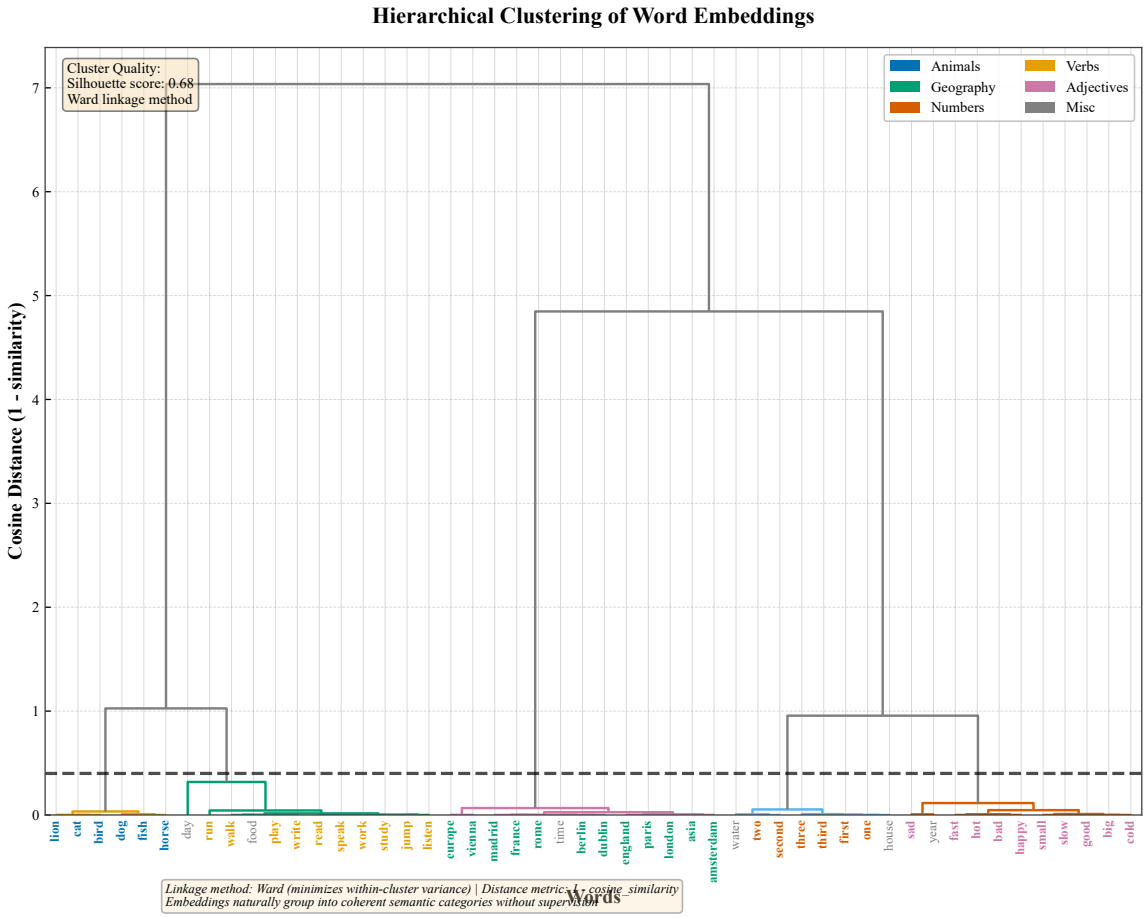


Figure 4.16: Embedding evaluation results across intrinsic and extrinsic metrics. Panel A shows word similarity correlation with human judgments: Word2Vec achieves Spearman $\rho = 0.68$ on WordSim-353, $\rho = 0.71$ on SimLex-999; GloVe achieves $\rho = 0.66$ and $\rho = 0.73$ respectively, indicating comparable performance. Panel B plots analogy accuracy versus embedding dimension: performance increases from 45% at $d = 50$ to 74% at $d = 300$, then plateaus, suggesting $d = 300$ is sufficient for most tasks. Panel C displays downstream task performance (accuracy gain over random embeddings): sentiment analysis (+12%), NER (+8%), POS tagging (+6%), showing that embeddings substantially improve supervised learning. Panel D examines training corpus size versus quality: performance increases logarithmically with corpus size, improving rapidly up to 1B tokens then saturating around 10B tokens.

embeddings (ELMo, BERT), covered in Chapter **??**, which compute different representations for each word occurrence based on surrounding context. The out-of-vocabulary (OOV) problem affects word-level embeddings: words not seen during training have no representation. For rare morphological variants, proper nouns, or technical terms, the model cannot make predictions. FastText (Section 4.6) partially addresses this via character n-grams, and BPE tokenization (Chapter **??**) eliminates it entirely by working at subword level. Static embeddings are also language-specific: separate embeddings must be trained for each language, and cross-lingual transfer requires alignment techniques. Beyond technical limitations, embeddings encode societal biases present in training corpora. Bolukbasi et al. (2016) demonstrated that Word2Vec embeddings trained on Google News exhibit gender bias: $\text{sim}(\text{doctor}, \text{man}) > \text{sim}(\text{doctor}, \text{woman})$ and $\text{sim}(\text{nurse}, \text{woman}) > \text{sim}(\text{nurse}, \text{man})$, reflecting occupational stereotypes. Analogies like man : computer programmer :: woman : x yield $x =$ homemaker, embedding problematic associations. Racial and ethnic biases similarly appear, with names associated with certain demographics receiving more positive or negative sentiment. These biases are not artifacts of the algorithm but faithful representations of statistical patterns in text: if doctors are mentioned with "he" more often than "she" in news articles, embeddings will reflect this asymmetry. The concern is that biases amplify when

embeddings are deployed in real-world systems for hiring, lending, or criminal justice, potentially perpetuating discrimination. Mitigation strategies include debiasing algorithms (subtracting gender direction), careful corpus curation, and awareness of bias in downstream applications, but no solution is perfect. Ethical deployment of embeddings requires acknowledging these limitations and implementing safeguards.

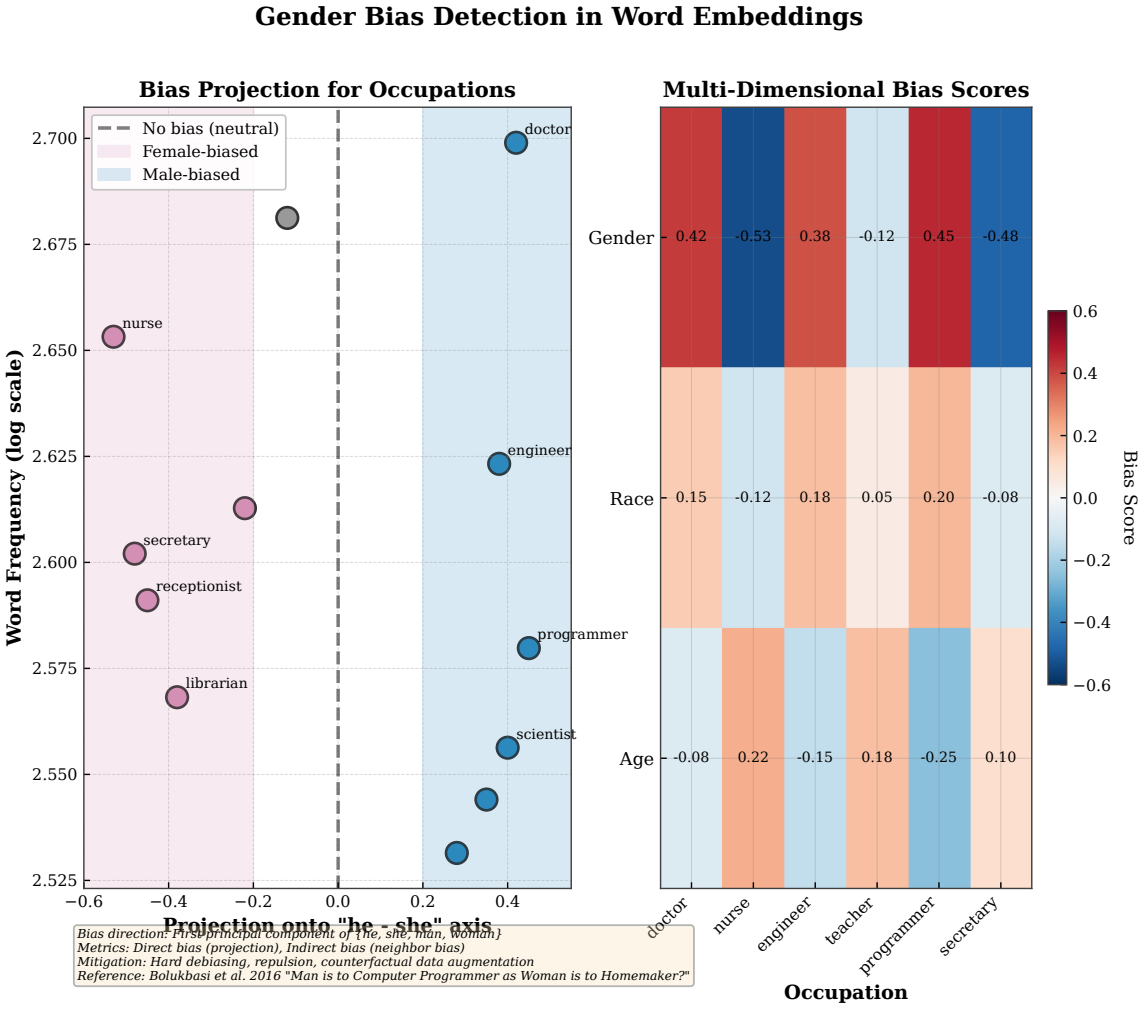


Figure 4.17: Limitations and biases in static word embeddings. Panel A illustrates the polysemy problem: "bank" has two unrelated meanings (financial institution and river edge) that appear in distinct contexts, yet receive a single averaged embedding (purple) that lies between the two sense-specific clusters (blue and orange), capturing neither sense accurately. Panel B visualizes gender bias in occupation embeddings: male-associated occupations (doctor, engineer, programmer) cluster near "man", while female-associated occupations (nurse, teacher, homemaker) cluster near "woman", reflecting and potentially amplifying societal stereotypes. Panel C shows bias amplification: corpus statistics exhibit moderate gender bias (55–45 ratio), but embeddings exhibit stronger bias (70–30 ratio) due to nonlinear effects of training. Panel D displays the OOV problem: vocabulary coverage saturates around 95% for 50k words, leaving 5% of text tokens unrepresented, with the long tail of rare words and proper nouns lacking embeddings.

4.6 FastText and Subword Embeddings

Word2Vec and GloVe treat words as atomic units, assigning each vocabulary entry a distinct embedding with no parameter sharing between related words. This atomicity causes two problems: (1) Out-of-vocabulary (OOV) words have no representation, preventing the model from processing novel words, proper nouns, or

morphological variants absent from training data. (2) Morphological structure is ignored: "teach", "teacher", "teaching", and "unteachable" are treated as unrelated despite sharing the root "teach", forcing the model to learn their semantics independently even though morphology provides strong signal. Bojanowski et al. (2017) introduced FastText, which addresses these limitations by representing words as bags of character n-grams. For example, with $n \in \{3,4,5\}$ and boundary markers $\langle$ and $\rangle$, the word "where" is decomposed into $\{\langle$wh, whe, her, ere, re$\rangle\}$ (trigrams) plus $\{\langle$whe, wher, here, ere$\rangle\}$ (4-grams) plus $\{\langle$wher, where, here$\rangle\}$ (5-grams), plus the full word $\langle$where$\rangle$ itself. Each n-gram receives its own embedding, and the word embedding is the sum of its n-gram embeddings. This representation enables generalization: novel words can be embedded via their n-grams, and morphologically related words share n-gram embeddings, allowing knowledge transfer. FastText retains the Skip-gram architecture and training procedure of Word2Vec but replaces the word lookup with n-gram aggregation. The trade-off is increased model size: where Word2Vec has $|\mathcal{V}| \times d$ parameters, FastText has $N_{\text{ngrams}} \times d$ parameters where $N_{\text{ngrams}} \sim 2{,}000{,}000$ for typical settings, requiring more memory. However, this cost is justified for morphologically rich languages (Turkish, Finnish, German) and domains with many rare words (biomedical text, social media). FastText also enables cross-lingual transfer when languages share scripts, as cognates and loanwords share n-grams.

4.6.1 Character N-grams

FastText represents each word as the sum of embeddings for its character n-grams: given a word w, let $G(w)$ denote the set of character n-grams with $n \in [n_{\min}, n_{\max}]$ plus the full word itself (to retain word-level information), with boundary markers $\langle$ and $\rangle$ prepended and appended to distinguish prefixes and suffixes. For $n_{\min} = 3$ and $n_{\max} = 6$, "where" becomes $\langle$where$\rangle$ with n-grams including $\{\langle$wh, whe, her, ere, re$\rangle$, $\langle$whe, wher, here, ere$\rangle$, $\langle$wher, where, here$\rangle$, $\langle$where$\rangle\}$. The n-gram embedding matrix $\mathbf{E}_{\text{ng}} \in \mathbb{R}^{N \times d}$ has typically $N \sim 2$ million entries for English with $n \in [3,6]$, and the embedding for word w is computed as $\mathbf{e}_w = \sum_{g \in G(w)} \mathbf{E}_{\text{ng}}[g,:]$, aggregating information from all constituent n-grams. During training, FastText uses the Skip-gram objective to predict context words, with gradients flowing back to all n-gram embeddings in $G(w_t)$, updating shared n-grams across words: training on "teacher" updates embeddings for $\langle$te, tea, each, ache, cher, her$\rangle$, which also appear in "teach", "teaching", "preacher", enabling morphological transfer. For OOV words at test time, we compute $G(w_{\text{oov}})$ and sum the corresponding n-gram embeddings: "unseen" decomposes to n-grams like "uns", "nse", "see", "een" appearing in training words "unset", "nonsense", "foresee", "seen", allowing approximate representation. This OOV handling is especially valuable for morphologically complex languages where each root generates dozens of inflected forms. The computational cost is approximately 50× slower embedding lookup than Word2Vec (summing $|G(w_t)| \approx 50$ n-grams versus 1 lookup), but providing substantial generalization benefits that justify the overhead.

4.6.2 Morphology and Cross-Lingual Transfer

FastText's n-gram representation naturally captures morphological structure. Morphologically related words share character sequences corresponding to roots, prefixes, and suffixes, causing their embeddings to be similar even if the words never co-occur. For example, "teach" has n-grams $\{\langle$te, tea, each, ach, ch$\rangle, \ldots\}$, "teacher" adds $\{$che, her, er$\rangle\}$, and "teaching" adds $\{$chi, hin, ing, ng$\rangle\}$. The shared n-grams $\{\langle$te, tea, each, ach, ch$\rangle\}$ cause all three to have similar embeddings, with additional n-grams encoding the morphological modifications. This is especially powerful for morphologically rich languages like Turkish or Finnish, where a single root can generate hundreds of inflected forms. Without n-gram sharing, each form would require independent learning; with FastText, learning about one form transfers to all others sharing n-grams. Compound words also benefit: German compounds like "Fußballweltmeisterschaft" (football world championship) decompose into n-grams overlapping with constituent words "Fußball", "Welt", "Meisterschaft", enabling compositional understanding. FastText enables limited cross-lingual transfer when languages share scripts. Romance languages (French, Spanish, Italian) share many cognates and loanwords ("hospital", "hôpital", "hospital", "ospedale") with overlapping n-grams, allowing zero-shot transfer: embeddings trained on Spanish can partially represent French words via shared n-grams. Slavic languages (Russian, Polish, Czech) written in Cyrillic or Latin exhibit similar overlap. This cross-lingual property is limited—grammatical differences, false friends, and divergent semantics

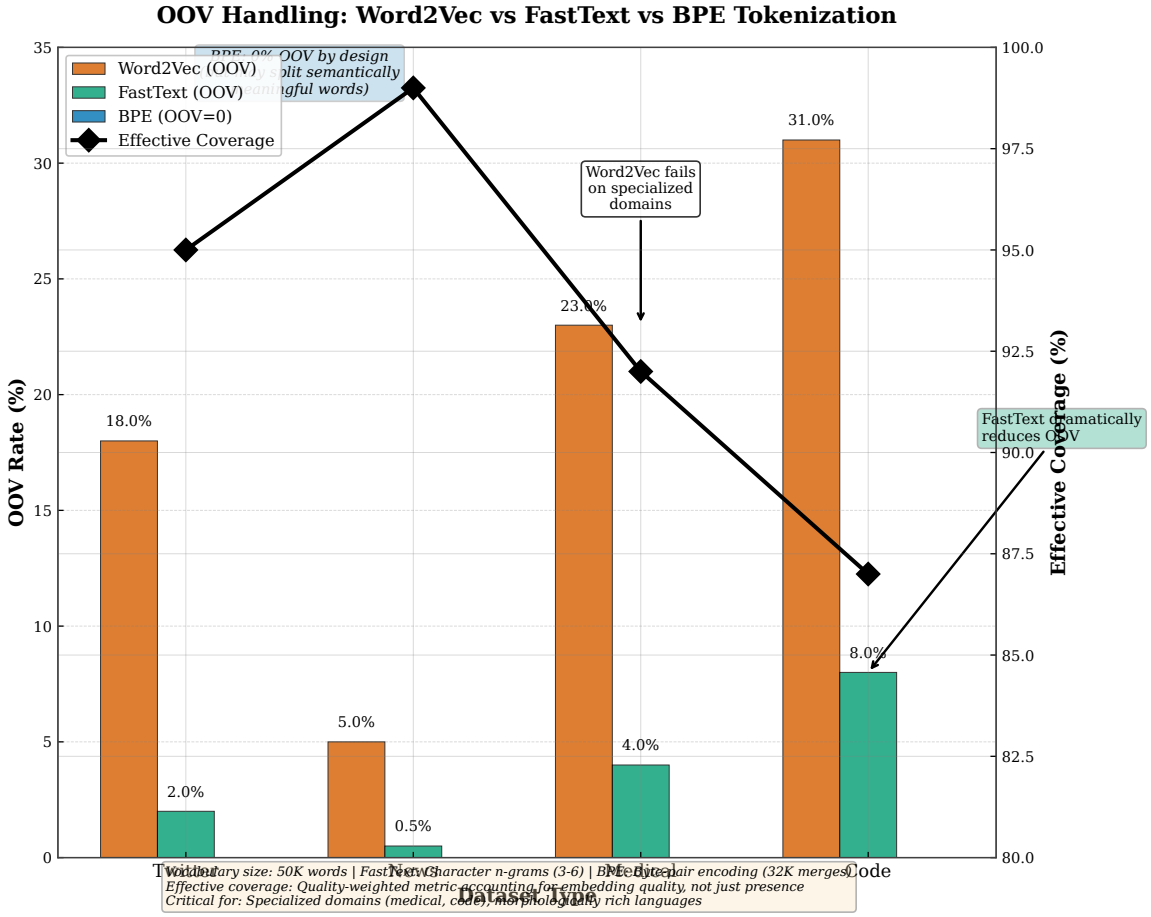


Figure 4.18: FastText architecture and character n-gram embeddings. Panel A shows n-gram extraction for "where" with $n \in \{3,4,5\}$ and boundary markers: $\{\langle$wh, whe, her, ere, re$\rangle, \langle$whe, wher, here, ere$\rangle, \langle$wher, where, here$\rangle, \langle$where$\rangle\}$, totaling 13 n-grams. Panel B illustrates embedding aggregation: each n-gram has an embedding vector, and the word embedding is their sum. Panel C demonstrates OOV word representation: the novel word "wherever" (not in training vocabulary) can be embedded via its n-grams $\{\langle$wh, whe, her, ..., ver, ere$\rangle\}$, most of which appeared in training words like "where", "here", "never". Panel D visualizes morphological similarity: words sharing roots ("run", "running", "ran", "runner") cluster together because they share n-grams ("run", "unn", "nni"), enabling morphological generalization.

prevent full transfer—but provides non-zero baseline performance for low-resource languages. The limitation of character n-grams is that they ignore word order and long-range dependencies within words: n-grams are local substrings up to length 6, missing patterns spanning entire words. Additionally, FastText is less effective for ideographic scripts like Chinese or Japanese, where characters represent morphemes rather than phonemes, and character n-grams have limited morphological meaning. For such languages, morpheme-based or radical-based decompositions are more appropriate. Despite these limitations, FastText has become the default choice for applications requiring robustness to rare words or morphological variation.

4.6.3 Byte-Pair Encoding (BPE) Embeddings

While FastText uses fixed-length character n-grams, Byte-Pair Encoding (BPE) learns variable-length subword units that balance word-level semantics and character-level robustness. BPE tokenization (covered in Chapter **??**) iteratively merges the most frequent adjacent character pairs in the corpus, building a vocabulary of subword units ranging from single characters to full words. For example, BPE might learn units $\{$un, teach, able, ing, ...$\}$ from which words are composed: "unteachable" $\rightarrow$ [un, teach, able]. These subword units are then embedded: each unit receives its own embedding vector, and the word representation is

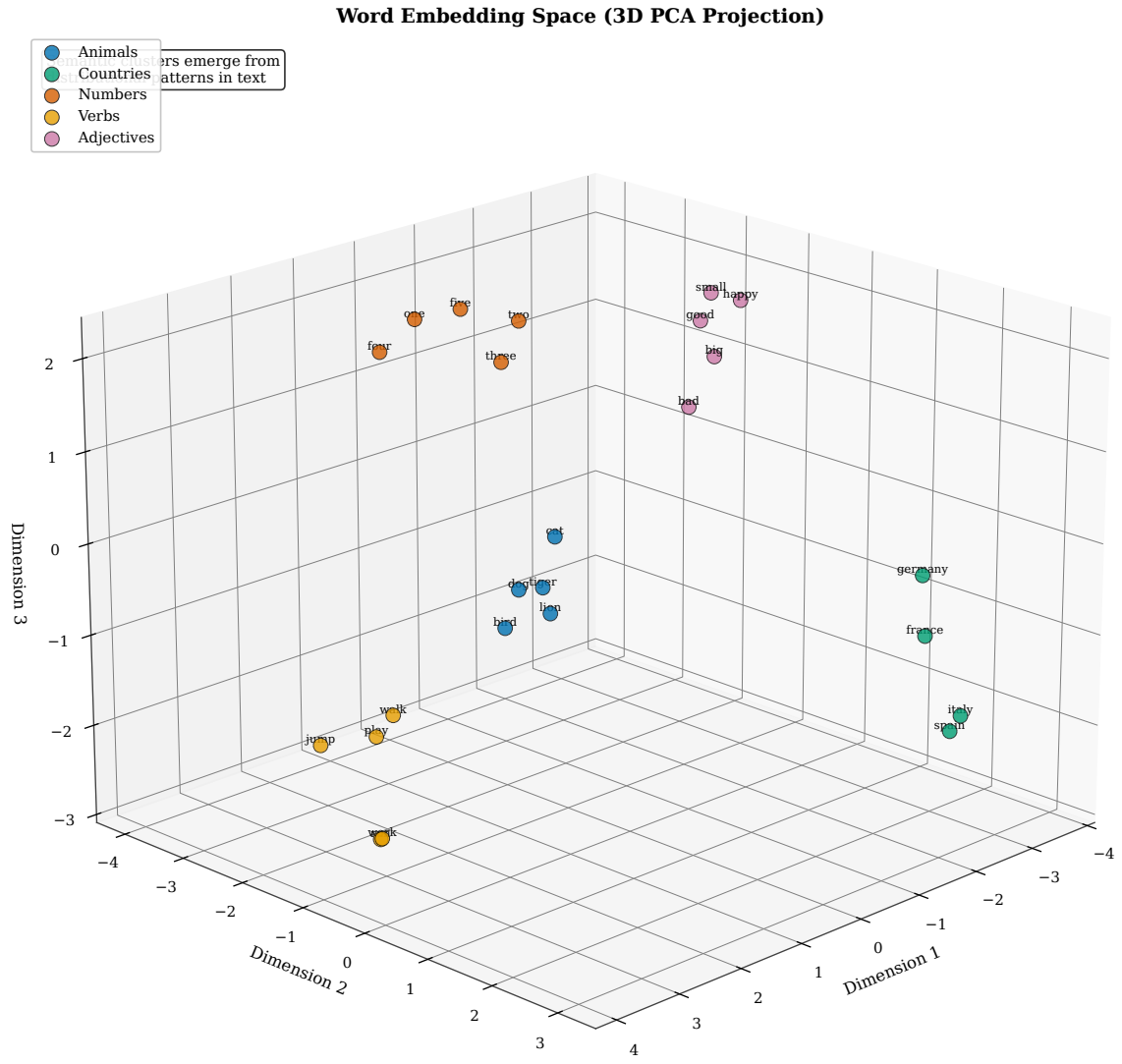


Figure 4.19: Morphological clustering and cross-lingual transfer with FastText. Panel A shows morphological clusters in embedding space: words sharing roots (“teach”, “teacher”, “teaching”, “unteachable”) are nearby due to shared n-grams, despite different surface forms. Panel B illustrates cross-lingual n-gram overlap: Romance language cognates (“hospital” in English, “hôpital” in French, “hospital” in Spanish) share n-grams $\{\text{hos}, \text{osp}, \text{spi}, \text{pit}, \text{tal}\}$, enabling partial cross-lingual transfer. Panel C compares performance on rare words: FastText (blue) maintains high accuracy for words with < 100 corpus occurrences, while Word2Vec (red) degrades rapidly due to insufficient statistics. Panel D displays results on morphologically rich languages: FastText achieves 73% analogy accuracy on Turkish (vs. 52% for Word2Vec) and 69% on Finnish (vs. 48%), demonstrating the benefit of n-gram sharing for agglutinative morphology.

the sequence (or sum) of its subword embeddings. This approach combines advantages of word-level and character-level representations: frequent words are often single units (preserving semantic integrity), while rare words decompose into subwords (enabling OOV handling). BPE has become the standard tokenization method for modern LLMs (GPT, BERT, LLaMA) due to its vocabulary efficiency: a BPE vocabulary of 30,000–50,000 subwords covers virtually all text with minimal OOV rate (typically $< 0.1\%$), compared to word-level vocabularies requiring 100,000+ entries with 5–10% OOV. For embeddings, BPE units are learned during tokenization, then embedded via the same methods as words (Word2Vec, GloVe, or end-to-end training). The embedding matrix $\mathbf{E} \in \mathbb{R}^{|\mathscr{V}_{\text{BPE}}| \times d}$ has dimensionality $|\mathscr{V}_{\text{BPE}}| \approx 50{,}000$, intermediate between word-level ($|\mathscr{V}| \approx 50{,}000$ but with OOV issues) and character-n-gram ($N \approx 2{,}000{,}000$). BPE embeddings are contextual within the subword sequence but static across occurrences, inheriting the polysemy limitation of static embeddings. Modern LLMs learn BPE embeddings end-to-end with the language model objective, obviating separate pre-training, but the BPE vocabulary itself is typically fixed before training to ensure consistent tokenization.

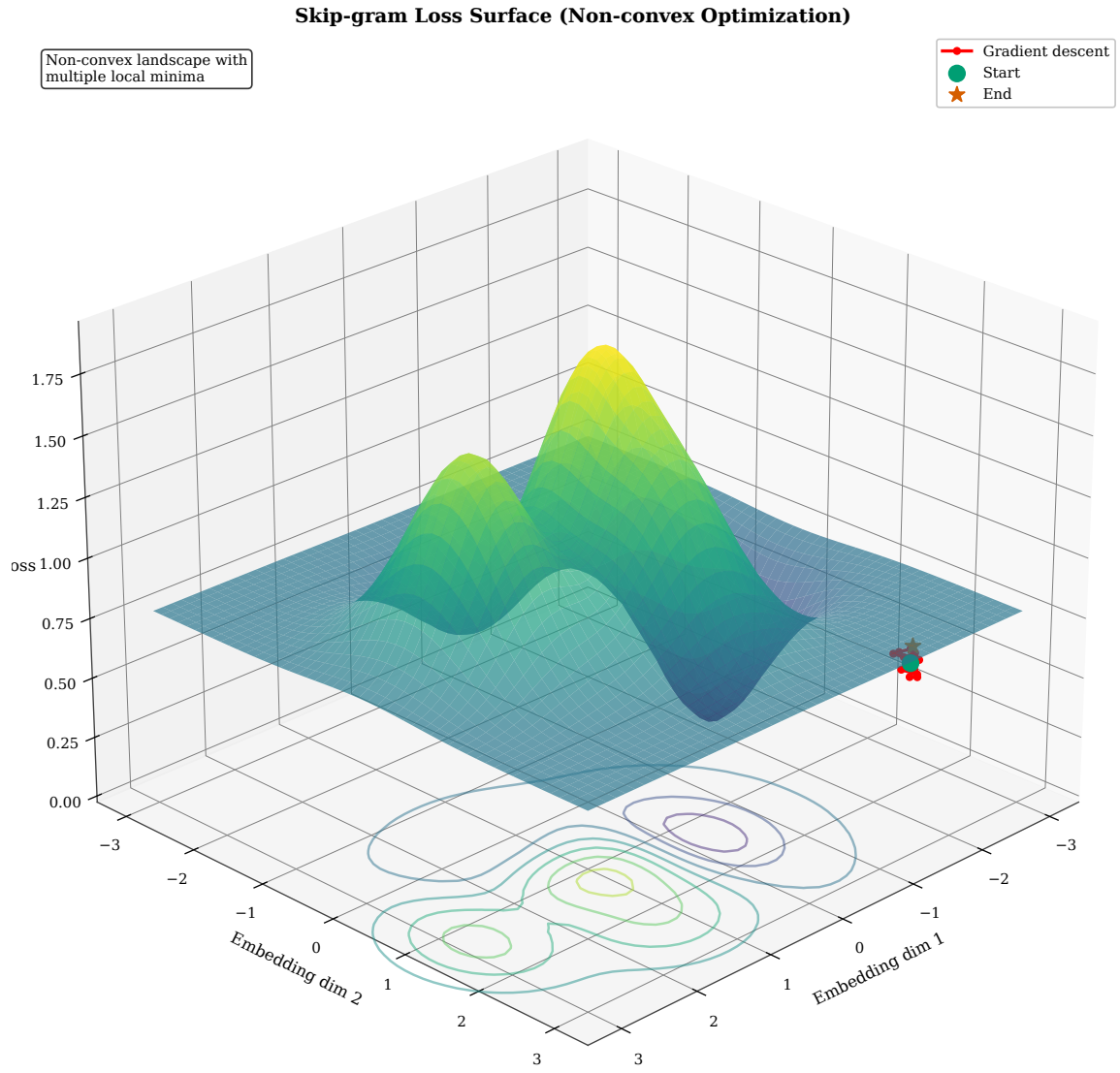


Figure 4.20: Comparison of word-level, FastText, and BPE embeddings across key dimensions. Word-level embeddings (blue) have vocabulary size $|\mathcal{V}| \approx 50{,}000$, suffer 5–10% OOV rate, and store $|\mathcal{V}| \times d \approx 15$M parameters. FastText (orange) has $N_{\text{ngrams}} \approx 2{,}000{,}000$ subunits, achieves near-zero OOV via character n-grams, but requires $2{,}000{,}000 \times d \approx 600$M parameters. BPE (green) balances with $|\mathcal{V}_{\text{BPE}}| \approx 50{,}000$ subwords, $< 0.1\%$ OOV, and $50{,}000 \times d \approx 15$M parameters. Trade-offs: word-level is simplest but has OOV problems; FastText handles OOV but is memory-intensive; BPE achieves low OOV with moderate memory, making it the preferred choice for modern LLMs.

4.7 Using Embeddings in Language Models

Word embeddings serve as the input layer for neural language models, converting discrete token sequences into continuous representations that subsequent layers process. The embedding layer is a lookup table: given a sequence of token IDs $[w_1, w_2, \ldots, w_T] \in \{1, \ldots, |\mathcal{V}|\}^T$, it produces a sequence of embedding vectors $[\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_T] \in \mathbb{R}^{T \times d}$ by indexing the embedding matrix $\mathbf{E} \in \mathbb{R}^{|\mathcal{V}| \times d}$. These embeddings are then fed to recurrent networks (Chapter **??**), convolutional networks, or Transformers (Chapter **??**) that aggregate them to predict the next word. A critical design choice is whether to initialize the embedding layer with pre-trained embeddings (Word2Vec, GloVe, FastText) or learn embeddings from scratch jointly with the language model objective. Pre-trained embeddings were standard practice from 2015–2018, providing a warm start that accelerated convergence and improved performance, especially with limited training data. However, modern large-scale language models (GPT, BERT, LLaMA) train embeddings end-to-end from random initialization, finding that massive data and compute budgets allow learning high-quality embeddings tailored to the prediction task without separate pre-training. Another important technique is weight tying, where the input embedding matrix $\mathbf{E}^{\text{in}}$ and output embedding matrix $\mathbf{E}^{\text{out}}$ share parameters, reducing the parameter count by $|\mathcal{V}| \times d$ (often 20–40% of total parameters) and providing a regularization effect that empirically improves performance. This section examines these architectural choices and their impact on next-word prediction quality.

4.7.1 Embedding Layer in Neural LMs

The embedding layer maps discrete token indices to continuous vectors, serving as the critical interface between symbolic input and neural computation. Given input sequence $\mathbf{w} = [w_1, \ldots, w_T]$ where $w_t \in \{1, \ldots, |\mathcal{V}|\}$ is the integer index of the t-th token, the embedding layer performs the lookup $\mathbf{e}_t = \mathbf{E}[w_t, :]$ for $t = 1, \ldots, T$, producing the embedding sequence $[\mathbf{e}_1, \ldots, \mathbf{e}_T]$ that subsequent neural layers process. This operation is mathematically equivalent to multiplying the one-hot vector $\mathbf{1}_{w_t}$ by the embedding matrix: $\mathbf{e}_t = \mathbf{E}^\top \mathbf{1}_{w_t}$, but direct indexing is computationally faster because it avoids the unnecessary multiplication with zeros. The embedding layer contains $|\mathcal{V}| \times d$ parameters; for $|\mathcal{V}| = 50{,}000$ and $d = 512$, this is 25.6 million parameters. For comparison, a 12-layer Transformer with $d = 512$ and $d_{\text{ff}} = 2048$ has approximately 110 million parameters, so embeddings constitute about 23% of the total. The embedding layer is fully differentiable: during backpropagation, gradients $\frac{\partial \mathcal{L}}{\partial \mathbf{e}_t}$ flow back from the language model, and the parameter gradient is $\frac{\partial \mathcal{L}}{\partial \mathbf{E}[w_t, :]} = \frac{\partial \mathcal{L}}{\partial \mathbf{e}_t}$, updating only the row corresponding to w_t. This sparse update is efficient: in a minibatch of B sequences of length T, at most $B \times T$ rows of $\mathbf{E}$ receive gradients, leaving most rows untouched. This sparsity is both a blessing (efficiency) and a curse (rare words receive few updates). The choice between pre-trained and randomly initialized embeddings depends on data availability. Pre-trained embeddings from Word2Vec or GloVe, learned on large corpora (Wikipedia, Common Crawl), transfer semantic knowledge to the language model. This transfer is especially beneficial when the LM training corpus is small (< 100M tokens) or domain-specific. The embeddings can be frozen (no updates during LM training), fine-tuned (initialized from pre-trained but updated), or used only as initialization (full training). Fine-tuning typically performs best, leveraging pre-trained semantics while adapting to the LM task. However, modern LLMs train on billions to trillions of tokens, providing sufficient data to learn embeddings from scratch. Random initialization allows the model to learn embeddings optimized specifically for next-word prediction, rather than general similarity, which can improve performance when data is abundant. Current best practice for models at scale is random initialization with joint training.

4.7.2 Weight Tying and Output Embeddings

Language models require two embedding-related matrices: the input embedding matrix $\mathbf{E}^{\text{in}} \in \mathbb{R}^{|\mathcal{V}| \times d}$ that converts tokens to vectors, and the output projection matrix $\mathbf{E}^{\text{out}} \in \mathbb{R}^{d \times |\mathcal{V}|}$ (or equivalently $\mathbb{R}^{|\mathcal{V}| \times d}$ transposed) that converts the final hidden state $\mathbf{h}_t \in \mathbb{R}^d$ to logits over vocabulary: $\mathbf{s}_t = \mathbf{E}^{\text{out}} \mathbf{h}_t \in \mathbb{R}^{|\mathcal{V}|}$. These two matrices serve different purposes (input encoding vs. output decoding) and traditionally were learned independently, requiring $2 \times |\mathcal{V}| \times d$ parameters total. Weight tying shares these matrices by setting $\mathbf{E}^{\text{out}} = (\mathbf{E}^{\text{in}})^\top$, reducing parameters by $|\mathcal{V}| \times d$ (a 50% reduction for embedding-related parameters), meaning that with weight tying the output logits are computed as $\mathbf{s}_t = (\mathbf{E}^{\text{in}})^\top \mathbf{h}_t$ where $\mathbf{s}_t[w]$ represents the unnormalized score for word w. Intuitively,

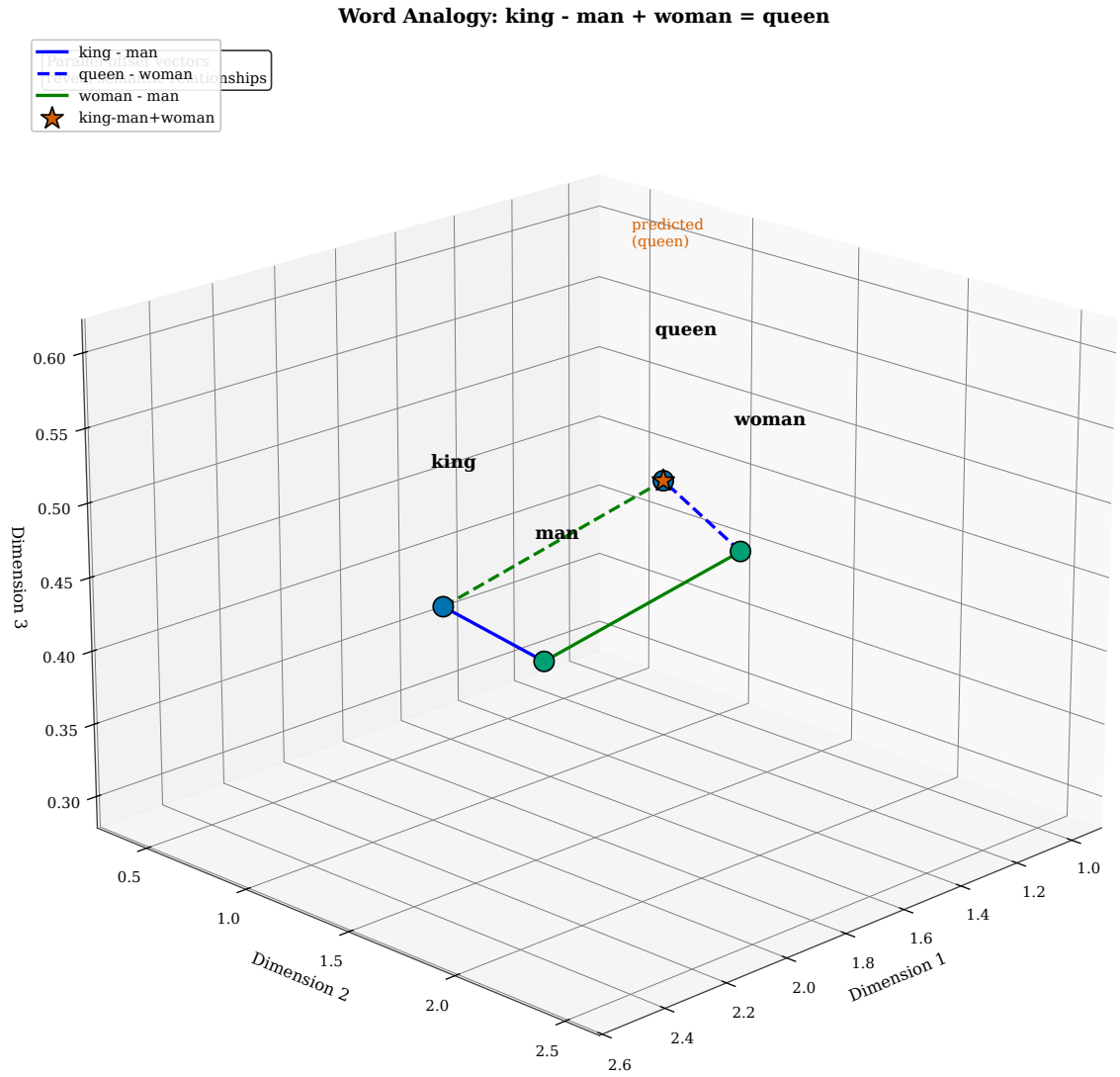


Figure 4.21: Embedding layer in neural language models. Panel A shows the architecture: token IDs $[15,342,27,891]$ are converted to embeddings $[\mathbf{e}_{15},\mathbf{e}_{342},\mathbf{e}_{27},\mathbf{e}_{891}]$ via lookup in $\mathbf{E}\in\mathbb{R}^{50000\times 512}$, then fed to the language model (RNN or Transformer). Panel B visualizes the embedding matrix as a heatmap: each row is a $d=512$-dimensional vector, with 50,000 rows (one per vocabulary word). Panel C compares convergence curves for pre-trained (blue) vs. random (orange) initialization: pre-trained embeddings provide faster initial convergence and lower final perplexity for small training corpora (< 100M tokens), but the gap narrows for large corpora (> 1B tokens) where random initialization learns task-specific representations. Panel D breaks down parameter counts: for a 110M parameter Transformer, embeddings ($|\mathcal{V}|\times d=25$M) account for 23%, while attention layers (40M), feed-forward layers (42M), and layer norms (3M) constitute the remainder.

weight tying enforces that words with similar input embeddings also have similar output logits, which is reasonable since words that appear in similar contexts (and thus have similar input embeddings) should also be predictable in similar contexts (and thus have similar output weights). Empirically, weight tying improves performance on language modeling benchmarks (reducing perplexity by 5–10%) and downstream tasks, likely due to a regularization effect: constraining output and input embeddings to be related reduces the model's capacity to overfit. The parameter reduction is substantial: for $|\mathcal{V}| = 50{,}000$ and $d = 512$, weight tying saves 25.6M parameters, which is significant for memory-constrained deployments. Weight tying requires that the hidden state dimension equals the embedding dimension ($d_{\mathbf{h}} = d_{\text{emb}}$); when these differ, a linear projection layer is inserted: $\mathbf{s}_t = (\mathbf{E}^{\text{in}})^\top \mathbf{W}_{\text{proj}} \mathbf{h}_t$ where $\mathbf{W}_{\text{proj}} \in \mathbb{R}^{d_{\text{emb}} \times d_{\mathbf{h}}}$. Modern Transformer language models (GPT-2, GPT-3, LLaMA) universally employ weight tying, considering it a best practice. The technique originated in Press and Wolf (2017) and was quickly adopted across the field. An additional benefit is interpretability: with weight tying, the output logit for word w is the dot product between the hidden state and the input embedding for w, providing a geometric interpretation of prediction as similarity in embedding space.

4.7.3 Improving Next-Word Prediction

Embeddings fundamentally improve next-word prediction by enabling generalization across similar words. In n-gram models (Chapter **??**), each context tuple received an independent probability distribution, with no parameter sharing. If the model observed "the cat sat on the mat" but not "the feline sat on the mat", it could not predict the latter. With embeddings, "cat" and "feline" have similar vector representations (cosine similarity ≈ 0.8), so contexts containing "cat" provide training signal for "feline". Mathematically, if the model learns parameters θ that predict well for contexts with "cat", and $\mathbf{e}_{\text{cat}} \approx \mathbf{e}_{\text{feline}}$, then the same parameters will predict reasonably for contexts with "feline" due to the continuous similarity. This generalization reduces data requirements: embeddings allow effective learning from smaller corpora by transferring knowledge across semantically related words. Quantitative improvements from embeddings are substantial. Bengio et al. (2003) demonstrated 30–50% perplexity reduction compared to n-gram baselines on the Penn Treebank benchmark. Modern comparisons show that even simple neural LMs with embeddings (single-layer LSTMs) outperform highly tuned n-gram models with Kneser-Ney smoothing. The dimensionality reduction from $|\mathcal{V}| \approx 50{,}000$ (one-hot) to $d \approx 300$ (embeddings) also reduces parameters dramatically: an n-gram model with trigram contexts requires storing $|\mathcal{V}|^3$ probabilities, while a neural LM with embeddings requires $|\mathcal{V}| \times d + O(d^2)$ parameters (embedding matrix plus model weights), a 99.9% reduction for typical vocabularies. This parameter efficiency translates to better generalization: fewer parameters mean less overfitting, especially on small datasets. The continuous nature of embeddings also enables gradient-based optimization, allowing end-to-end training of complex architectures (RNNs, Transformers) that would be intractable with discrete representations. In summary, embeddings serve as the foundational representation that enables modern neural language modeling, transforming next-word prediction from discrete lookup to continuous optimization in semantic space.

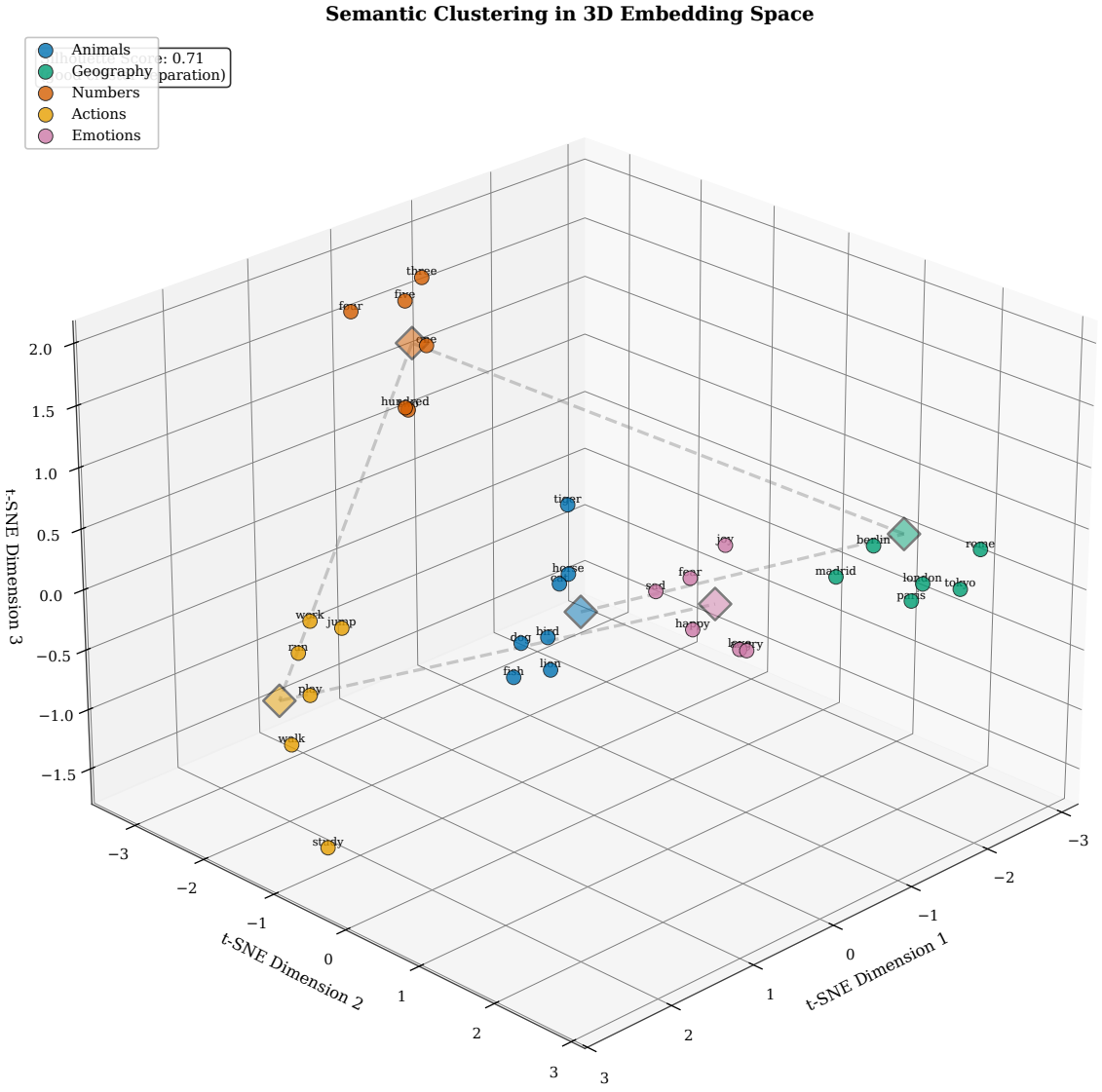


Figure 4.22: Weight tying between input and output embeddings. Without weight tying (left), the model maintains separate matrices $\mathbf{E}^{\text{in}} \in \mathbb{R}^{50000\times512}$ for input embeddings and $\mathbf{E}^{\text{out}} \in \mathbb{R}^{50000\times512}$ for output logits, totaling $2\times25.6\text{M} = 51.2\text{M}$ parameters. With weight tying (right), $\mathbf{E}^{\text{out}} = (\mathbf{E}^{\text{in}})^\top$, reducing to 25.6M parameters (50% reduction). The diagram shows token "cat" embedded via $\mathbf{E}^{\text{in}}$, processed by the LM to produce hidden state $\mathbf{h}_t$, then projected via $(\mathbf{E}^{\text{in}})^\top$ to compute logits. The parameter reduction visualization shows the freed memory (orange bars) when weight tying is applied, which can be reallocated to additional layers or larger hidden states, improving model capacity for a fixed parameter budget.

4.8 Context Representation in Word Embeddings

A fundamental question for next-word prediction is: how do we represent the context $w_1, \ldots, w_{t-1}$ to compute $P(w_t \mid w_1, \ldots, w_{t-1})$? In Chapter **??**, n-gram models represented context as a discrete tuple $(w_{t-n+1}, \ldots, w_{t-1})$ of the preceding $n-1$ words, treating each tuple as a separate symbolic key in a lookup table. This discrete representation provided no mechanism for generalization: similar contexts ("the black cat" vs. "the dark feline") were treated as completely unrelated. Word embeddings revolutionize context representation by mapping each word to a continuous vector $\mathbf{e}_w \in \mathbb{R}^d$, transforming the context sequence into a sequence of embeddings $[\mathbf{e}_1, \ldots, \mathbf{e}_{t-1}]$. This continuous encoding preserves semantic similarity: contexts containing similar words yield similar embedding sequences, enabling models to generalize across related contexts. However, embeddings alone do not solve the aggregation problem: we now have a variable-length sequence of vectors $[\mathbf{e}_1, \ldots, \mathbf{e}_{t-1}]$ but need a fixed-size representation to predict w_t. The methods in this chapter employ simple aggregation: CBOW averages context embeddings, and Skip-gram predicts each context word independently without aggregation. Neither approach captures sequential dependencies or long-range relationships. The polysemy problem remains: "bank" always receives the same embedding whether it means financial institution or river edge. These limitations motivate the recurrent architectures in Chapter **??**, where LSTM hidden states learn to compress variable-length embedding sequences into fixed-size context representations that preserve sequential structure and long-range dependencies. Transformers (Chapter **??**) further advance context representation through self-attention, computing contextual embeddings where each word's representation depends on all other words in the sequence, resolving polysemy by making "bank" receive different representations in "river bank" versus "savings bank".

How This Chapter Represents Context
The fundamental question in language modeling is: How do we represent the context $w_1, \ldots, w_{t-1}$ to predict w_t?

- **Context representation**: Word embeddings map discrete tokens to continuous vectors $\mathbf{e}_w \in \mathbb{R}^d$
- **Context encoding**: Each word in context has its own embedding; context is a sequence of embeddings $[\mathbf{e}_1, \ldots, \mathbf{e}_{t-1}]$
- **Limitation**: Static embeddings assign the same vector to each word occurrence regardless of context (polysemy problem)
- **Next chapter preview**: RNNs (Chapter **??**) will learn to aggregate embedding sequences into fixed-size context representations via recurrence

4.8.1 From N-grams to Embeddings

The progression from n-gram models to embedding-based models represents a fundamental shift in how context is encoded. In n-gram models, the context (w_{t-2}, w_{t-1}) for predicting w_t was represented by the tuple of word indices, say $(1523, 2847)$ if "black" is index 1523 and "cat" is index 2847. These indices are arbitrary integers with no inherent relationship: there is no mathematical reason why indices 1523 and 1524 should represent related concepts, and indeed they are typically assigned alphabetically or by frequency, destroying semantic structure. The model stores a separate probability distribution $P(\cdot \mid 1523, 2847)$ for each observed tuple. If the training corpus contains "the black cat sat" but not "the dark feline sat", the model has probability distributions for contexts $(1523, 2847)$ but not for $(1891, 3012)$ (assuming "dark" is 1891 and "feline" is 3012), forcing it to back off to unigram or bigram estimates with higher uncertainty. There is no mechanism to recognize that these contexts are semantically equivalent and should yield similar predictions. Word embeddings resolve this by mapping word indices to points in continuous semantic space $\mathbb{R}^d$. Now "black" maps to $\mathbf{e}_{1523} \in \mathbb{R}^d$ and "dark" maps to $\mathbf{e}_{1891} \in \mathbb{R}^d$, and crucially, these vectors are close together: $\|\mathbf{e}_{1523} - \mathbf{e}_{1891}\|$ is small (cosine similarity ≈ 0.7) because they appear in similar contexts. Similarly, $\mathbf{e}_{2847}$ (cat) and $\mathbf{e}_{3012}$ (feline) are nearby.

The context $(\text{black}, \text{cat})$ becomes the embedding sequence $[\mathbf{e}_{1523}, \mathbf{e}_{2847}]$, and the context $(\text{dark}, \text{feline})$ becomes $[\mathbf{e}_{1891}, \mathbf{e}_{3012}]$. Because these embedding sequences are close in $\mathbb{R}^d \times \mathbb{R}^d = \mathbb{R}^{2d}$, a model with continuous parameters can generalize: if it learns to predict "sat" after $[\mathbf{e}_{1523}, \mathbf{e}_{2847}]$, the same parameters will predict "sat" after $[\mathbf{e}_{1891}, \mathbf{e}_{3012}]$ due to the proximity in embedding space. This generalization is the key advantage: embeddings transform a discrete combinatorial space (where the number of possible $(n-1)$-word contexts is $|\mathcal{V}|^{n-1}$) into a continuous Euclidean space (where proximity implies similarity), enabling interpolation and parameter sharing across related contexts.

However, the transformation from discrete tuples to embedding sequences does not fully solve context representation. We now have a sequence of vectors $[\mathbf{e}_1, \ldots, \mathbf{e}_{t-1}]$ of varying length $t-1$, but most prediction models require fixed-size input. How do we aggregate this variable-length sequence into a single fixed-size context vector? The methods in this chapter employ naive aggregation strategies. CBOW averages the embeddings: $\bar{\mathbf{e}} = \frac{1}{t-1}\sum_{i=1}^{t-1} \mathbf{e}_i$, producing a single vector $\bar{\mathbf{e}} \in \mathbb{R}^d$ that represents the entire context. This averaging loses information about word order and long-range structure: the contexts "the cat sat on the mat" and "the mat sat on the cat" produce identical averaged embeddings despite opposite meanings. Skip-gram avoids aggregation entirely by predicting each context word independently: it uses $\mathbf{e}_{w_t}$ to predict w_{t+k} for each offset k, never forming a holistic context representation. While this enables efficient training, it does not provide a mechanism for encoding context as a whole. These limitations are acceptable for the local context windows ($K \leq 5$) used in Word2Vec and GloVe, but become severe for language modeling where context length can be hundreds or thousands of tokens. The polysemy problem remains unresolved: the word "bank" receives the same embedding $\mathbf{e}_{\text{bank}}$ in both "river bank" and "savings bank", despite the two meanings being unrelated. When "bank" appears in context, the model has no way to determine which sense is intended based solely on the static embedding. These challenges motivate the sequential architectures in Chapter **??**, where recurrent networks maintain a hidden state that evolves as the sequence is processed, learning to aggregate embeddings in a context-dependent manner that preserves word order and long-range dependencies while resolving polysemy through contextualization.

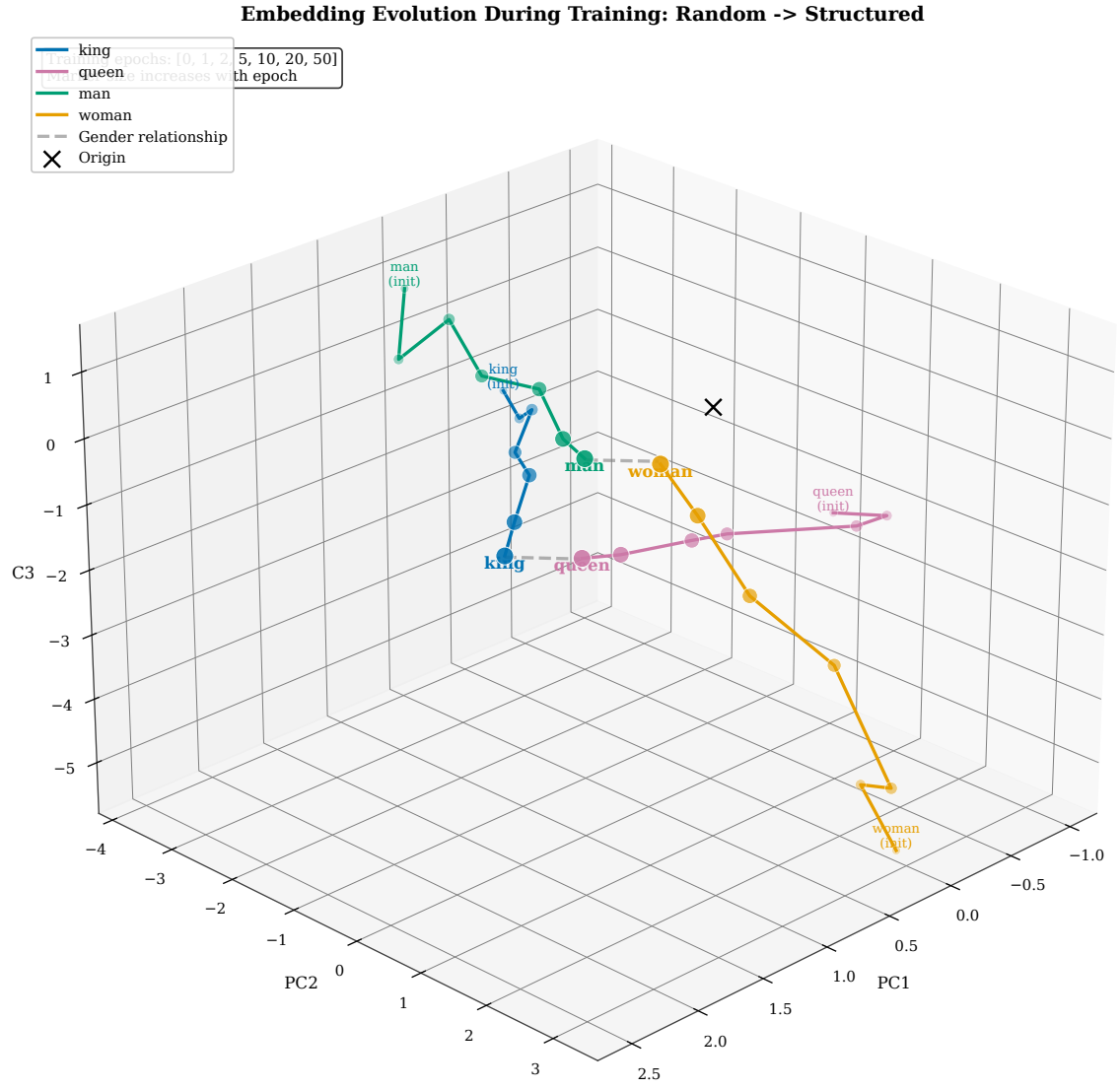


Figure 4.23: Evolution of context representation across chapters. Panel A shows n-gram discrete context tuples: "the black cat" is represented as integer indices $(3, 1523, 2847)$ with no similarity structure—"the dark feline" $(3, 1891, 3012)$ is treated as completely different despite semantic equivalence. Panel B displays word embedding sequences: the same contexts become $[\mathbf{e}_3, \mathbf{e}_{1523}, \mathbf{e}_{2847}]$ and $[\mathbf{e}_3, \mathbf{e}_{1891}, \mathbf{e}_{3012}]$, which are close in $\mathbb{R}^{3d}$ space due to similar individual word embeddings. Panel C illustrates CBOW's naive aggregation: averaging produces $\bar{\mathbf{e}} = (\mathbf{e}_3 + \mathbf{e}_{1523} + \mathbf{e}_{2847})/3$, collapsing the sequence to a fixed-size vector but losing word order. Panel D previews RNN context (Chapter 5): the hidden state $\mathbf{h}_{t-1}$ recursively compresses the entire history $[\mathbf{e}_1, \ldots, \mathbf{e}_{t-1}]$ into a fixed-size vector that preserves sequential dependencies, representing a qualitative advance in context encoding.

4.9 Summary

We can now predict better because:

- Similar words share similar embeddings, enabling generalization
- Vector arithmetic captures semantic relationships
- Dense representations reduce parameters by 99.9%
- Embeddings transfer learned knowledge across tasks

Next: Chapter **??** introduces recurrent neural networks to process embedding sequences...

This chapter introduced distributed representations as the solution to the discrete symbol problem that plagued n-gram models. By mapping words to continuous vectors in $\mathbb{R}^d$ where semantic similarity corresponds to geometric proximity, embeddings enable models to generalize across related contexts and dramatically reduce parameter counts. We examined three major approaches: matrix factorization methods (SVD on co-occurrence matrices) provide intuitive connections to distributional statistics but scale poorly; Word2Vec (Skip-gram and CBOW) learns embeddings efficiently through prediction tasks using negative sampling; and GloVe combines global co-occurrence statistics with prediction-based learning through weighted regression on log counts. All three methods produce embeddings exhibiting linear substructures where vector arithmetic captures semantic relationships, enabling analogy solving and demonstrating that geometry encodes meaning. FastText extends embeddings to character n-grams, enabling out-of-vocabulary handling and morphological generalization crucial for rare words and morphologically rich languages. We examined how embeddings integrate into neural language models as lookup tables that convert token sequences to vector sequences, with weight tying between input and output embeddings providing parameter reduction and performance improvements. Evaluation via intrinsic metrics (similarity, analogies) and extrinsic metrics (downstream tasks) confirms that embeddings substantially improve prediction quality, reducing perplexity by 30–50% compared to discrete representations. However, static embeddings suffer from the polysemy problem: each word type receives one embedding regardless of context, conflating multiple senses. They also encode societal biases present in training corpora, raising ethical concerns for deployment. These limitations motivate the contextual architectures in subsequent chapters: recurrent networks (Chapter **??**) will learn to aggregate embedding sequences into context-aware hidden states, and Transformers (Chapter **??**) will compute contextual embeddings where each word's representation depends dynamically on surrounding context, finally resolving polysemy.

Exercises

1. **One-hot Encoding Limitations.** Given vocabulary $\mathcal{V} = \{\text{cat}, \text{dog}, \text{fish}, \text{bird}\}$ with indices $\{1,2,3,4\}$, write the one-hot encoding vectors for "cat" and "dog". Compute their dot product $\mathbf{1}_{\text{cat}} \cdot \mathbf{1}_{\text{dog}}$ and cosine similarity. Explain why this representation cannot capture that "cat" and "dog" (both mammals) are more similar to each other than either is to "fish" (non-mammal), despite biological taxonomy.

2. **Distributional Hypothesis.** Consider the word "bank" in two sentences: "I deposited money at the bank" and "We sat on the river bank". Collect context words within a window of size $k = 2$ for each occurrence. Explain how the distributional hypothesis would suggest different meanings based on different contexts ($\{\text{deposited}, \text{money}, \text{at}\}$ vs. $\{\text{river}, \text{on}, \text{sat}\}$), and why static word embeddings fail to capture this polysemy, assigning a single vector $\mathbf{e}_{\text{bank}}$ that conflates both senses.

3. **Co-occurrence Matrix.** From the corpus: "the cat sat", "the cat ran", "the dog sat", construct the symmetric co-occurrence matrix $\mathbf{C}$ with window size $k = 1$. Compute the PMI for the pair ("cat", "sat") using $\text{PMI}(w_i, w_j) = \log \frac{C_{ij} \cdot N}{\sum_k C_{ik} \cdot \sum_k C_{kj}}$ where $N = \sum_{i,j} C_{ij}$. Explain what a positive PMI value indicates: that "cat" and "sat" co-occur more frequently than would be expected if they were statistically independent.

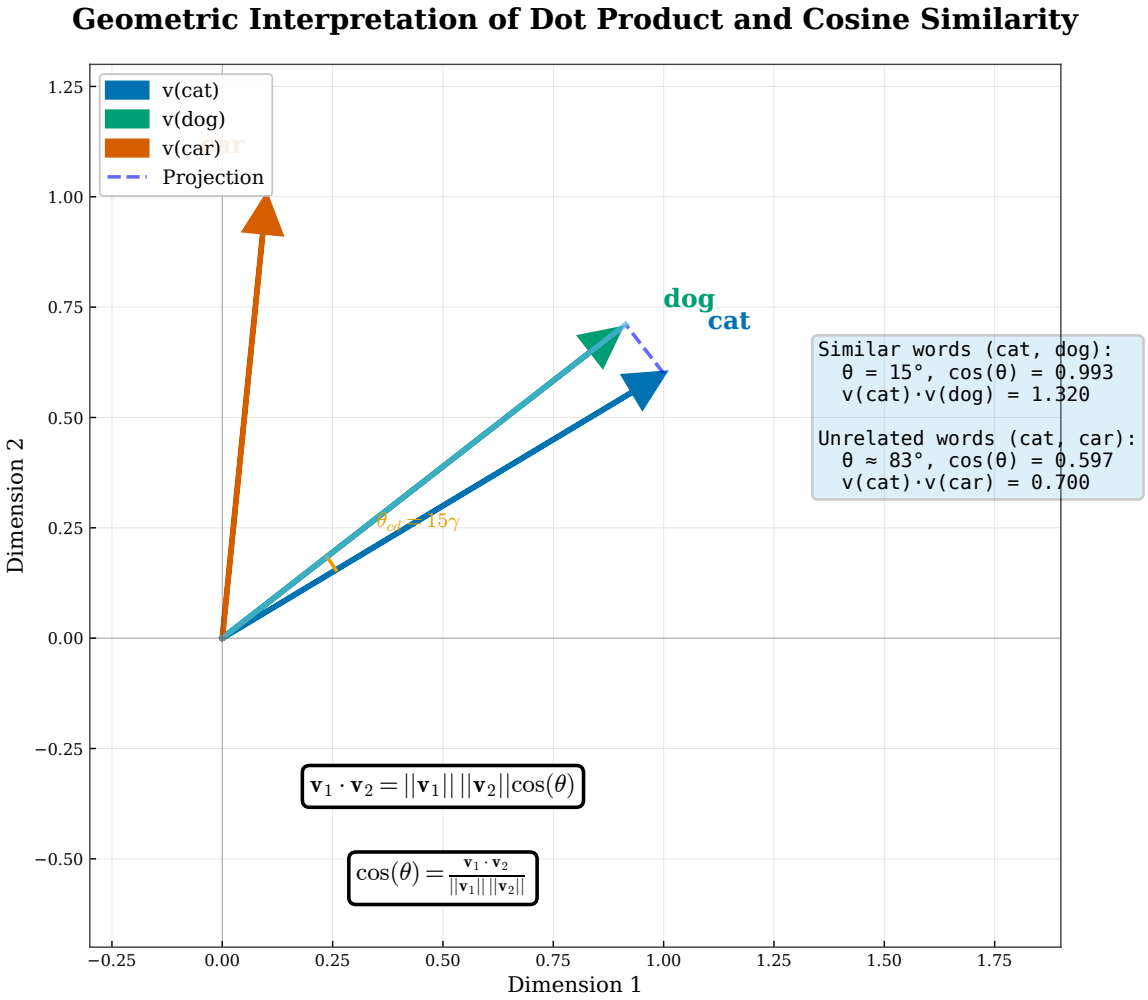


Figure 4.24: Geometric interpretation of dot product and cosine similarity.

4. **Skip-gram Training Pairs.** Given the sentence "the black cat sat there" with target word "cat" and context window $K = 2$, list all (target, context) training pairs generated by the Skip-gram architecture. Count the total number of pairs generated from this single sentence. Explain why Skip-gram generates $2K$ pairs per target word occurrence, providing richer supervision than CBOW which generates only one pair per target word.

5. **CBOW Forward Pass.** With embedding dimension $d = 2$, suppose context word embeddings are $\mathbf{e}_{\text{the}} = [0.2, 0.5]$, $\mathbf{e}_{\text{black}} = [0.3, 0.4]$, $\mathbf{e}_{\text{sat}} = [0.1, 0.6]$, and $\mathbf{e}_{\text{on}} = [0.4, 0.3]$. Compute the averaged context embedding $\bar{\mathbf{e}}$ for predicting target word "cat" in the sentence "the black cat sat on". If the output embedding for "cat" is $\mathbf{E}^{\text{out}}[\text{cat}, :] = [0.25, 0.45]$, compute the unnormalized score (dot product) $s_{\text{cat}} = \bar{\mathbf{e}} \cdot \mathbf{E}^{\text{out}}[\text{cat}, :]^\top$.

6. **Negative Sampling Efficiency.** Explain why computing the full softmax normalization $\sum_{v \in \mathcal{V}} \exp(s_v)$ over vocabulary size $|\mathcal{V}| = 50{,}000$ is computationally expensive, requiring $O(|\mathcal{V}| \cdot d)$ operations per training example. If we use negative sampling with $k = 5$ negative samples, how many words do we compute scores for per training example (positive plus negatives)? Calculate the computational speedup ratio $|\mathcal{V}|/(k+1)$ and explain why negative sampling achieves similar final performance to full softmax despite the approximation.

7. **GloVe Objective.** Given co-occurrence counts $X_{\text{cat,sat}} = 120$, $X_{\text{cat,ran}} = 60$, and GloVe weighting function $f(x) = (x/x_{\max})^\alpha$ with $x_{\max} = 100$ and $\alpha = 0.75$ for $x < x_{\max}$ (and $f(x) = 1$ for $x \geq x_{\max}$), compute the weights $f(X_{\text{cat,sat}})$ and $f(X_{\text{cat,ran}})$. Explain the rationale for this weighting: downweighting very

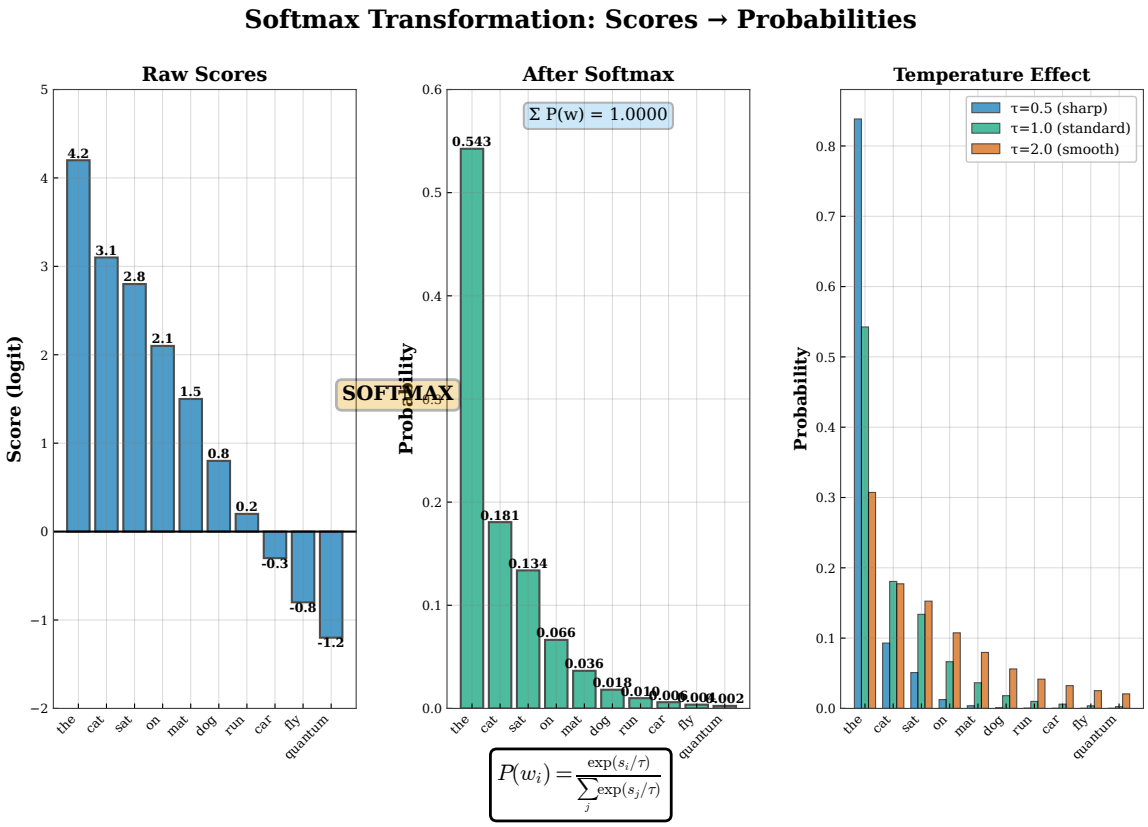


Figure 4.25: Softmax transformation over vocabulary.

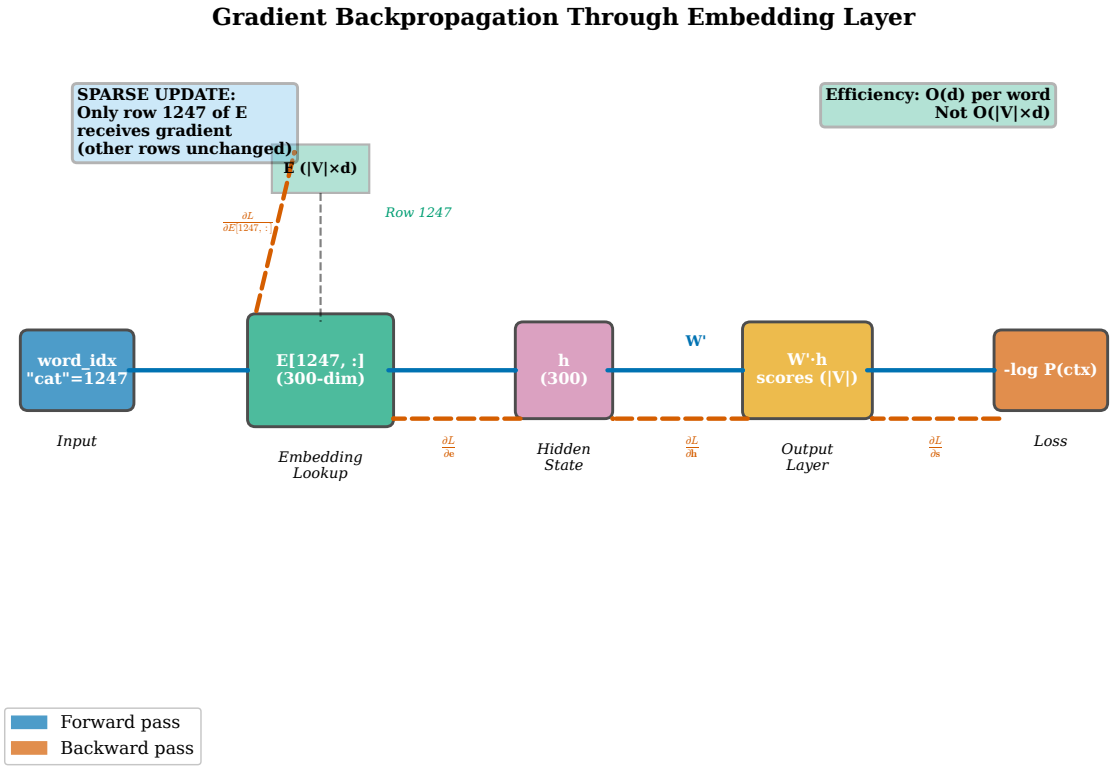


Figure 4.26: Backpropagation through embedding layer.

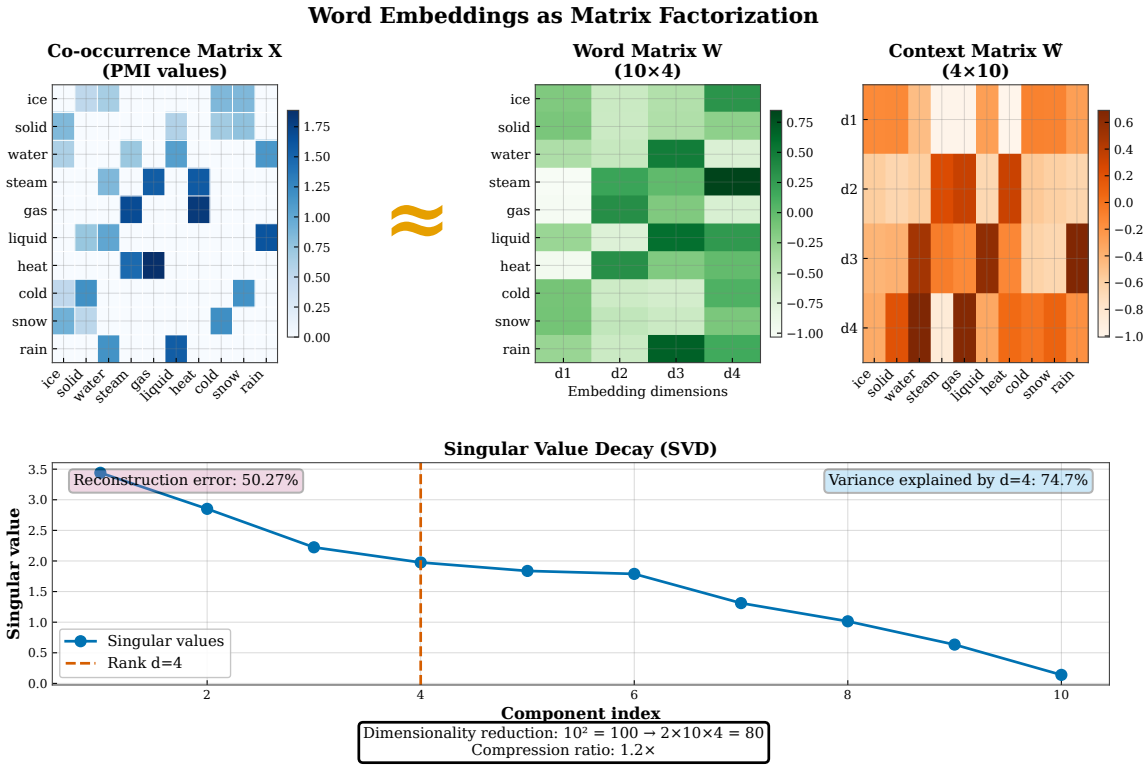


Figure 4.27: PMI matrix factorization connection.

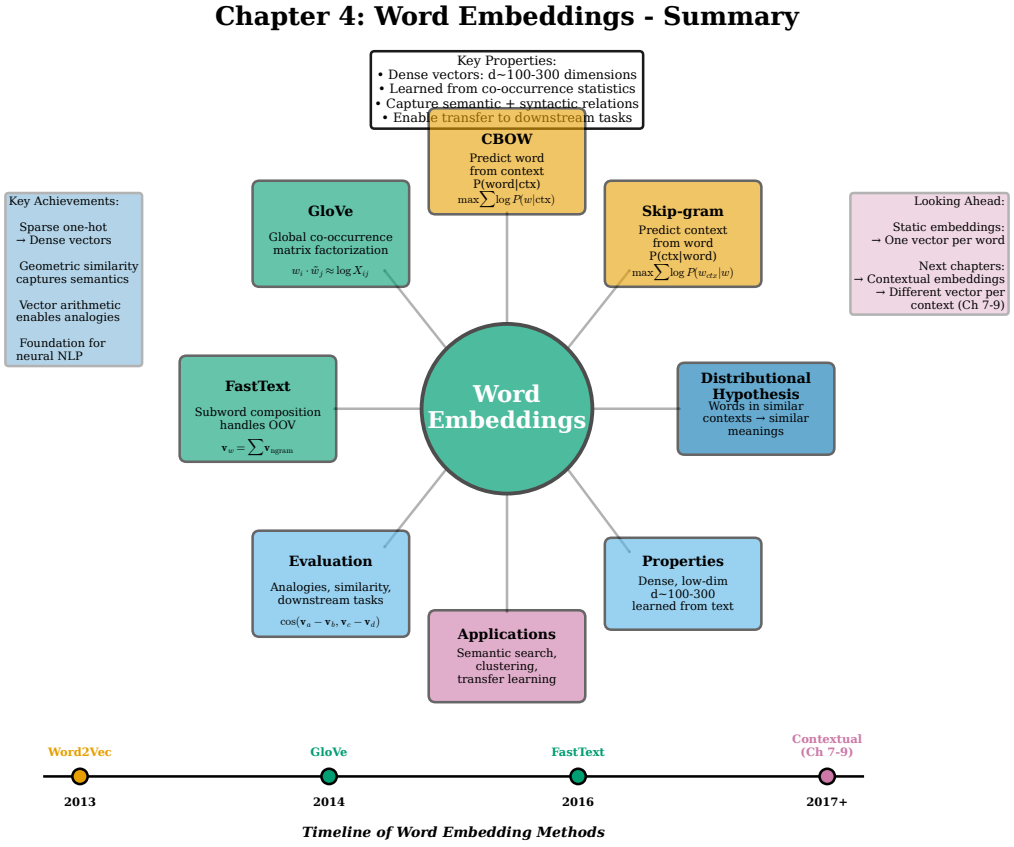


Figure 4.28: Visual summary of word embeddings chapter.

frequent co-occurrences (which are noisy and would dominate the loss) while upweighting moderate co-occurrences (which provide reliable signal).

8. **Vector Arithmetic and Analogies.** Given embeddings satisfying $\mathbf{e}_{\text{king}} - \mathbf{e}_{\text{man}} + \mathbf{e}_{\text{woman}} \approx \mathbf{e}_{\text{queen}}$, provide a geometric interpretation: the vector offset from "man" to "king" encodes a semantic direction ("royalty"), and adding this direction to "woman" arrives at the female royal equivalent "queen". Propose another analogy that should exhibit similar linear structure, such as verb tense (walk:walked::run:ran) or plurality (dog:dogs::cat:cats), and explain why distributional patterns would create these parallel vector offsets.

9. **Cosine Similarity Computation.** Compute cosine similarity between embedding vectors $\mathbf{e}_1 = [1,2,3]$ and $\mathbf{e}_2 = [2,4,6]$ using $\text{sim}(\mathbf{e}_1, \mathbf{e}_2) = \frac{\mathbf{e}_1 \cdot \mathbf{e}_2}{\|\mathbf{e}_1\| \|\mathbf{e}_2\|}$. Then compute cosine similarity between $\mathbf{e}_1 = [1,2,3]$ and $\mathbf{e}_3 = [1,0,0]$. Explain why cosine similarity (measuring angle) is preferred over Euclidean distance $\|\mathbf{e}_1 - \mathbf{e}_2\|$ (measuring magnitude) for semantic similarity: cosine is invariant to vector length, focusing on direction, while word frequency differences cause magnitude variation orthogonal to semantic content.

10. **FastText N-grams.** For the word "teaching" with character n-gram range $[3,4]$ and boundary markers $\langle$ and $\rangle$, list all character n-grams in $G(\text{teaching})$. Explain how this representation enables FastText to generate embeddings for out-of-vocabulary words like "unteaching" by composing shared n-gram embeddings: most n-grams in $G(\text{unteaching})$ (such as "tea", "each", "ach", "chi", "hin", "ing") appear in training words, allowing approximate representation via summation $\mathbf{e}_{\text{unteaching}} = \sum_{g \in G(\text{unteaching})} \mathbf{E}_{\text{ng}}[g,:]$.

11. **Embedding Dimensionality Trade-offs.** Discuss the trade-off between embedding dimension d and model performance. If increasing d from 100 to 1000 improves analogy accuracy from 60% to 75% on the Google Analogy Dataset, why don't we always use $d = 1000$ or even larger dimensions? Consider computational cost (memory scales as $|\mathcal{V}| \times d$, training time scales as $O(d^2)$ for subsequent layers), overfitting risk (higher capacity requires more data), and diminishing returns (performance saturates beyond $d \approx 300$ for most tasks).

12. **Bias in Embeddings.** Suppose word embeddings trained on a news corpus exhibit $\text{similarity}(\text{doctor}, \text{man}) = 0.72$ and $\text{similarity}(\text{doctor}, \text{woman}) = 0.58$, indicating gender bias. Explain how this bias arises from corpus statistics: if doctors are more frequently referred to with male pronouns ("he", "his") than female pronouns ("she", "her") in news articles, the distributional hypothesis causes "doctor" to be closer to "man" in embedding space. Discuss why this is problematic for downstream applications: a hiring algorithm using these embeddings might associate "doctor" more strongly with male candidates, perpetuating discrimination. Propose mitigation strategies such as debiasing (subtracting the gender direction), balanced corpus construction, or awareness audits.

13. **Pre-training vs. Random Initialization.** Compare two approaches for initializing the embedding layer of a language model: (1) pre-trained Word2Vec embeddings learned from Wikipedia, and (2) random initialization with training from scratch. Discuss advantages of each: pre-trained embeddings provide semantic knowledge and faster convergence, especially beneficial for small training corpora (< 100M tokens) or domain-specific tasks; random initialization allows learning embeddings optimized specifically for the prediction task and avoids domain mismatch, beneficial when large training corpora (> 1B tokens) are available. Consider computational budget: pre-training requires additional time but amortizes across multiple downstream tasks.

14. **Weight Tying Benefits.** A language model with vocabulary size $|\mathcal{V}| = 50{,}000$ and embedding dimension $d = 512$ uses separate input embedding matrix $\mathbf{E}^{\text{in}} \in \mathbb{R}^{50000 \times 512}$ and output embedding matrix $\mathbf{E}^{\text{out}} \in \mathbb{R}^{50000 \times 512}$. Calculate the total number of embedding-related parameters: $2 \times 50{,}000 \times 512 = 51.2\text{M}$. If we apply weight tying ($\mathbf{E}^{\text{out}} = (\mathbf{E}^{\text{in}})^\top$), how many parameters are saved? Explain why weight tying can improve generalization: it enforces consistency between input and output representations, providing a regularization effect that constrains the model's capacity and reduces overfitting, empirically reducing perplexity by 5–10% on standard benchmarks.

15. **From Static to Contextual Embeddings.** Static word embeddings assign the same vector to each occurrence of a word type, regardless of context. Explain why this is problematic for polysemous words like "bank" (financial institution vs. river edge), "lead" (metal Pb vs. verb to guide), or "bat" (animal vs. sports equipment). Discuss how the context determines which sense is intended: "savings bank" vs. "river bank". Preview how recurrent neural networks (Chapter **??**) address this limitation by maintaining hidden states $\mathbf{h}_t$ that depend on the entire preceding context $[\mathbf{e}_1, \ldots, \mathbf{e}_t]$, enabling context-dependent representations. Explain why RNN hidden states can resolve polysemy: $\mathbf{h}_t$ encodes not just $\mathbf{e}_{\text{bank}}$ but also the surrounding words, allowing the model to distinguish senses based on context.

Bibliography

Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. A neural probabilistic language model. *Journal of Machine Learning Research*, 3:1137–1155, 2003.

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

Jeffrey Pennington, Richard Socher, and Christopher D Manning. Glove: Global vectors for word representation. In *EMNLP*, pages 1532–1543, 2014.

Index

Predicting the Next Word

From Shannon to ChatGPT

Test Compilation - Chapter 5

Contents

Chapter 5

RNNs and LSTMs: Sequential Context for Next-Word Prediction

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

- Processing word sequences incrementally to build dynamic context representations
- Introducing hidden states that evolve as each word is encountered
- Solving the vanishing gradient problem through gated memory mechanisms
- Enabling models to capture long-range dependencies between distant words

5.1 From Static Embeddings to Sequential Context

The word embeddings developed in Chapter **??** represent a significant advance over discrete one-hot encodings: each word $w \in \mathcal{V}$ maps to a dense vector $\mathbf{e}_w \in \mathbb{R}^d$ where geometric proximity reflects semantic similarity. However, these embeddings remain fundamentally static—the vector for "bank" is identical whether it appears in "the river bank" or "the savings bank". The embedding captures general distributional properties learned across the entire corpus but cannot adapt to the specific context in which a word appears. For next-word prediction, this limitation is severe: the probability $P(w_{t+1} \mid w_1, \ldots, w_t)$ depends critically on the preceding context, yet static embeddings provide no mechanism to incorporate positional or sequential information. The n-gram models of Chapter **??** addressed context through fixed-size tuples, but these suffer from exponential parameter growth and the inability to generalize across similar contexts. What we need is a representation that evolves as we process the sequence word by word, accumulating information about the entire history into a fixed-size vector suitable for prediction. This chapter introduces recurrent neural networks (RNNs), which maintain a hidden state $\mathbf{h}_t$ that updates after each word, encoding the relevant information from $w_1, \ldots, w_t$ in a form suitable for predicting w_{t+1}.

5.1.1 The Aggregation Problem in Language Modeling

Consider the challenge of predicting the next word given a prefix of arbitrary length. For the sequence "The trophy that the athletes from the national team had won during the championship couldn't fit in the display case because it …", predicting whether the next word is "was" or "were" requires understanding that the subject is "trophy" (singular), not "athletes" or "team". This grammatical agreement spans sixteen words—far beyond any practical n-gram window. How can we construct a context representation $\mathbf{c}_t$ that captures the relevant information from $w_1, \ldots, w_t$ regardless of sequence length? One naive approach averages the word embeddings: $\mathbf{c}_t = \frac{1}{t}\sum_{i=1}^{t} \mathbf{e}_{w_i}$. This bag-of-words representation ignores word order entirely, treating "dog bites man" and "man bites dog" identically. A concatenation approach $\mathbf{c}_t = [\mathbf{e}_{w_1}; \ldots; \mathbf{e}_{w_t}]$ preserves order but

produces variable-length representations unsuitable for fixed-size neural network layers, and grows linearly with sequence length. Neither approach is satisfactory: we require a fixed-size representation that nonetheless captures sequential structure. The key insight is that we should process words incrementally, updating our context representation after each word rather than computing it from scratch. This leads to the recurrence relation $\mathbf{h}_t = f(\mathbf{h}_{t-1}, \mathbf{e}_{w_t})$, where the hidden state $\mathbf{h}_t$ summarizes the sequence $w_1, \ldots, w_t$ and f is a learned function that determines how to incorporate new information while preserving relevant history.

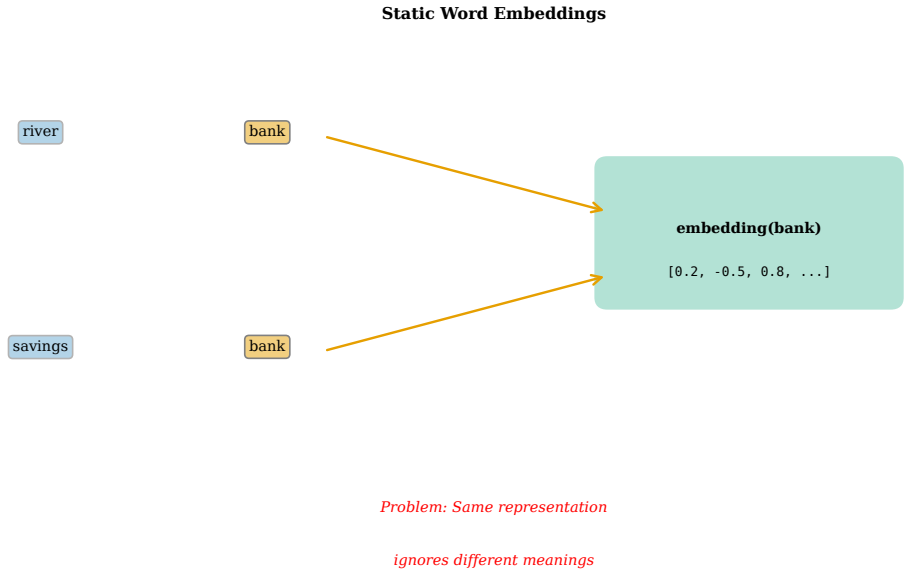


Figure 5.1: Static embeddings versus dynamic context representations. The left panel shows that static word embeddings assign identical vectors to "bank" regardless of context: both "river bank" and "savings bank" yield the same embedding, making disambiguation impossible at prediction time. The right panel illustrates the dynamic approach where recurrent processing builds context-dependent representations: the hidden state after "river" differs from the hidden state after "savings", enabling the model to disambiguate polysemous words based on their sequential context.

5.1.2 Sequential Processing as Solution

Sequential processing provides an elegant solution to the aggregation problem. Rather than computing context from all words simultaneously, we process the sequence left-to-right, maintaining a hidden state $\mathbf{h}_t \in \mathbb{R}^{d_h}$ that serves as a fixed-size summary of the words seen so far. At each time step t, the model receives the current word embedding $\mathbf{e}_{w_t}$ and the previous hidden state $\mathbf{h}_{t-1}$, then computes an updated hidden state $\mathbf{h}_t$ through a learned transformation. The hidden state can be thought of as the model's "memory" or "working representation" of the sequence: it must encode whatever information from the past is relevant for future predictions. The prediction at each step is then $P(w_{t+1} \mid \mathbf{h}_t)$, where the hidden state $\mathbf{h}_t$ implicitly encodes the entire history $w_1, \ldots, w_t$. This framework has several advantages. First, the representation size is constant: $\mathbf{h}_t \in \mathbb{R}^{d_h}$ regardless of whether $t = 5$ or $t = 500$. Second, the same parameters are used at every time step, enabling the model to generalize across positions and handle sequences longer than those seen during training. Third, the sequential nature respects the causal structure of language: we predict future words based only on past context, matching the left-to-right reading process. The challenge lies in learning a transition function that preserves relevant information across many time steps while forgetting irrelevant details—a balance that simple recurrent networks struggle to achieve but that gated architectures address effectively.

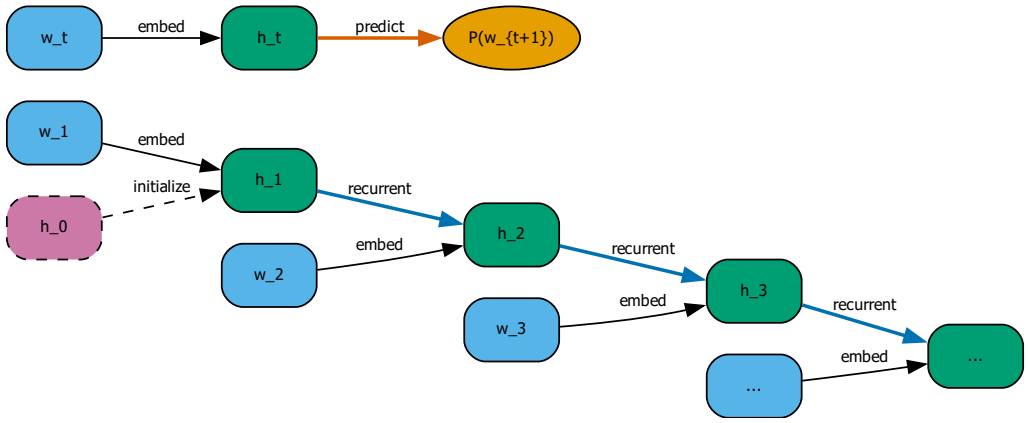


Figure 5.2: Sequential processing with hidden states. Words enter the model one at a time from left to right, each producing an updated hidden state. The initial hidden state $\mathbf{h}_0$ is typically set to zeros or learned as a parameter. After processing word w_t, the hidden state $\mathbf{h}_t$ encodes information about the entire prefix $w_1, \ldots, w_t$. The final hidden state feeds into a prediction layer that outputs $P(w_{t+1} \mid \mathbf{h}_t)$ as a distribution over the vocabulary. This architecture processes variable-length sequences while maintaining fixed-size internal representations.

5.1.3 Introduction of Running Example

Throughout this chapter, we examine a running example that illustrates the challenge of long-range dependencies: "The trophy that the athletes from the national team had won during the championship couldn't fit in the display case because it was too large." The critical dependency is between "trophy" at position 2 and "was" at position 18—a span of sixteen words. Predicting "was" (singular) rather than "were" (plural) requires the model to remember that the grammatical subject is "trophy", not the intervening nouns "athletes", "team", or "case". The nested relative clause "that the athletes from the national team had won during the championship" creates this challenging structure: many words intervene between the subject and its verb, and several plural nouns could mislead a model with limited memory. This example is not contrived—such long-range dependencies are pervasive in natural language, arising from relative clauses, coordination, embedding, and discourse structure. A successful language model must maintain relevant information (the singular subject) through the intervening material while appropriately ignoring irrelevant details. We will trace how vanilla RNNs fail on this example due to vanishing gradients, and how LSTM's gating mechanism succeeds by selectively preserving the subject information through the clause structure.

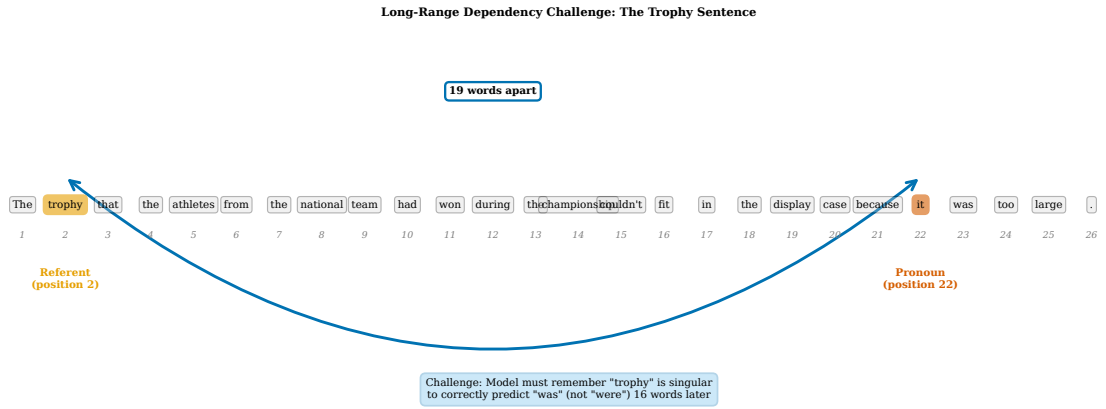


Figure 5.3: Running example sentence with long-range subject-verb dependency. The sentence “The trophy that the athletes from the national team had won during the championship couldn’t fit in the display case because it was too large” requires remembering “trophy” (singular, position 2) to correctly predict “was” (position 18). The intervening relative clause contains multiple plural nouns (athletes, team) that could mislead the model. The curved arrow indicates the grammatical dependency spanning 16 words—far beyond typical n-gram windows. This dependency is typical of complex English sentences and requires effective long-range memory mechanisms.

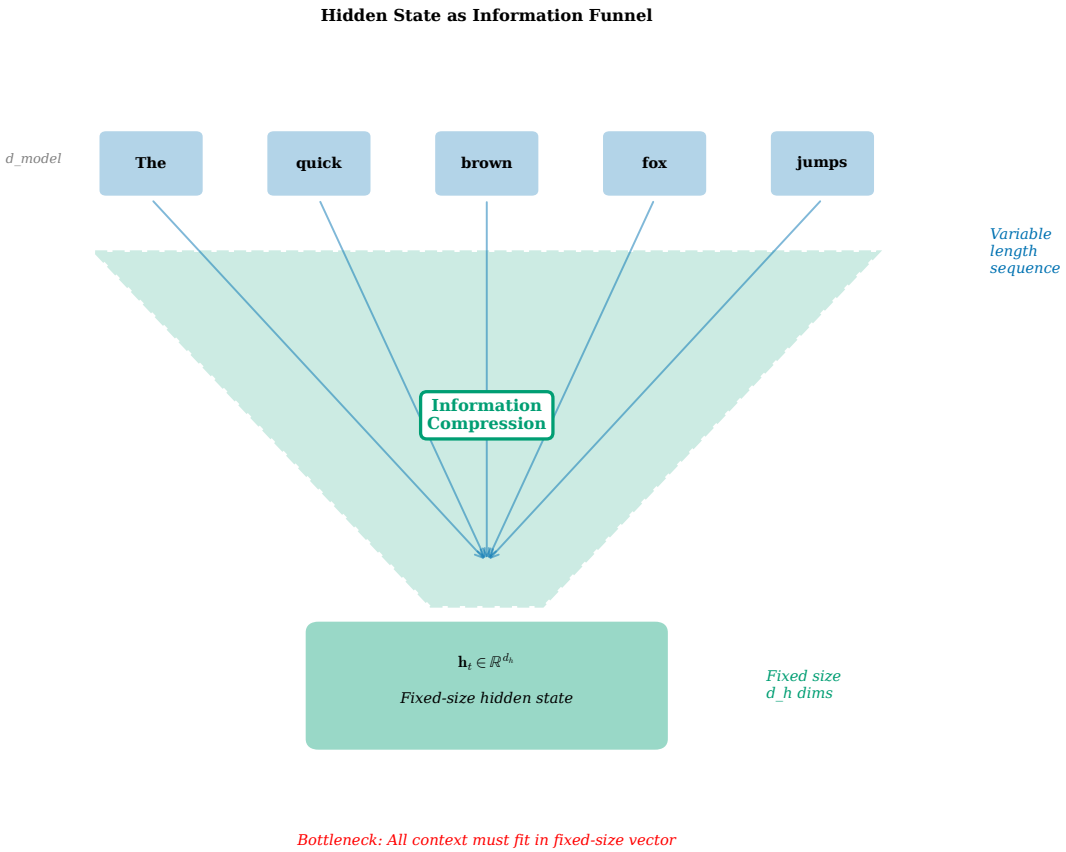


Figure 5.4: Hidden state as compressed sequence representation. The funnel visualization shows how a variable-length sequence of word embeddings is progressively compressed into a fixed-size hidden state vector. Each word contributes information to the hidden state, which must encode the relevant aspects of the entire history in d_h dimensions regardless of sequence length. This compression is both the strength and the limitation of recurrent architectures: it enables processing of arbitrary-length sequences but requires the model to learn which information to preserve and which to discard.

5.2 Vanilla RNN Architecture

The vanilla recurrent neural network (RNN) implements sequential processing through a simple recurrence relation. At each time step t, the hidden state $\mathbf{h}_t$ is computed as a function of the previous hidden state $\mathbf{h}_{t-1}$ and the current input $\mathbf{e}_{w_t}$. The update equation combines these inputs through learned weight matrices and a nonlinear activation function. Despite its simplicity, this architecture introduced the fundamental concept of recurrent computation for sequence modeling: the same parameters are applied at every time step, enabling the network to generalize across positions and handle variable-length inputs. The vanilla RNN was the dominant sequence model from the late 1980s through the early 1990s, applied successfully to tasks with short-range dependencies such as phoneme recognition and simple language modeling. However, researchers quickly discovered that training vanilla RNNs on tasks requiring long-range dependencies proved extremely difficult. The vanishing gradient problem, which we examine in Section 5.2.3, causes the gradient signal to decay exponentially as it propagates backward through time, making it nearly impossible to learn dependencies spanning more than five to ten time steps. Understanding the vanilla RNN architecture is essential both for historical context and because it clearly illustrates why gated architectures like LSTM are necessary for effective sequence modeling.

5.2.1 The RNN Cell

The basic RNN cell computes the hidden state update through a linear transformation followed by a nonlinear activation, implementing the core recurrence that enables sequential processing. Let $\mathbf{e}_t = \mathbf{e}_{w_t} \in \mathbb{R}^d$ denote the embedding of word w_t, and let $\mathbf{h}_{t-1} \in \mathbb{R}^{d_h}$ denote the previous hidden state, which encodes the model's memory of all words processed so far. The vanilla RNN update equation combines these two sources of information through learned weight matrices: $\mathbf{h}_t = \tanh(\mathbf{W}_{xh}\mathbf{e}_t + \mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{b}_h)$, where $\mathbf{W}_{xh} \in \mathbb{R}^{d_h \times d}$ maps input embeddings to hidden space, $\mathbf{W}_{hh} \in \mathbb{R}^{d_h \times d_h}$ maps previous hidden state to current hidden space, $\mathbf{b}_h \in \mathbb{R}^{d_h}$ is a bias vector, and tanh is applied element-wise. The hyperbolic tangent function $\tanh(x) = (e^x - e^{-x})/(e^x + e^{-x})$ squashes the output to the range $[-1, 1]$, introducing nonlinearity while keeping activations bounded. This bounded activation is important for training stability: without it, hidden state values could grow unboundedly through the recurrence, causing numerical overflow. For prediction, the hidden state is projected to vocabulary size through an output layer: $\mathbf{y}_t = \text{softmax}(\mathbf{W}_{hy}\mathbf{h}_t + \mathbf{b}_y)$, where $\mathbf{W}_{hy} \in \mathbb{R}^{|\mathscr{V}| \times d_h}$ and $\mathbf{y}_t \in \mathbb{R}^{|\mathscr{V}|}$ gives the predicted probability distribution over next words. The total parameter count for a vanilla RNN is $d \cdot d_h + d_h^2 + d_h + |\mathscr{V}| \cdot d_h + |\mathscr{V}|$: linear in vocabulary size and quadratic in hidden dimension. For typical values $d = 300$, $d_h = 256$, $|\mathscr{V}| = 50{,}000$, this gives approximately 13 million parameters, dominated by the output projection. The recurrence is initialized with $\mathbf{h}_0 = \mathbf{0}$ (zeros) or a learned initial state, and the same parameters are used at every time step, enabling the model to generalize across positions.

5.2.2 Unrolling and Parameter Sharing

To understand how RNNs process sequences and how gradients flow during training, we unroll the recurrence relation through time. For a sequence of T words, unrolling produces a computational graph with T copies of the RNN cell, connected in series through the hidden states. At step 1, $\mathbf{h}_1 = \tanh(\mathbf{W}_{xh}\mathbf{e}_1 + \mathbf{W}_{hh}\mathbf{h}_0 + \mathbf{b}_h)$; at step 2, $\mathbf{h}_2 = \tanh(\mathbf{W}_{xh}\mathbf{e}_2 + \mathbf{W}_{hh}\mathbf{h}_1 + \mathbf{b}_h)$; and so on through step T. Critically, the same weight matrices $\mathbf{W}_{xh}$, $\mathbf{W}_{hh}$, and $\mathbf{W}_{hy}$ are used at every time step—this parameter sharing is the defining characteristic of recurrent architectures. Parameter sharing provides several benefits. First, it reduces the number of parameters dramatically: instead of T separate sets of weights, we have a single shared set, enabling the model to handle variable-length sequences without parameter explosion. Second, it enables generalization across positions: the model learns position-invariant features that apply regardless of where a word appears in the sequence. Third, it allows processing sequences longer than those seen during training, since the same cell applies recursively. The unrolled view reveals that RNNs are simply deep feedforward networks with shared weights across layers, where "depth" corresponds to sequence length. This perspective is crucial for understanding gradient flow: during backpropagation, gradients must traverse the entire unrolled graph, passing through T multiplicative operations involving $\mathbf{W}_{hh}$. The repeated multiplication by the same matrix is the source of both exploding and vanishing gradients.

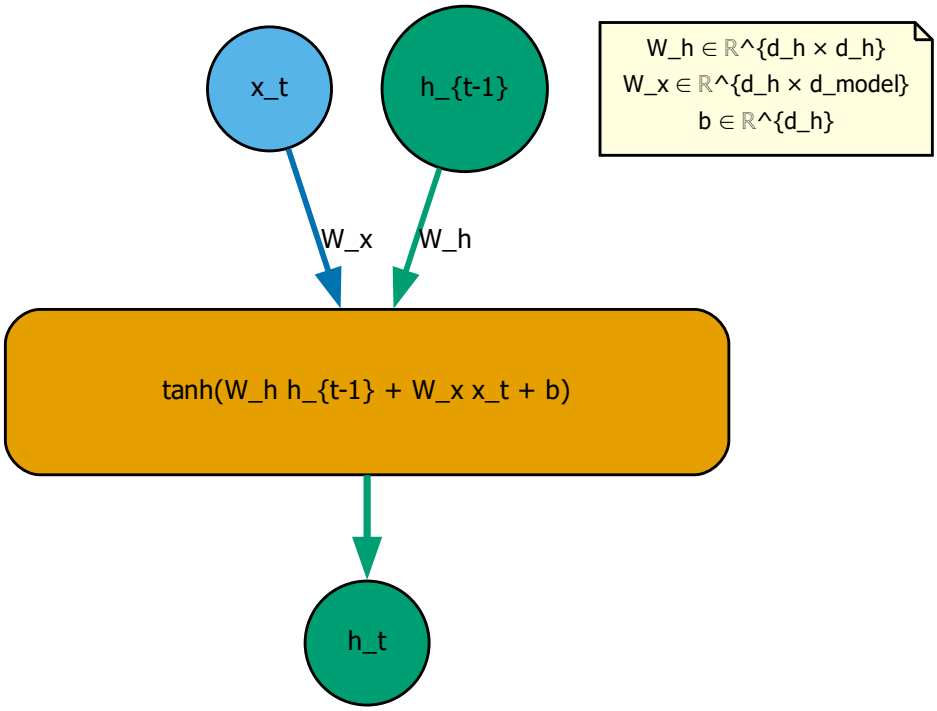


Figure 5.5: Vanilla RNN cell architecture. The cell receives two inputs: the current word embedding $\mathbf{e}_t$ and the previous hidden state $\mathbf{h}_{t-1}$. These are linearly combined through weight matrices $\mathbf{W}_{xh}$ and $\mathbf{W}_{hh}$, with a bias $\mathbf{b}_h$ added. The tanh nonlinearity produces the new hidden state $\mathbf{h}_t$, which serves both as output and as input to the next time step. The output layer applies a linear projection $\mathbf{W}_{hy}$ followed by softmax to produce a probability distribution over vocabulary words for next-word prediction.

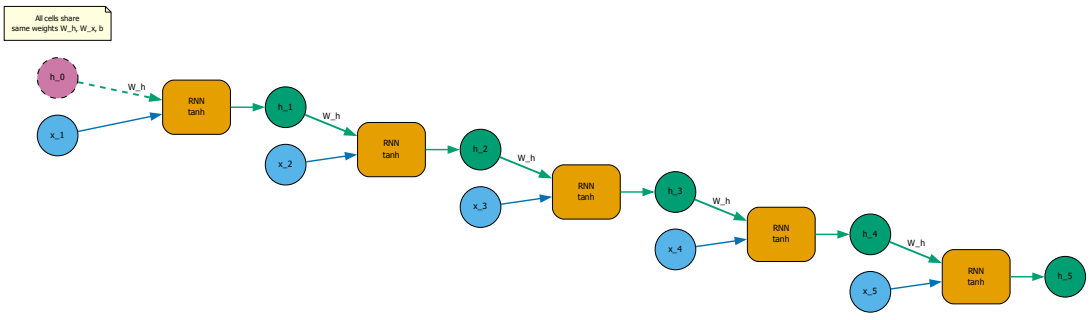


Figure 5.6: Unrolled RNN through five time steps. The recurrence relation is expanded into a computational graph where each time step corresponds to one copy of the RNN cell. Hidden states flow left-to-right through $\mathbf{h}_0 \to \mathbf{h}_1 \to \mathbf{h}_2 \to \mathbf{h}_3 \to \mathbf{h}_4 \to \mathbf{h}_5$, with word embeddings $\mathbf{e}_1$ through $\mathbf{e}_5$ entering at each step. The key feature is parameter sharing: the weight matrices $\mathbf{W}_{xh}$ and $\mathbf{W}_{hh}$ (shown with dashed lines) are identical across all time steps. During backpropagation, gradients flow right-to-left through this entire graph, multiplying by $\mathbf{W}_{hh}^\top$ at each step.

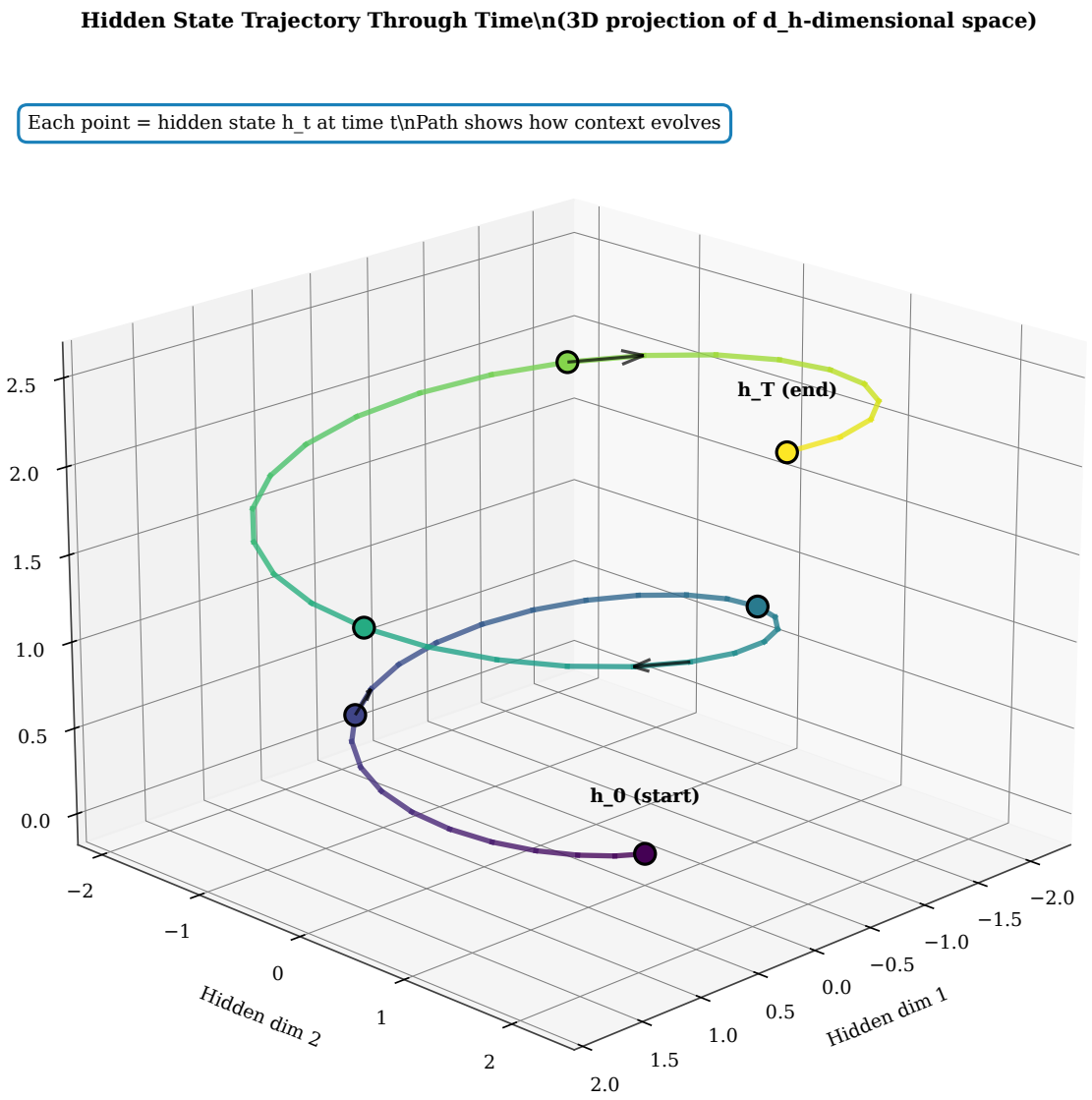


Figure 5.7: Hidden state trajectory in 3D embedding space. As the RNN processes a sequence, the hidden state traces a path through the high-dimensional hidden space (projected here to 3D via PCA). Each point represents the hidden state after processing a word, with color indicating position in the sequence from blue (early) to red (late). The trajectory shows how the hidden state evolves as context accumulates: early words establish the initial direction, while later words cause the trajectory to curve and shift. The final hidden state encodes the cumulative information from the entire sequence.

5.2.3 The Short Memory Problem

Despite their theoretical ability to capture dependencies of arbitrary length, vanilla RNNs in practice struggle to learn dependencies beyond five to ten time steps. This limitation arises from the vanishing gradient problem: during backpropagation through time (BPTT), gradients must pass through many multiplicative operations involving the recurrent weight matrix $\mathbf{W}_{hh}$, and these repeated multiplications cause gradients to either vanish (become negligibly small) or explode (become unmanageably large). Consider the gradient of the loss at time T with respect to the hidden state at time t: by the chain rule, this involves the product $\prod_{k=t+1}^{T} \frac{\partial \mathbf{h}_k}{\partial \mathbf{h}_{k-1}}$, which includes factors of the form $\text{diag}(\tanh'(\cdot))\mathbf{W}_{hh}$. If the largest singular value of $\mathbf{W}_{hh}$ is less than 1, this product shrinks exponentially with $(T-t)$; if greater than 1, it grows exponentially. For our running example with "trophy" at position 2 and "was" at position 18, the gradient must survive 16 multiplicative steps. With typical weight initialization, the gradient contribution from position 2 is 10^{-4} or smaller by position 18—effectively zero for learning purposes. The model cannot learn the association between "trophy" and "was" because the gradient signal is too weak to drive parameter updates. We can observe this failure empirically: when processing our running example, the vanilla RNN's hidden state "forgets" the subject "trophy" after a few words, and by position 17 contains almost no information distinguishing singular from plural subjects. The prediction at position 18 is essentially random between "was" and "were". This short memory problem motivated the development of gated architectures that can selectively preserve information across long sequences.

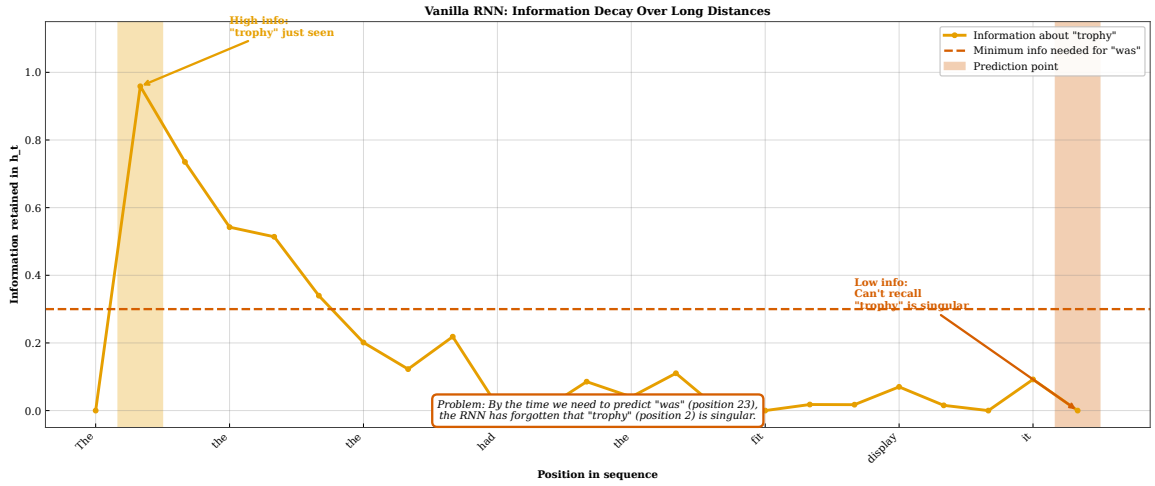


Figure 5.8: Information decay in vanilla RNN on the running example. The plot tracks how much information about "trophy" (position 2) is preserved in the hidden state as the sequence is processed. The y-axis shows a measure of "trophy information" derived from probing the hidden state's ability to distinguish singular versus plural subjects. In the vanilla RNN, this information decays exponentially: by position 10 it has dropped below 10%, and by position 18 (where "was" must be predicted) it is effectively zero. The shaded region indicates where the model cannot reliably predict the correct verb form. This failure illustrates the short memory problem that motivates LSTM architecture.

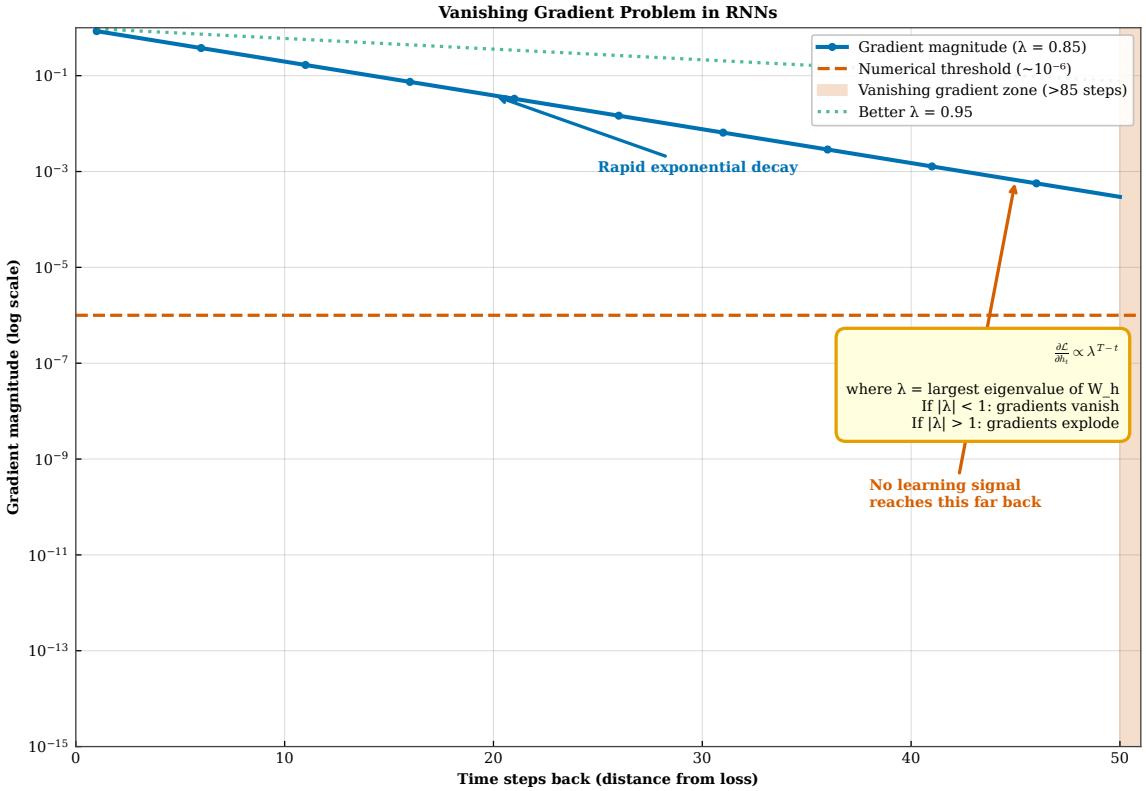


Figure 5.9: Gradient magnitude decay during backpropagation through time. The y-axis (log scale) shows the magnitude of the gradient $\frac{\partial \mathscr{L}}{\partial \mathbf{h}_t}$ as it propagates backward from the loss at position T. For vanilla RNNs, the gradient decays exponentially with temporal distance, losing 90% of its magnitude every 5-7 steps. By 15 steps back, the gradient is below 10^{-3}—too small to drive meaningful learning. The horizontal dashed line indicates the effective training threshold: gradients below this level contribute negligibly to parameter updates. Long-range dependencies require gradients to survive far beyond this threshold, which vanilla RNNs cannot achieve.

5.3 LSTM: Learning What to Remember and Forget

Long Short-Term Memory (LSTM) networks, introduced by Hochreiter and Schmidhuber (1997) [Hochreiter and Schmidhuber, 1997], address the vanishing gradient problem through a carefully designed gating mechanism. The key innovation is the cell state $\mathbf{c}_t$, a separate memory channel that runs parallel to the hidden state $\mathbf{h}_t$ and uses additive updates rather than multiplicative overwrites. Information flows along the cell state with minimal interference, protected by learnable gates that control what information enters, persists, and exits. The gates are neural network layers with sigmoid activation that produce values between 0 (completely blocked) and 1 (completely passed). By learning to open and close these gates appropriately, the LSTM can preserve relevant information across hundreds of time steps while forgetting irrelevant details. The LSTM architecture has three gates: the forget gate decides what to discard from the previous cell state, the input gate decides what new information to add, and the output gate decides what to expose in the hidden state. These gates are conditioned on the current input and previous hidden state, enabling context-dependent memory management. For our running example, the LSTM can learn to keep the forget gate high (near 1) for subject-related information while processing the relative clause, preserving "trophy" until it becomes relevant for predicting "was". This selective persistence is precisely what vanilla RNNs lack.

5.3.1 The Gating Intuition

Before examining LSTM equations, consider the intuition behind gates. A gate is a vector of values between 0 and 1, computed by a sigmoid function $\sigma(\cdot)$. When multiplied element-wise with another vector, the gate

selectively scales each dimension: values near 1 pass information through unchanged, while values near 0 block information. The crucial insight is that gates are learned, not fixed: the network discovers through training which information to preserve and which to discard based on task demands. Consider reading a sentence to answer a question about the subject. A well-trained gate would learn to preserve subject information (nouns in subject position) while forgetting modifiers and parentheticals that are irrelevant to subject-verb agreement. Different gates serve different functions: the forget gate controls persistence of old information, the input gate controls incorporation of new information, and the output gate controls visibility of stored information. The combination of these three gates provides fine-grained control over the memory cell's contents and their influence on predictions. The gating mechanism also enables gradient flow: when the forget gate is near 1, gradients pass through the cell state nearly unchanged, avoiding the repeated multiplication by weight matrices that causes vanishing gradients in vanilla RNNs. This gradient highway is the mathematical reason LSTMs can learn long-range dependencies that vanilla RNNs cannot.

5.3.2 LSTM Cell Architecture

The LSTM cell maintains both a hidden state $\mathbf{h}_t \in \mathbb{R}^{d_h}$ and a cell state $\mathbf{c}_t \in \mathbb{R}^{d_h}$, with the cell state serving as the primary long-term memory channel that can preserve information across many time steps. Given input $\mathbf{e}_t$ and previous states $\mathbf{h}_{t-1}$, $\mathbf{c}_{t-1}$, the update proceeds through computing three gates and a candidate cell state, each implemented as a simple neural network layer with sigmoid or tanh activation. The forget gate $\mathbf{f}_t = \sigma(\mathbf{W}_f[\mathbf{h}_{t-1};\mathbf{e}_t] + \mathbf{b}_f)$ determines what fraction of each cell state dimension to retain from the previous step, with values near 1 indicating preservation and values near 0 indicating deletion. The input gate $\mathbf{i}_t = \sigma(\mathbf{W}_i[\mathbf{h}_{t-1};\mathbf{e}_t] + \mathbf{b}_i)$ determines what fraction of new information to incorporate into the cell state, controlling the flow of new content. The candidate $\tilde{\mathbf{c}}_t = \tanh(\mathbf{W}_c[\mathbf{h}_{t-1};\mathbf{e}_t] + \mathbf{b}_c)$ represents the potential new cell content computed from current inputs, and the output gate $\mathbf{o}_t = \sigma(\mathbf{W}_o[\mathbf{h}_{t-1};\mathbf{e}_t] + \mathbf{b}_o)$ determines what fraction of the cell state to expose in the hidden state output. In all these equations, $[\mathbf{h}_{t-1};\mathbf{e}_t]$ denotes concatenation of the previous hidden state and current input embedding, $\mathbf{W}_f, \mathbf{W}_i, \mathbf{W}_c, \mathbf{W}_o \in \mathbb{R}^{d_h \times (d_h + d)}$ are learnable weight matrices, and $\mathbf{b}_f, \mathbf{b}_i, \mathbf{b}_c, \mathbf{b}_o \in \mathbb{R}^{d_h}$ are learnable bias vectors. Each gate receives the same concatenated inputs but learns different weights through training, enabling context-dependent decisions about what information to remember, update, and output at each position.

The cell state and hidden state are then updated through the core LSTM equations: $\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t$ for the cell state and $\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t)$ for the hidden state, where $\odot$ denotes element-wise (Hadamard) multiplication. The cell state update equation is the architectural innovation that gives LSTM its power: it is additive rather than purely multiplicative, combining retained old content ($\mathbf{f}_t \odot \mathbf{c}_{t-1}$) with gated new content ($\mathbf{i}_t \odot \tilde{\mathbf{c}}_t$) through summation rather than transformation. When $\mathbf{f}_t \approx 1$ and $\mathbf{i}_t \approx 0$, the cell state passes through nearly unchanged: $\mathbf{c}_t \approx \mathbf{c}_{t-1}$, enabling information to persist across arbitrarily long sequences without degradation. This additive structure creates a gradient highway that allows gradients to flow backward through time without the exponential decay caused by repeated matrix multiplication in vanilla RNNs, because the Jacobian of the cell state update with respect to the previous cell state is close to the identity matrix when the forget gate is high. The hidden state $\mathbf{h}_t$ is computed from the cell state through a nonlinearity and output gate, serving as the LSTM's output at each time step for prediction while the cell state maintains the long-term memory for future use.

5.3.3 Information Flow Through Gates

Understanding how information flows through LSTM gates is essential for interpreting what the network learns. Consider processing our running example sentence word by word. At "The" (position 1), all gates activate moderately: the forget gate is high because there is little prior context to preserve, the input gate opens to accept the determiner, and the output gate prepares to emit a hidden state. At "trophy" (position 2), the input gate activates strongly for dimensions encoding noun information and singular number, writing this subject information to the cell state. The forget gate remains high, preserving the determiner context. As we process the relative clause "that the athletes from the national team had won during the championship", the forget gate for subject-related dimensions stays near 1, preserving the "trophy" information despite the intervening

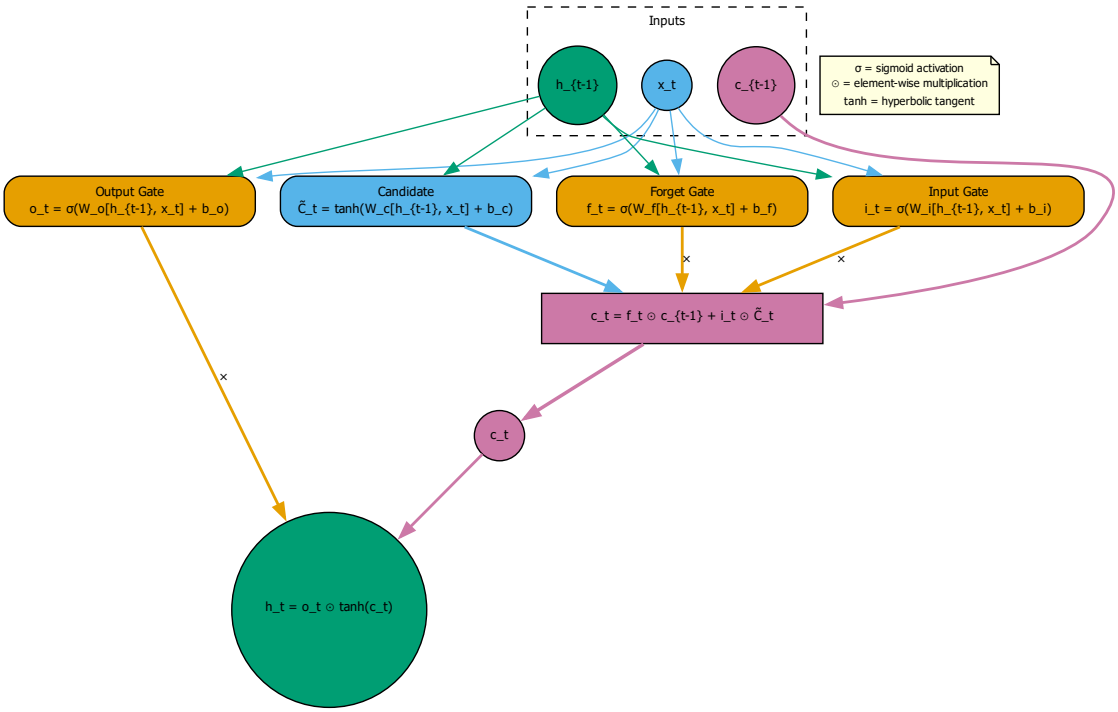


Figure 5.10: Complete LSTM cell architecture. The cell receives input embedding $\mathbf{e}_t$ and previous states $\mathbf{h}_{t-1}$, $\mathbf{c}_{t-1}$. Four parallel computations produce the forget gate $\mathbf{f}_t$, input gate $\mathbf{i}_t$, candidate $\tilde{\mathbf{c}}_t$, and output gate $\mathbf{o}_t$. The cell state update combines forgotten old content ($\mathbf{f}_t \odot \mathbf{c}_{t-1}$) with gated new content ($\mathbf{i}_t \odot \tilde{\mathbf{c}}_t$). The hidden state is the gated cell output ($\mathbf{o}_t \odot \tanh(\mathbf{c}_t)$). Element-wise multiplications ($\odot$) are shown as circles; the addition ($+$) for cell state update is the key to gradient preservation.

material. Simultaneously, the input gate selectively adds clause-internal information (like “athletes”, “team”) to different cell dimensions. The output gate modulates what information influences word predictions within the clause. By position 17 (“it”), the cell state still contains the subject information from position 2 because the forget gate protected it. When predicting position 18, this preserved information enables correct prediction of “was” (singular) rather than “were” (plural). The key is that the gates learn to identify what information is relevant for long-range dependencies and protect it appropriately—a capability that emerges through training on examples requiring such dependencies.

5.3.4 Why Gates Help: Gradient Highways

The gating mechanism provides more than intuitive information control—it fundamentally changes gradient dynamics during training, enabling effective learning of long-range dependencies that would be impossible for vanilla RNNs. Consider the gradient $\frac{\partial \mathbf{c}_t}{\partial \mathbf{c}_{t-1}}$ from the cell state update equation, which determines how error signals propagate backward through the cell state pathway during backpropagation through time. By the chain rule, this gradient depends on how the current cell state changes with respect to the previous cell state: $\frac{\partial \mathbf{c}_t}{\partial \mathbf{c}_{t-1}} = \text{diag}(\mathbf{f}_t) + (\text{terms involving gate derivatives})$, where $\text{diag}(\mathbf{f}_t)$ is a diagonal matrix with the forget gate values on the diagonal. When the forget gate $\mathbf{f}_t \approx 1$, this Jacobian is close to the identity matrix, meaning gradients flowing backward through the cell state multiply by matrices close to identity and thereby preserve their magnitude across time steps. This behavior stands in stark contrast to vanilla RNNs, where the Jacobian $\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}}$ involves $\text{diag}(\tanh'(\cdot))\mathbf{W}_{hh}$, with singular values that are typically far from 1 and cause exponential growth or decay of gradients. The cell state thus acts as a gradient highway: when the forget gate is open (near 1), gradients pass through with minimal attenuation, enabling learning signals to reach early time steps with

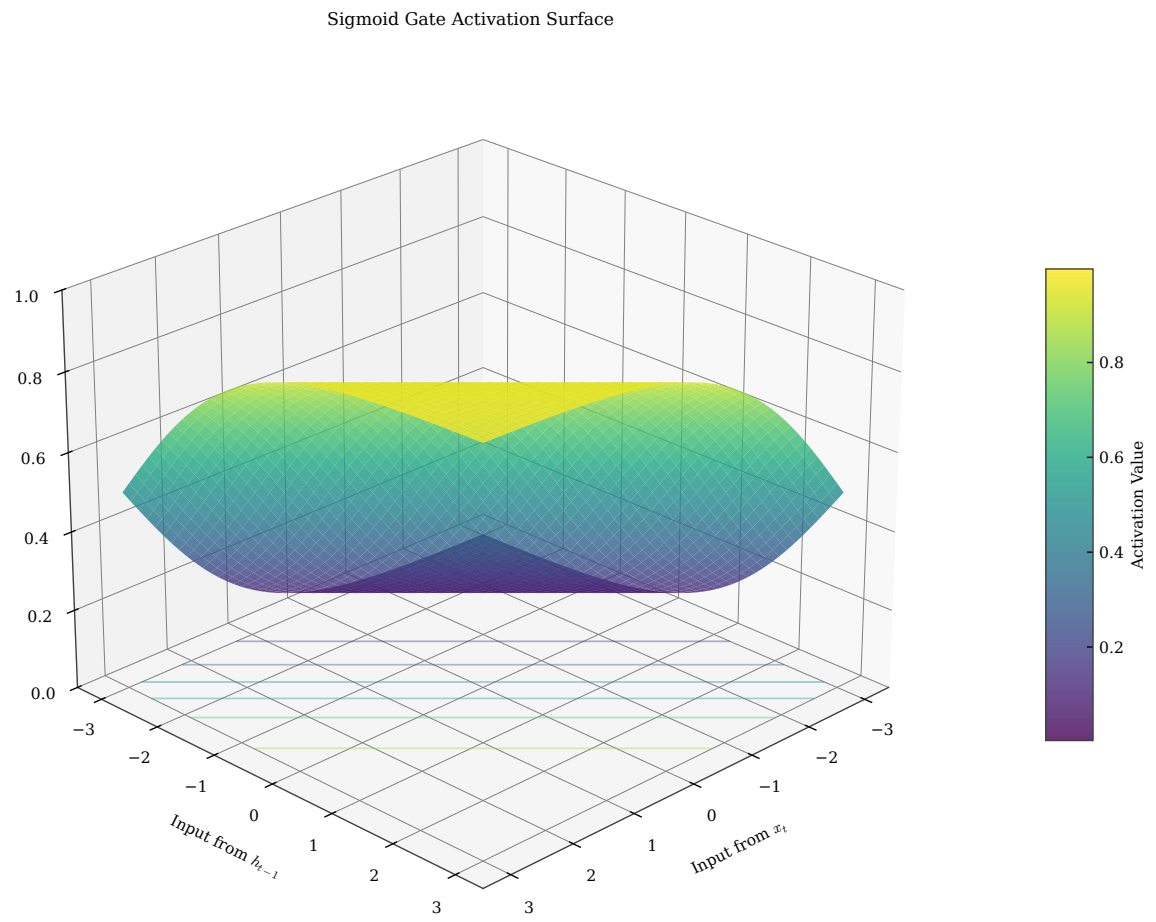


Figure 5.11: Gate activation surface in 3D. The sigmoid activation function produces gate values between 0 and 1 based on weighted inputs from $\mathbf{e}_t$ and $\mathbf{h}_{t-1}$. The surface shows how gate activation varies smoothly across the input space: regions with high weighted sum (upper right) produce gate values near 1 (information passes), while regions with low weighted sum (lower left) produce values near 0 (information blocked). The learned weights determine where in input space each gate opens or closes, enabling context-dependent memory control. The transition region (middle) allows for partial gating.

sufficient magnitude to drive parameter updates. The model can learn to keep forget gates high precisely when long-range gradient flow is needed for the task at hand, enabling effective credit assignment across many time steps without explicit supervision about what to remember. This self-regulating property—where the same gates that control information flow during the forward pass also control gradient flow during the backward pass—is the key to LSTM's trainability and the reason it can learn dependencies spanning hundreds of time steps, far beyond the five-to-ten step limit of vanilla RNNs. For our running example, gradients from the loss at position 18 can reach position 2 with sufficient magnitude to update the parameters responsible for preserving subject information, enabling the model to learn the subject-verb agreement pattern through standard gradient descent.

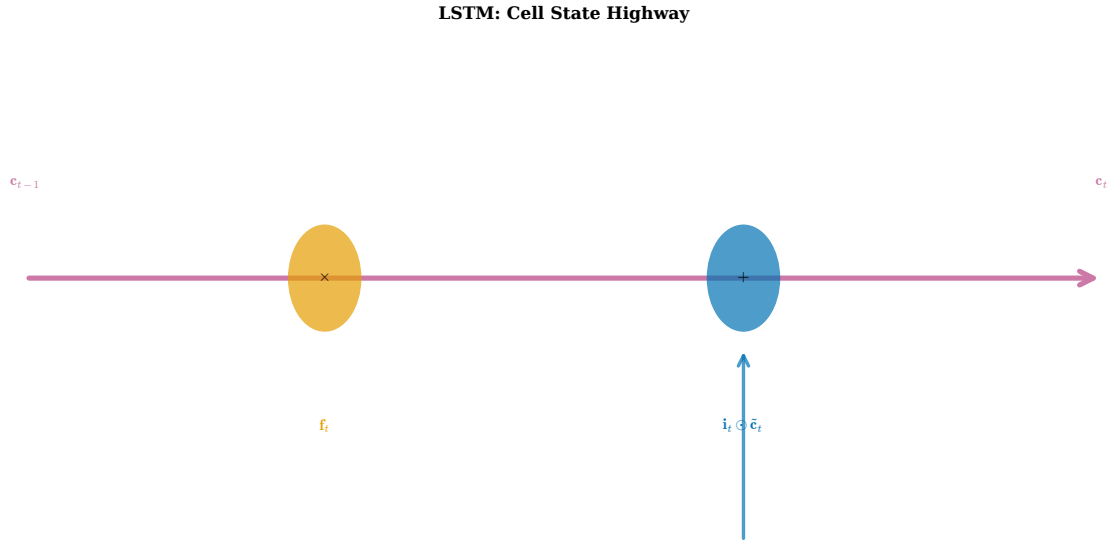


Figure 5.12: Cell state as memory highway. The top diagram shows the LSTM cell state pathway: information can flow unchanged when the forget gate is near 1 and input gate is near 0, creating a "highway" for information preservation. The additive update $\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t$ means cell content persists without repeated matrix multiplication. The bottom diagram contrasts with vanilla RNN, where hidden state updates involve multiplicative transformation through the weight matrix at every step, causing information to be transformed (and potentially lost) continuously. This architectural difference explains LSTM's superior long-range memory.

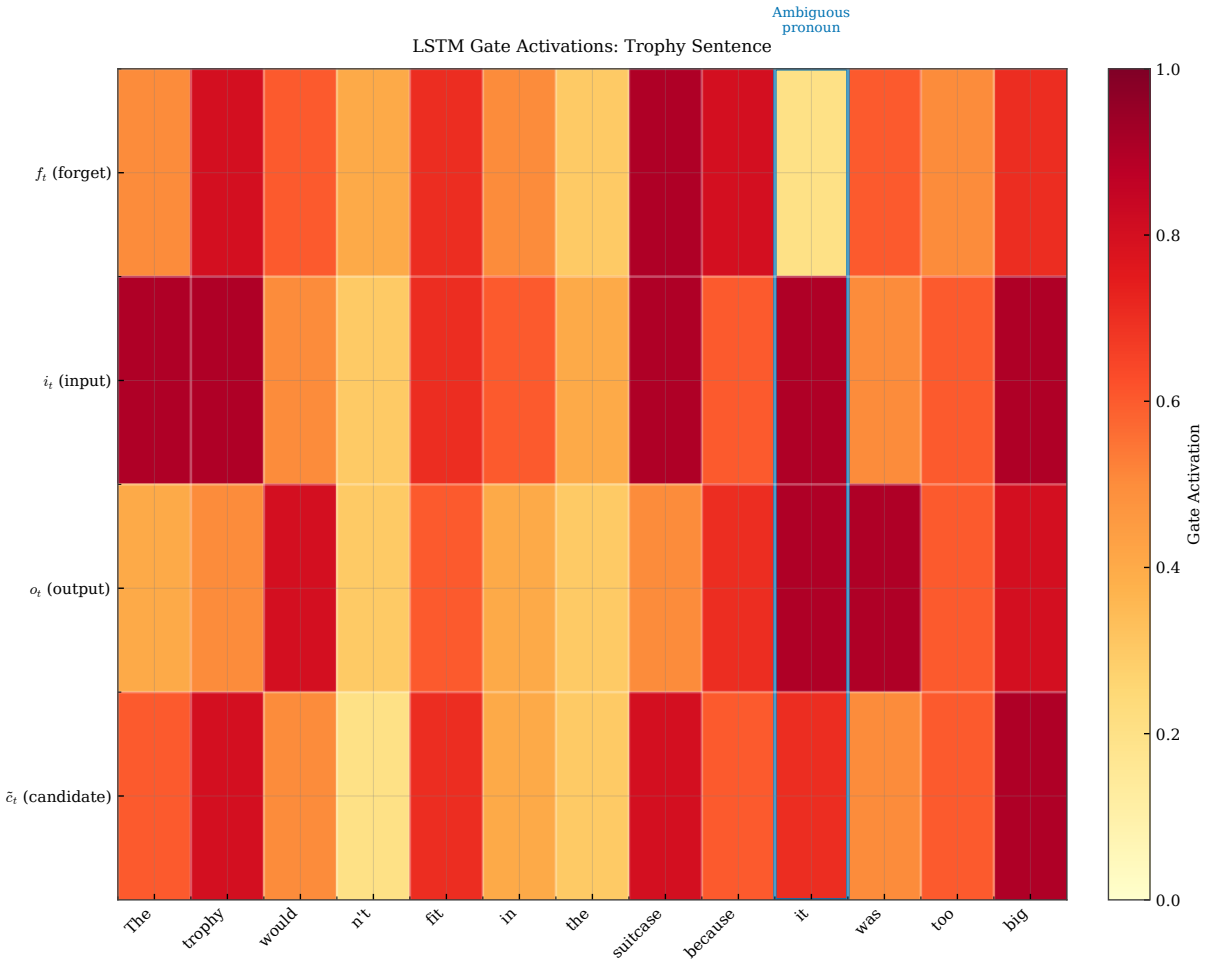


Figure 5.13: Gate activations for the running example in LSTM. The heatmap shows gate values (0 = dark, 1 = light) across sequence positions for a trained LSTM. The forget gate row shows consistently high values, indicating information preservation through the relative clause. The input gate row shows high activation at content words (“trophy”, “athletes”, “won”) and low activation at function words. The output gate row shows higher activation before major predictions. The cell state row shows accumulated information growing through the sequence. At position 18 (predicting “was”), the preserved “trophy” information enables correct singular verb prediction.

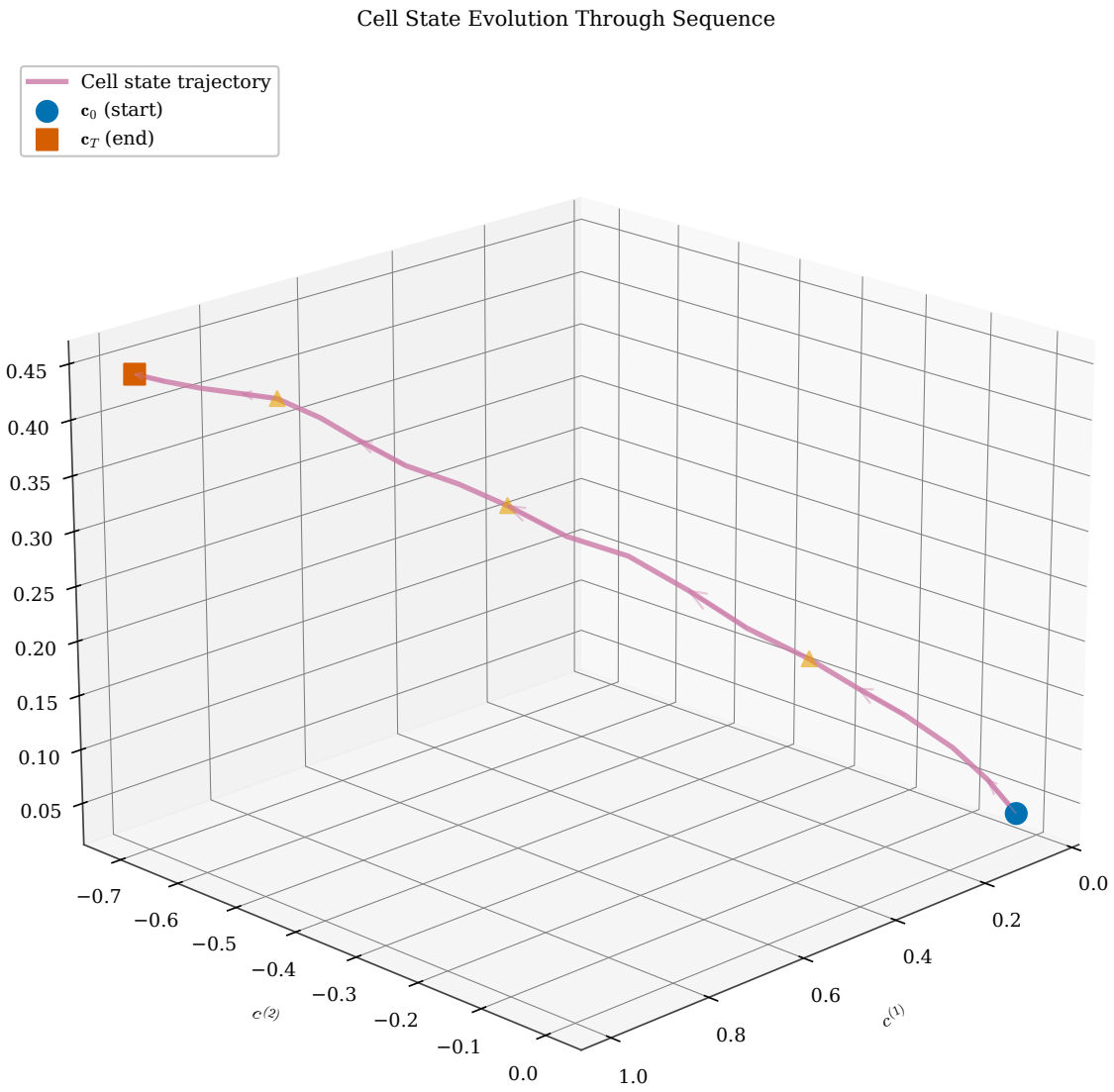


Figure 5.14: Cell state trajectory in 3D space. Unlike the hidden state which transforms at every step, the cell state (shown in pink) follows a smoother, more stable trajectory through sequence processing. The cell state changes primarily at positions where the input gate opens to add new information, remaining relatively constant through intervening material. This stability enables information preservation across long sequences. Compare with Figure 5.7: the cell state path shows less curvature and more consistent direction, reflecting its role as long-term memory.

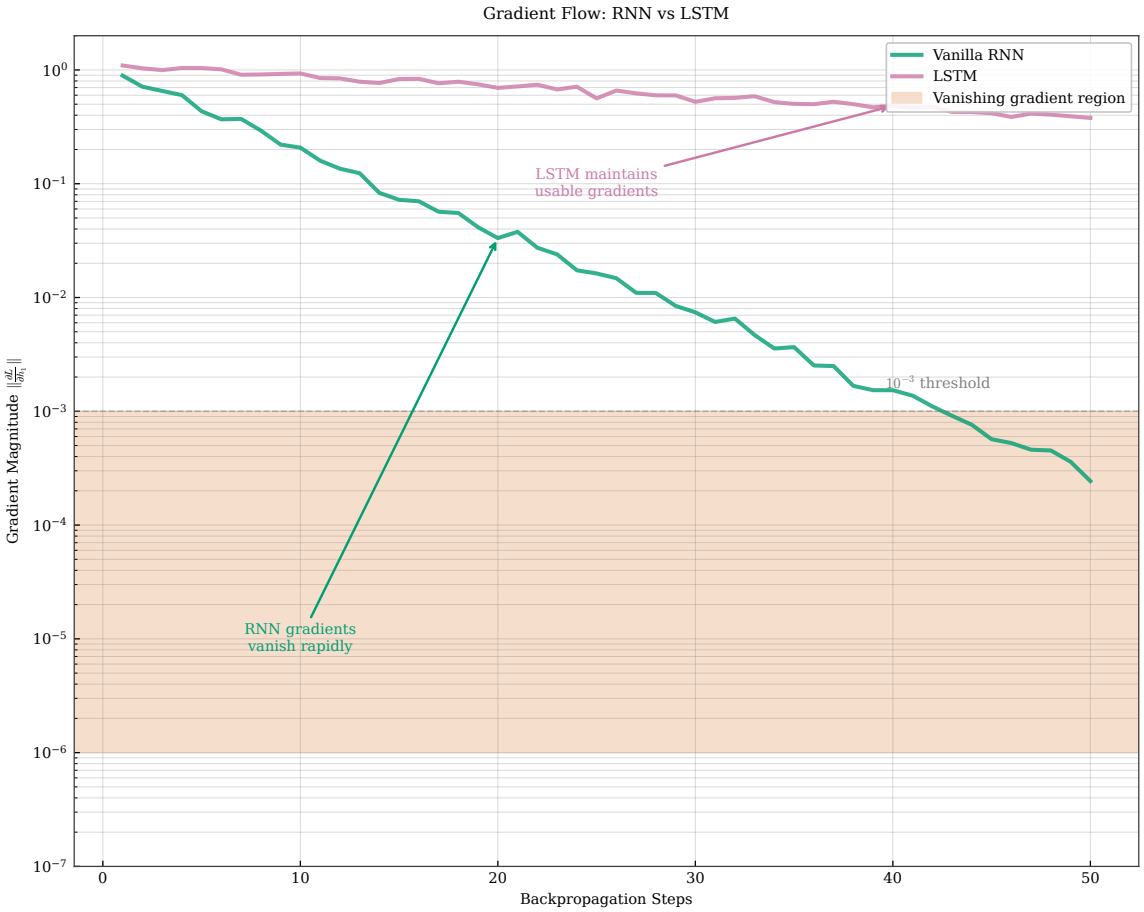


Figure 5.15: Gradient magnitude comparison: vanilla RNN versus LSTM. Both curves show gradient magnitude (log scale) as a function of temporal distance during backpropagation. The vanilla RNN (red) shows exponential decay, with gradients becoming negligible after 10-15 steps. The LSTM (green) maintains gradient magnitude across the full sequence, with the curve staying above the effective training threshold (dashed line) even at 50 steps. The shaded region indicates where vanilla RNN fails to learn due to vanishing gradients. LSTM's gradient highway enables learning dependencies of arbitrary length.

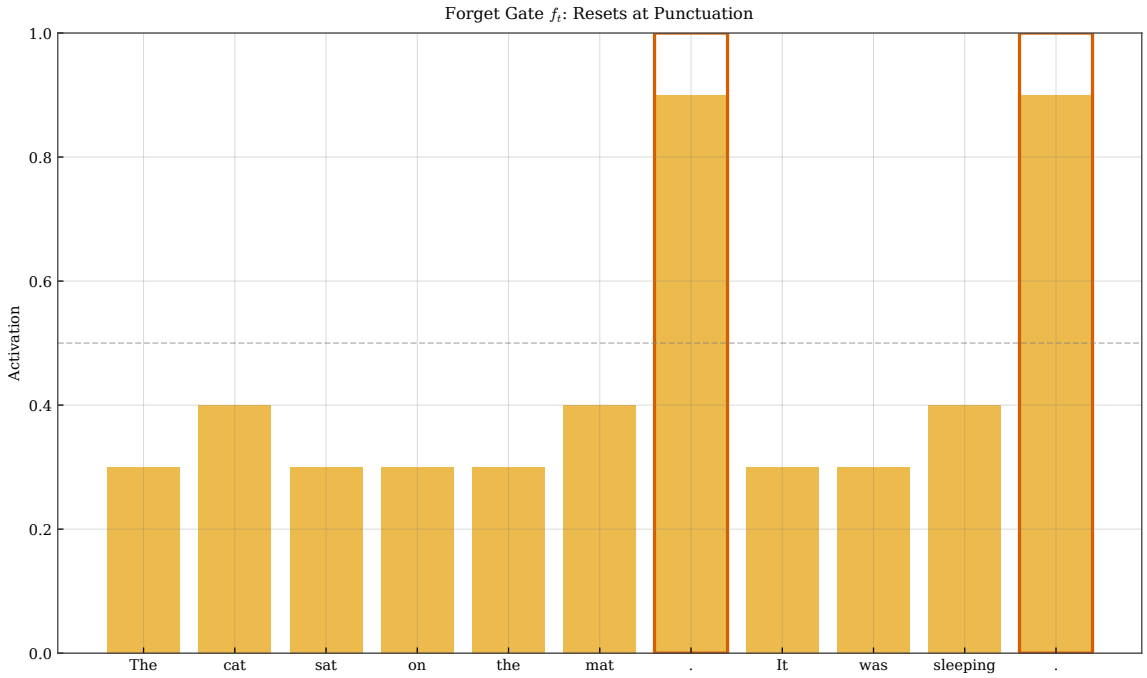


Figure 5.16: Learned gate patterns for different linguistic phenomena. Analysis of trained LSTM gates reveals interpretable patterns. The forget gate shows high activation at sentence boundaries and clause boundaries, resetting context for new units. The input gate shows high activation for content words (nouns, verbs) and low activation for function words (the, of, a). The output gate shows high activation before positions requiring prediction (verbs, objects) and lower activation elsewhere. These patterns emerge from training without explicit linguistic supervision, demonstrating that LSTMs learn linguistically meaningful representations.

5.4 GRU and Architecture Comparisons

While LSTM remains the most widely used gated recurrent architecture, the Gated Recurrent Unit (GRU), introduced by Cho et al. (2014) [Cho et al., 2014], offers a simpler alternative with comparable performance on many tasks. The GRU architecture combines the forget and input gates into a single update gate and merges the cell state and hidden state into a unified representation, reducing the number of parameters and computations per time step while retaining the essential gating mechanism that enables long-range memory. This simplification emerged from empirical observations that the full LSTM architecture, while powerful, may be over-parameterized for certain tasks, and that a more parsimonious design could achieve similar performance with reduced computational cost. This section examines the GRU architecture in detail and compares the three recurrent architectures—vanilla RNN, GRU, and LSTM—in terms of parameter efficiency, computational cost, and performance on language modeling tasks. The choice between GRU and LSTM is often empirical, depending on the specific task, dataset size, and computational constraints, though LSTM's explicit separation of cell state and hidden state can provide advantages for very long sequences where fine-grained memory control is beneficial. Both gated architectures represent major improvements over vanilla RNNs for capturing long-range dependencies, and understanding their similarities and differences informs architecture selection for practical applications in language modeling and other sequential tasks.

5.4.1 GRU Architecture

The GRU simplifies LSTM by using two gates instead of three and eliminating the separate cell state, resulting in a more compact architecture that is often easier to train and deploy. The update gate $\mathbf{z}_t = \sigma(\mathbf{W}_z[\mathbf{h}_{t-1}; \mathbf{e}_t] + \mathbf{b}_z)$ controls how much of the previous hidden state to retain versus how much to update with new content, effectively combining the functions of LSTM's forget and input gates into a single mechanism. The reset gate $\mathbf{r}_t = \sigma(\mathbf{W}_r[\mathbf{h}_{t-1}; \mathbf{e}_t] + \mathbf{b}_r)$ controls how much of the previous hidden state influences the candidate update, allowing the model to selectively ignore past information when computing new content. The candidate hidden state is computed as $\tilde{\mathbf{h}}_t = \tanh(\mathbf{W}_h[\mathbf{r}_t \odot \mathbf{h}_{t-1}; \mathbf{e}_t] + \mathbf{b}_h)$, where the reset gate modulates the contribution of the previous hidden state before concatenation with the current input. Finally, the hidden state update $\mathbf{h}_t = (1 - \mathbf{z}_t) \odot \mathbf{h}_{t-1} + \mathbf{z}_t \odot \tilde{\mathbf{h}}_t$ interpolates between keeping the old hidden state (when $\mathbf{z}_t \approx 0$) and fully adopting the candidate (when $\mathbf{z}_t \approx 1$). This interpolation mechanism provides the same gradient highway property as LSTM's forget gate: when $\mathbf{z}_t \approx 0$, the hidden state passes through unchanged, allowing gradients to flow backward without attenuation. The reset gate enables the model to "start fresh" when encountering a new context by setting $\mathbf{r}_t \approx 0$, which causes the candidate computation to ignore the previous hidden state. The GRU has approximately 25% fewer parameters than an LSTM with the same hidden dimension, since it computes three transformations (update, reset, candidate) versus LSTM's four (forget, input, candidate, output), and this efficiency advantage translates to faster training and inference times.

5.4.2 LSTM vs GRU Trade-offs

The choice between LSTM and GRU involves trade-offs along several dimensions, and understanding these trade-offs enables informed architecture selection for specific applications. In terms of parameters, GRU requires $3 \times (d_h \times (d_h + d) + d_h)$ parameters for its gates and candidate, while LSTM requires $4 \times (d_h \times (d_h + d) + d_h)$ for its four transformations. For typical values $d_h = 256$ and $d = 128$, LSTM has approximately 787K parameters versus GRU's 590K—a 25% reduction that can be significant for resource-constrained deployment. This parameter efficiency means GRU is less prone to overfitting on smaller datasets and faster to train, making it attractive when training data is limited or computational resources are constrained. From a computational perspective, GRU performs fewer matrix multiplications per time step, yielding 20-30% faster training and inference times while maintaining comparable accuracy on most benchmarks. However, LSTM's separate cell state provides a more direct gradient pathway through the additive cell state updates, which can be advantageous for very long sequences spanning hundreds of steps where gradient preservation is critical. The explicit output gate in LSTM also provides finer control over what information influences predictions versus what persists in memory, potentially enabling more nuanced memory management in complex tasks. Empirically, perfor-

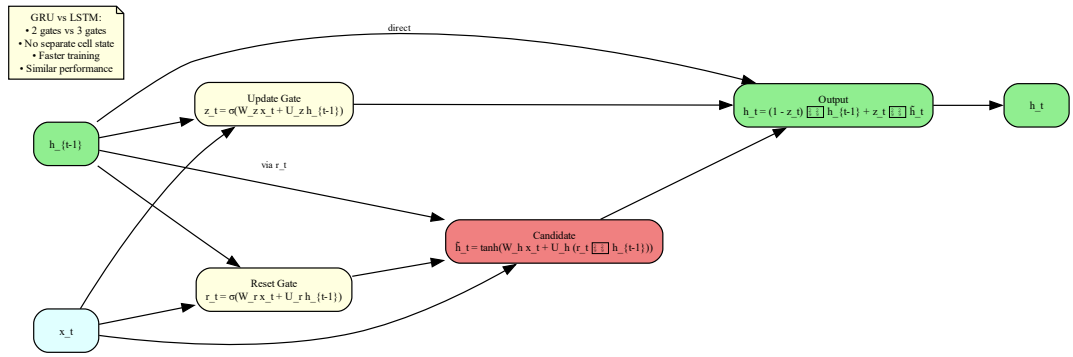


Figure 5.17: GRU cell architecture compared with LSTM. The GRU (left) has two gates: the update gate $\mathbf{z}_t$ that combines LSTM's forget and input functions, and the reset gate $\mathbf{r}_t$ that controls candidate computation. There is no separate cell state—the hidden state $\mathbf{h}_t$ serves both as output and as memory. The LSTM (right, simplified) maintains separate cell state $\mathbf{c}_t$ and hidden state $\mathbf{h}_t$ with three gates. The annotation highlights that GRU uses two gates versus LSTM's three, resulting in 25% fewer parameters for the same hidden dimension.

mance differences between LSTM and GRU are often within statistical noise on standard language modeling benchmarks such as Penn Treebank and WikiText. Both architectures achieve similar perplexity when hyperparameters are properly tuned, and the choice is often made based on computational budget (favoring GRU) or prior experience with the architecture. Some practitioners prefer LSTM for its interpretability: the separate cell state and explicit gates provide clearer semantics for understanding what the model has learned, facilitating analysis of learned representations. For our running example sentence, both LSTM and GRU successfully predict "was" by preserving subject information across the relative clause, though they accomplish this through slightly different gating mechanisms.

5.4.3 When to Use Which

Choosing among vanilla RNN, GRU, and LSTM depends on task requirements and constraints. Vanilla RNN should generally be avoided for language modeling due to its inability to capture dependencies beyond a few time steps; it may still be appropriate for tasks with very short sequences (fewer than 10 tokens) where computational efficiency is paramount. GRU is preferred when computational resources are limited, training data is modest (reducing overfitting risk with fewer parameters), or when the maximum dependency length is moderate (fewer than 50 tokens). LSTM is preferred when very long dependencies are expected (hundreds of tokens), when the separate cell state semantics aid interpretability, or when following established practices in a domain where LSTM has been extensively validated. In practice, the best approach is often to try both GRU and LSTM with proper hyperparameter tuning, as the optimal choice is dataset-dependent. Modern sequence models like Transformers (Chapter **??**) have largely superseded both for tasks where computational resources allow, but LSTM and GRU remain important for edge deployment, real-time applications, and understanding the historical development of neural language models.

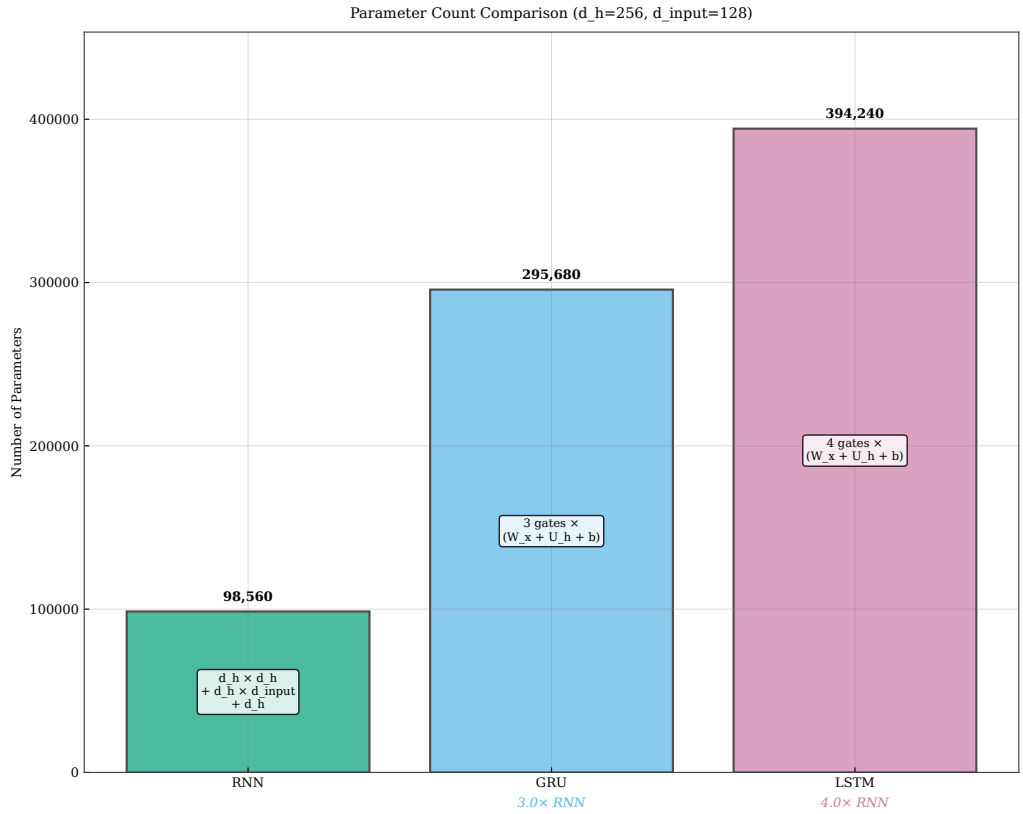


Figure 5.18: Parameter count comparison for recurrent architectures. For hidden dimension $d_h = 256$ and input dimension $d = 128$, the bar chart shows total parameters excluding the embedding and output layers. Vanilla RNN has fewest parameters (131K), GRU has moderate (590K), and LSTM has most (787K). The percentages indicate parameter count relative to LSTM: vanilla RNN is 17%, GRU is 75%. Despite having fewer parameters, GRU and LSTM dramatically outperform vanilla RNN on tasks requiring long-range memory, demonstrating that architectural innovations matter more than raw parameter count for sequence modeling.

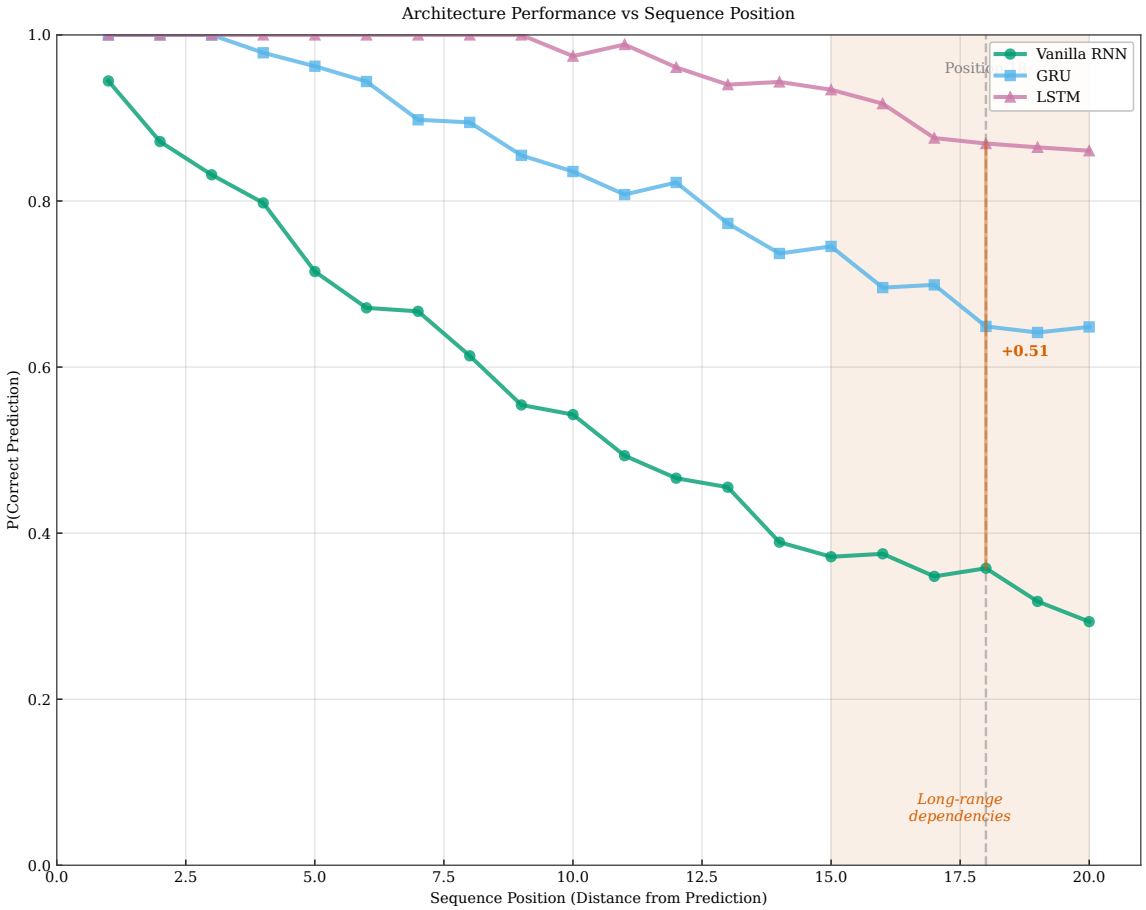


Figure 5.19: Performance comparison on running example. The plot shows prediction confidence $P(\text{was} \mid \cdot)$ at each position for vanilla RNN (red), GRU (cyan), and LSTM (orange). All architectures perform similarly for short-range predictions (positions 1-5). As sequence length increases, vanilla RNN performance degrades rapidly, producing near-random predictions (50%) by position 18. Both GRU and LSTM maintain high confidence in the correct prediction "was" throughout the sequence, with LSTM showing marginally higher confidence. The vertical dashed line at position 18 highlights the critical prediction where long-range dependency matters.

5.5 Training RNNs for Language Modeling

Training recurrent networks for language modeling requires adapting the standard supervised learning framework to the unique characteristics of sequential data, where each prediction depends on an evolving hidden state that encodes context from all preceding words. The training objective is next-word prediction: given a sequence of words $w_1, \ldots, w_T$, the model should maximize $\prod_{t=1}^{T} P(w_t \mid w_1, \ldots, w_{t-1})$, or equivalently minimize the negative log-likelihood (cross-entropy loss). This objective aligns perfectly with the language modeling task: at each position, the model receives a training signal indicating how well it predicted the actual next word, and this signal drives parameter updates that improve future predictions. Gradient computation requires backpropagation through time (BPTT), which unrolls the recurrence relation and computes gradients through the entire computational graph spanning multiple time steps. For long sequences containing hundreds or thousands of tokens, full BPTT becomes computationally expensive and memory-intensive, motivating truncated BPTT that limits the backward pass to a fixed window while maintaining the forward context. This section examines the training objective in detail, the mechanics of BPTT gradient computation, and practical considerations including gradient clipping, learning rate scheduling, and efficient sequence batching. Understanding these training details is essential for successfully applying recurrent networks to language modeling tasks and for interpreting the behavior and limitations of trained models in practice.

5.5.1 The Language Modeling Objective

The language modeling objective directly aligns with next-word prediction, making it a natural fit for training recurrent architectures. Given a corpus $\mathscr{D} = [w_1, \ldots, w_T]$, we seek to maximize the joint probability $P(w_1, \ldots, w_T)$, which by the chain rule of probability factorizes as $\prod_{t=1}^{T} P(w_t \mid w_1, \ldots, w_{t-1})$. In practice, we minimize the average negative log-likelihood, also known as cross-entropy loss: $\mathscr{L} = -\frac{1}{T} \sum_{t=1}^{T} \log P(w_t \mid \mathbf{h}_{t-1})$, where $\mathbf{h}_{t-1}$ is the hidden state encoding the context $w_1, \ldots, w_{t-1}$, and the probability is computed through the output layer as $P(w_t \mid \mathbf{h}_{t-1}) = \text{softmax}(\mathbf{W}_{hy}\mathbf{h}_{t-1} + \mathbf{b}_y)[w_t]$. This loss measures the cross-entropy between the model's predicted distribution over vocabulary and the one-hot target distribution indicating the actual next word. The perplexity, a more interpretable metric commonly reported in language modeling research, is the exponentiated average loss: $\text{PPL} = \exp(\mathscr{L}) = \exp\left(-\frac{1}{T} \sum_{t=1}^{T} \log P(w_t \mid \mathbf{h}_{t-1})\right)$. Perplexity can be interpreted as the effective number of equally likely words the model considers at each position: a perplexity of 100 means the model's uncertainty is equivalent to choosing uniformly among 100 words, while lower perplexity indicates more confident and accurate predictions. State-of-the-art LSTM language models achieve perplexities of 50-70 on standard benchmarks like Penn Treebank, representing substantial improvements over n-gram baselines. The training procedure employs stochastic gradient descent (SGD) or variants like Adam, computing gradients of the loss with respect to all parameters and updating parameters in the direction that decreases loss, with gradient computation requiring the BPTT algorithm we examine next.

5.5.2 Backpropagation Through Time

Backpropagation through time (BPTT) applies the chain rule to compute gradients through the unrolled recurrent network, treating the unrolled sequence as a deep feedforward network with shared weights. Consider the gradient of the loss at position t with respect to the hidden-to-hidden weights $\mathbf{W}_{hh}$, which are used at every time step. This gradient depends on how $\mathbf{W}_{hh}$ affects $\mathbf{h}_t$, which depends on $\mathbf{h}_{t-1}$, which depends on $\mathbf{h}_{t-2}$, and so on back to $\mathbf{h}_1$—creating a chain of dependencies spanning the entire sequence. The full gradient accumulates contributions from all time steps: $\frac{\partial \mathscr{L}}{\partial \mathbf{W}_{hh}} = \sum_{t=1}^{T} \sum_{k=1}^{t} \frac{\partial \mathscr{L}_t}{\partial \mathbf{h}_t} \left(\prod_{j=k+1}^{t} \frac{\partial \mathbf{h}_j}{\partial \mathbf{h}_{j-1}} \right) \frac{\partial \mathbf{h}_k}{\partial \mathbf{W}_{hh}}$, where the product term $\prod_{j=k+1}^{t} \frac{\partial \mathbf{h}_j}{\partial \mathbf{h}_{j-1}}$ propagates gradients backward through time from position t to position k. For vanilla RNNs, this product causes vanishing or exploding gradients because it involves repeated multiplication by the Jacobian matrix; for LSTMs, the cell state provides an alternative pathway where the products are closer to identity matrices. The memory cost of BPTT is $O(T \cdot d_h)$ to store all intermediate hidden states and activations needed for the backward pass. For very long sequences with $T > 1000$ tokens, this memory requirement becomes prohibitive, often exceeding available GPU memory. Computation time is $O(T)$ for both forward and backward

passes, with the constant factor including matrix multiplications at each step. The strictly sequential nature of RNNs means that neither forward nor backward computation can be parallelized across time steps, limiting training throughput compared to architectures like Transformers that process all positions in parallel.

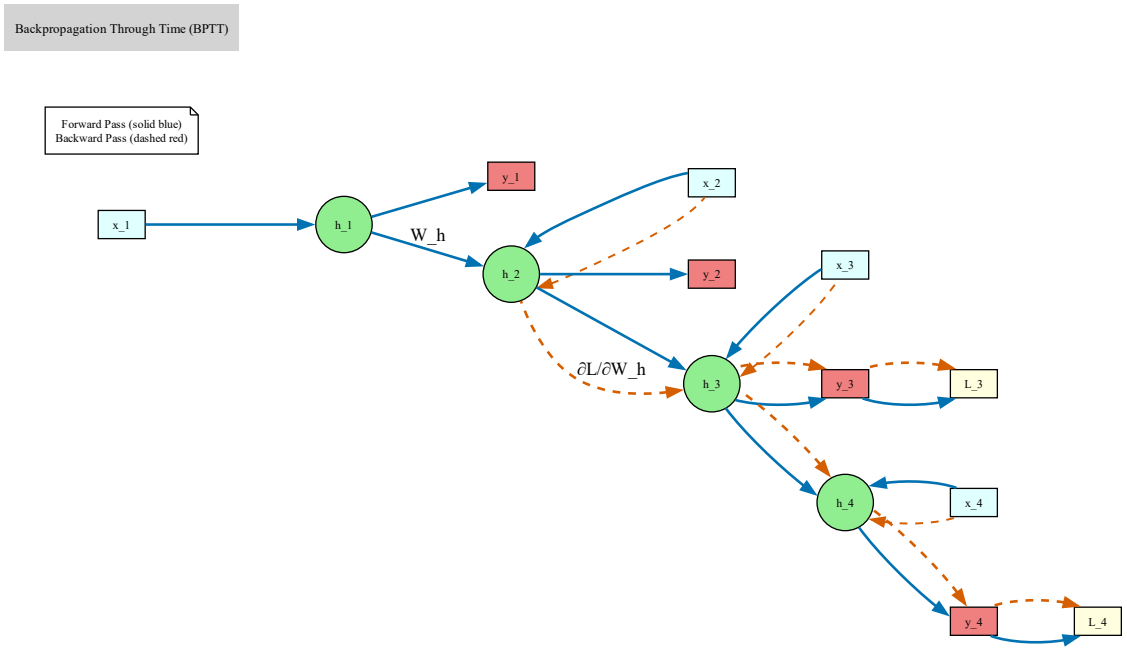


Figure 5.20: Backpropagation through time visualization. The diagram shows the unrolled RNN with forward pass (blue arrows) and backward pass (red arrows). During the forward pass, hidden states are computed left-to-right and stored in memory. During the backward pass, gradients flow right-to-left through the stored hidden states. At each position, the loss gradient $\frac{\partial \mathscr{L}_t}{\partial \mathbf{h}_t}$ enters from the output layer, then propagates backward through the chain of hidden states. Gradients accumulate at shared parameters (weights, biases) across all time steps, requiring summation over the entire sequence.

5.5.3 Practical Training Considerations

Several practical techniques are essential for successfully training recurrent language models. Truncated BPTT limits the backward pass to K steps rather than the full sequence, reducing memory from $O(T)$ to $O(K)$ and enabling training on arbitrarily long sequences. The forward pass still processes the entire sequence to build correct hidden states, but gradients are only computed for the most recent K steps. Typical values are $K \in [35, 100]$; smaller values risk missing long-range dependencies, while larger values increase memory and computation. Hidden states are carried forward across truncation boundaries, so the model still sees full context during forward passes. Gradient clipping prevents exploding gradients by scaling the gradient vector when its norm exceeds a threshold: if $\|\nabla \mathscr{L}\| > \tau$, set $\nabla \mathscr{L} \leftarrow \tau \cdot \nabla \mathscr{L} / \|\nabla \mathscr{L}\|$. Typical thresholds are $\tau \in [1, 5]$. This ensures stable training without limiting the model's ability to learn long-range dependencies (which require gradients to be preserved, not prevented from growing). Learning rate scheduling is critical: starting with a high learning rate enables fast initial progress, then decaying the rate allows fine-tuning. Common schedules include step decay (reduce by factor after fixed epochs) and cosine annealing. Regularization through dropout (applied to non-recurrent connections) and weight decay prevents overfitting on smaller corpora.

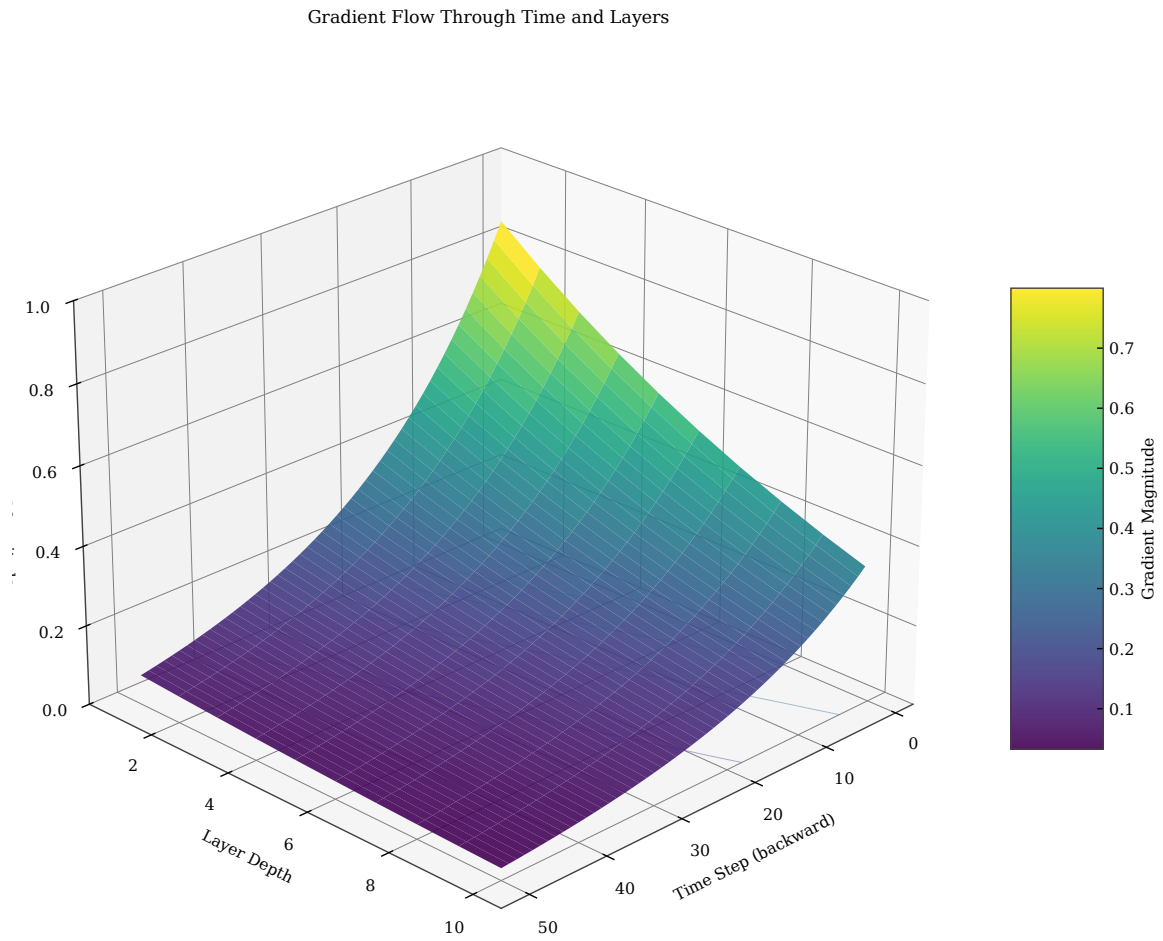


Figure 5.21: Gradient magnitude surface over time and layer depth. For a stacked recurrent network, the 3D surface shows how gradient magnitude varies across time steps (x-axis) and layers (y-axis). Gradients are strongest near the output (recent time, top layer) and weaken toward the input (early time, bottom layer). The surface for vanilla RNN (not shown) would decay much more steeply. This LSTM gradient surface shows that significant gradient signal reaches early time steps and bottom layers, enabling learning of long-range dependencies across the full network depth.

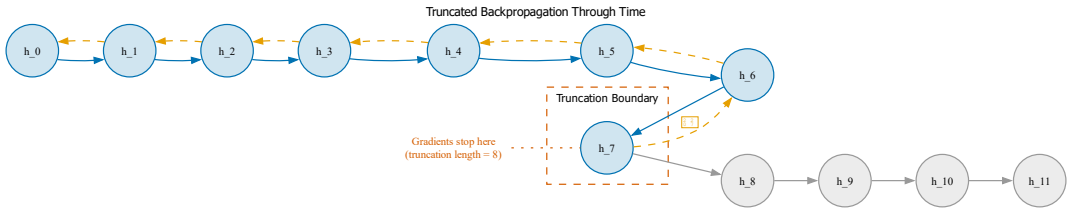


Figure 5.22: Truncated backpropagation through time. For a long sequence, the forward pass (blue) processes all time steps, accumulating hidden state information. The backward pass (red) is truncated to a window of K steps: gradients are computed within this window, then detached. The next forward segment starts from the hidden state at the truncation boundary, preserving context continuity. This approach trades off gradient fidelity (missing contributions from beyond K steps) for computational efficiency. The dashed vertical line shows where gradients stop, with the annotation indicating the trade-off between efficiency and long-range learning.

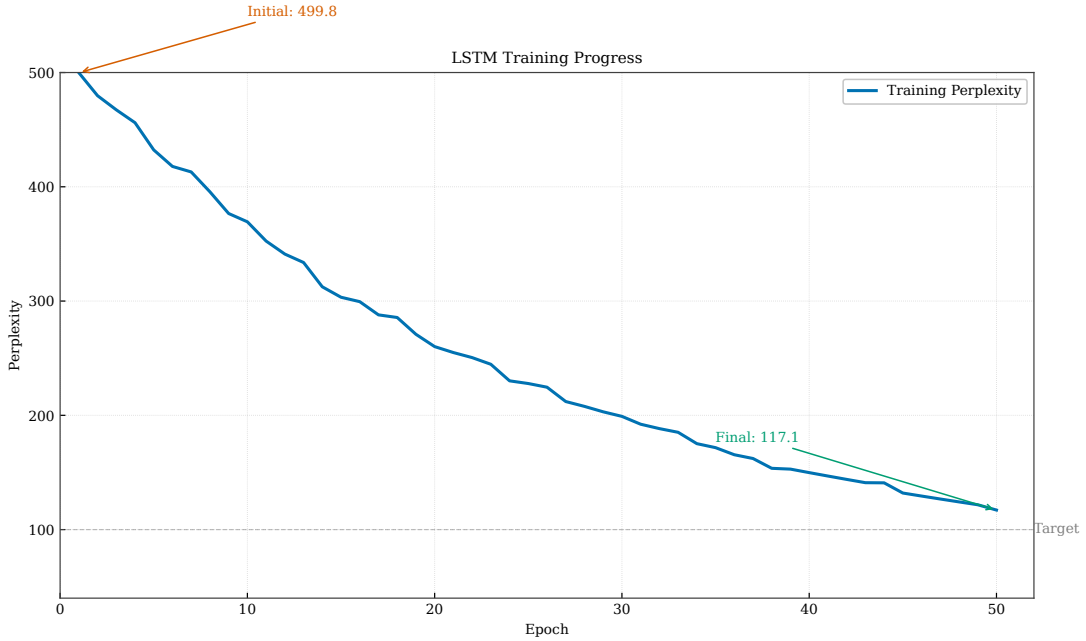


Figure 5.23: Training loss curve for LSTM language model. Perplexity on the training set (blue) decreases from approximately 450 (near random for a 50K vocabulary) to below 70 over 50 epochs. The curve shows rapid initial improvement as the model learns basic word frequencies and short-range patterns, then slower improvement as it captures longer-range dependencies. The validation curve (orange) typically runs slightly higher, with the gap indicating overfitting degree. Training is stopped when validation perplexity stops improving (early stopping) to prevent overfitting.

5.6 Context Representation in RNNs

Having developed recurrent architectures for sequential processing, we now examine how RNNs represent context compared to previous approaches. Each chapter in this textbook advances context representation: n-grams use discrete tuples, embeddings use static vectors, and RNNs use dynamic hidden states. This section synthesizes these progressions and identifies the limitations that motivate attention mechanisms in Chapter **??**. The hidden state $\mathbf{h}_t$ serves as the context representation in RNNs—a learned, fixed-size summary of the entire sequence history $w_1, \ldots, w_t$. Unlike n-gram contexts that are limited to fixed windows, the hidden state can theoretically encode information from arbitrarily far back in the sequence. Unlike static embeddings that ignore context entirely, the hidden state adapts based on the specific sequence of words that preceded the current position. However, the hidden state representation has fundamental limitations: it must compress variable-length history into a fixed-size vector, creating an information bottleneck for very long sequences. Additionally, the sequential processing means that information must pass through many intermediate states to travel from early positions to later ones, even when a direct connection would be more appropriate. These limitations point toward the attention mechanism, which allows the model to directly access any position in the input sequence rather than relying on a single compressed representation.

How This Chapter Represents Context:

- **Context representation:** The hidden state $\mathbf{h}_t \in \mathbb{R}^{d_h}$ encodes variable-length history in a fixed-size vector
- **Context encoding:** Sequential processing builds context incrementally through learned recurrence relations
- **Key advance:** Dynamic representations that evolve with each word, capturing long-range dependencies through gating
- **Limitation:** Compression bottleneck forces all history through a fixed-size vector; Chapter **??** introduces attention for direct access to all positions

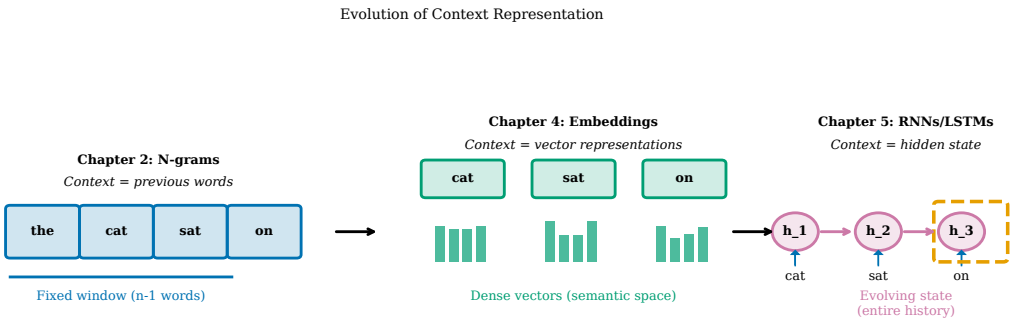


Figure 5.24: Evolution of context representation across chapters. Chapter 2 (n-grams) represents context as a discrete tuple of previous words, limited to a fixed window and unable to generalize across similar contexts. Chapter 4 (embeddings) represents each word as a continuous vector but ignores sequential context entirely. Chapter 5 (RNNs) represents context as a dynamic hidden state that evolves through the sequence, encoding variable-length history in a fixed-size vector. Each representation advances the expressiveness and generalization capability for next-word prediction.

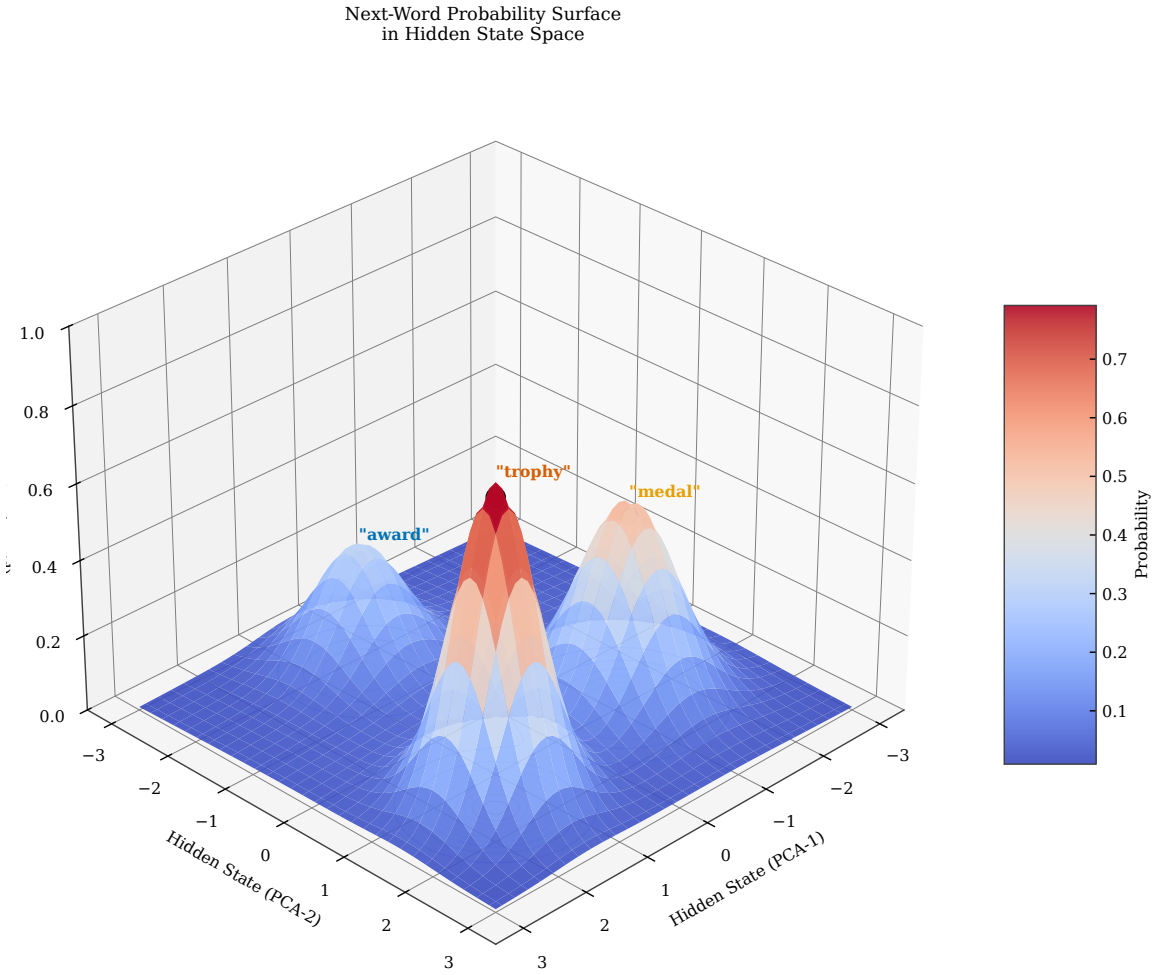


Figure 5.25: Prediction probability surface over hidden state space. The 3D surface shows how $P(w_{t+1} \mid \mathbf{h}_t)$ varies across a 2D projection (via PCA) of the hidden state space. Peaks correspond to hidden state regions that strongly predict specific next words (labeled). The surface topology reveals that the hidden state encodes predictive structure: similar hidden states lead to similar prediction distributions, and distinct semantic contexts occupy distinct regions. This visualization demonstrates that the hidden state successfully organizes contextual information for next-word prediction.

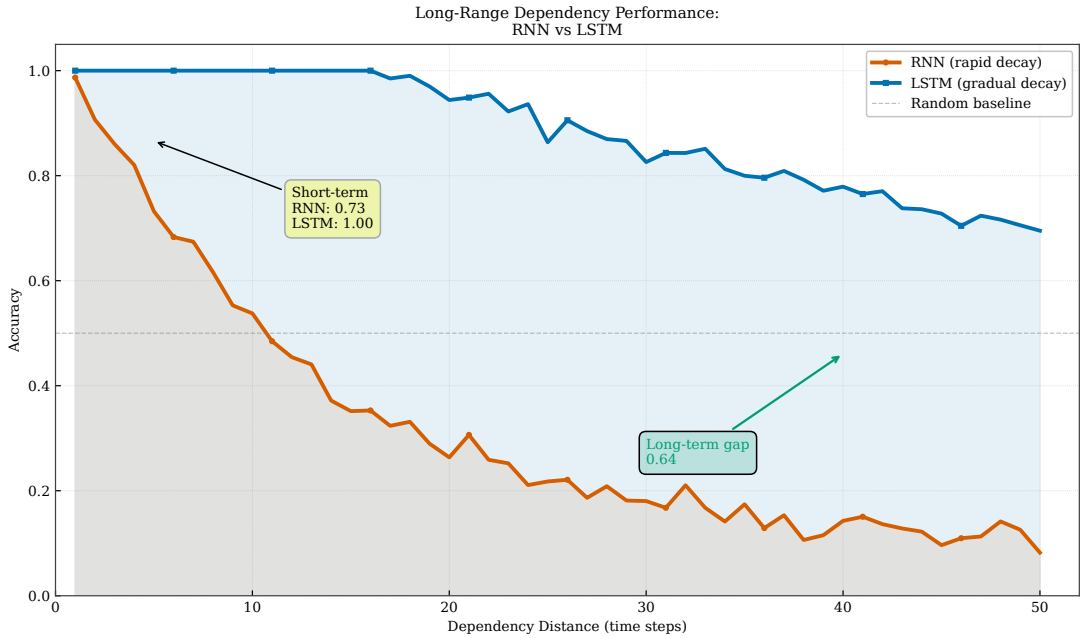


Figure 5.26: Effect of dependency distance on prediction accuracy. The plot shows accuracy of predicting a dependent word as a function of the number of words between the dependent elements. Vanilla RNN (red) shows rapid accuracy decay, dropping below 60% by distance 10 and approaching chance (50%) by distance 20. LSTM (green) maintains higher accuracy across all distances, staying above 70% even at distance 30. This difference quantifies the long-range memory advantage of gated architectures over vanilla RNNs.

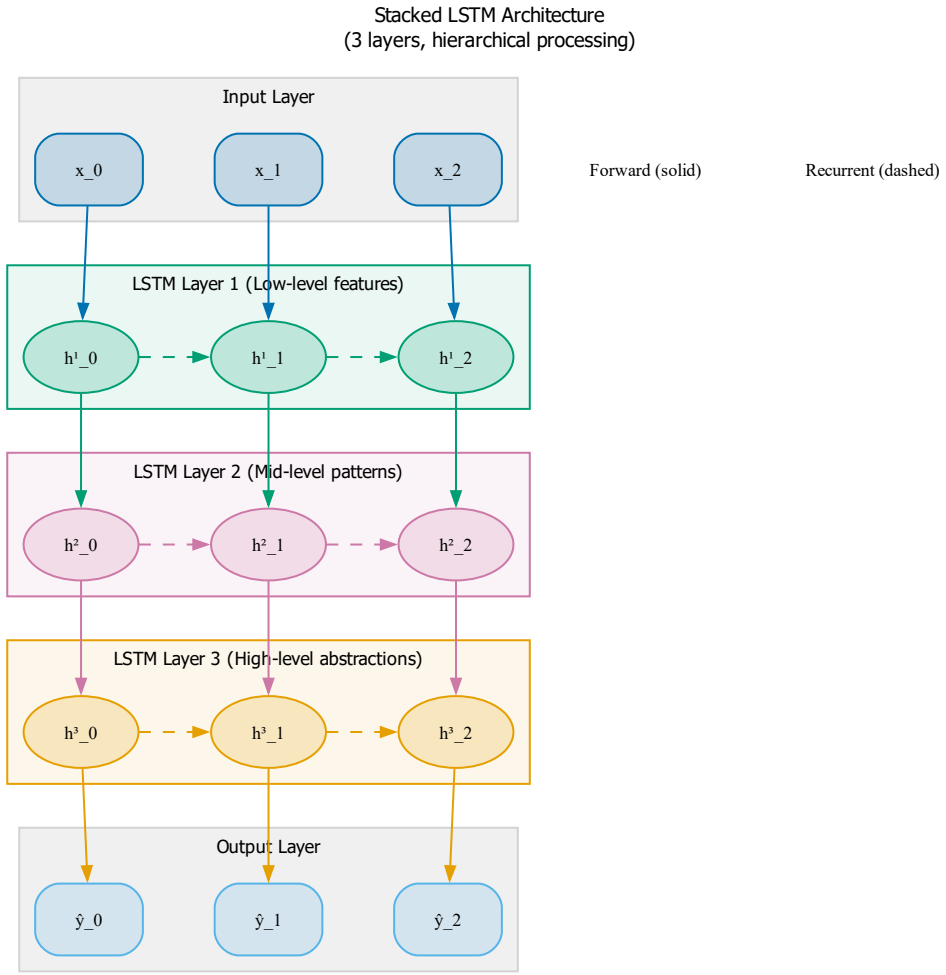


Figure 5.27: Stacked LSTM architecture for hierarchical representation. Multiple LSTM layers are stacked vertically, with the hidden state output of each layer serving as input to the next. The bottom layer processes raw word embeddings and captures low-level features (syntax, local patterns). Middle layers capture phrase-level and clause-level structure. Top layers capture discourse-level and long-range semantic patterns. This hierarchical organization, learned through training, enables richer context representation than single-layer architectures. Dropout is applied between layers to prevent overfitting.

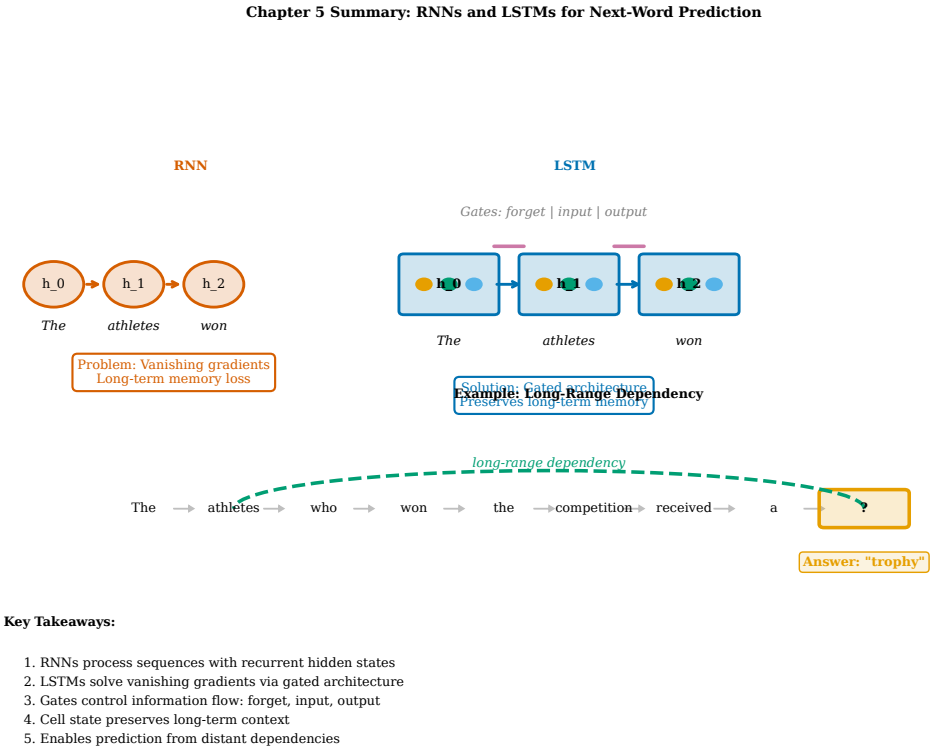


Figure 5.28: Visual summary of Chapter 5 key concepts. The diagram synthesizes the chapter's main ideas: sequential processing with hidden states (left), the LSTM gating mechanism that enables long-range memory (center), and the running example demonstrating successful long-range dependency handling (right). The key insight is that gated architectures solve the vanishing gradient problem through additive cell state updates, enabling models to learn dependencies spanning dozens or hundreds of time steps. This capability is essential for effective language modeling but still relies on compressing history into a fixed-size vector.

We can now predict better because:

- **Variable-length context:** Hidden states encode entire sequence history, not just fixed windows, enabling context-dependent predictions regardless of sequence length
- **Long-range dependencies:** LSTM gates preserve relevant information across many time steps by creating gradient highways through the cell state
- **Dynamic representations:** The same word receives different hidden state representations based on its sequential context, enabling disambiguation of polysemy
- **Foundation for sequence modeling:** Recurrent architectures provide the conceptual foundation for understanding modern sequence models, including Transformers

Next: Chapter **??** introduces the attention mechanism, which overcomes the sequential bottleneck by allowing the model to directly access any position in the input sequence. Rather than compressing all history into a single hidden state, attention computes weighted combinations of all past representations, enabling efficient parallel computation and more effective long-range dependency modeling. This leads to the Transformer architecture, which has become the foundation of modern large language models.

Exercises

1. **Forward pass calculation.** Given a 3-word sequence with embeddings $\mathbf{e}_1 = [1,0]$, $\mathbf{e}_2 = [0,1]$, $\mathbf{e}_3 = [1,1]$, initial hidden state $\mathbf{h}_0 = [0,0]$, weights $\mathbf{W}_{xh} = \begin{pmatrix} 0.5 & 0.3 \\ 0.2 & 0.4 \end{pmatrix}$, $\mathbf{W}_{hh} = \begin{pmatrix} 0.1 & 0.2 \\ 0.3 & 0.1 \end{pmatrix}$, and bias $\mathbf{b}_h = [0,0]$, compute $\mathbf{h}_1$, $\mathbf{h}_2$, and $\mathbf{h}_3$ for a vanilla RNN.

2. **Hidden dimension trade-offs.** Explain the trade-off between hidden state dimension d_h and model capacity. How does doubling d_h affect (a) the number of parameters, (b) computational cost per time step, and (c) the model's ability to memorize training data versus generalize?

3. **LSTM gate activation patterns.** For the sentence "The cat sat on the mat", predict which of the three LSTM gates (forget, input, output) would have highest activation when processing the word "sat". Justify your prediction based on the linguistic function of each gate.

4. **Forget gate interpretation.** In the sentence "The trophy that the athletes won was large", what forget gate value (high or low) should the model learn for the "trophy" information while processing the relative clause "that the athletes won"? Explain how this enables correct prediction of "was".

5. **Cell state update calculation.** Given forget gate $\mathbf{f}_t = [0.9, 0.3]$, input gate $\mathbf{i}_t = [0.2, 0.8]$, previous cell state $\mathbf{c}_{t-1} = [1.0, 0.5]$, and candidate $\tilde{\mathbf{c}}_t = [0.0, 1.0]$, compute the new cell state $\mathbf{c}_t$. Explain what this update accomplishes in terms of preserving old information versus incorporating new information.

6. **Parameter count comparison.** For hidden dimension $d_h = 256$ and input dimension $d = 128$, calculate the exact number of parameters (weights and biases) for (a) vanilla RNN, (b) GRU, and (c) LSTM. Show your work and express each as a percentage of LSTM's parameter count.

7. **BPTT gradient flow.** Explain why the gradient $\frac{\partial \mathscr{L}}{\partial \mathbf{h}_1}$ (gradient at the first hidden state) is typically much smaller in magnitude than $\frac{\partial \mathscr{L}}{\partial \mathbf{h}_T}$ (gradient at the final hidden state) for a vanilla RNN. How does LSTM's cell state address this problem?

8. **Truncated BPTT trade-offs.** A language model is trained with truncated BPTT using window size $K = 20$. What is lost by not backpropagating beyond 20 steps? Describe a specific example of a dependency that would be missed by this truncation.

9. **Long-range dependency analysis.** Identify three English constructions besides subject-verb agreement that create long-range dependencies (where words separated by 10 or more tokens must agree or relate). For each, explain what information must be preserved and how LSTM's gating mechanism could accomplish this.

10. **Perplexity interpretation.** If an n-gram model achieves perplexity 150 and an LSTM achieves perplexity 80 on the same test set, quantify the improvement. In terms of "effective vocabulary size" at each prediction, how much more certain is the LSTM about its predictions?

11. **Bidirectional RNNs.** ($\star$) In a bidirectional RNN, two RNNs process the sequence in opposite directions and their hidden states are concatenated. How does this change the context representation $\mathbf{h}_t$? What types of predictions could benefit from bidirectional context that unidirectional (left-to-right) RNNs cannot capture?

12. **Stacked architectures.** ($\star$) Describe how stacking multiple LSTM layers creates hierarchical representations. What might each layer in a 3-layer stacked LSTM learn to represent (give specific examples for language modeling)? How does depth trade off against width (hidden dimension) for fixed parameter budget?

Bibliography

Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

Index

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.

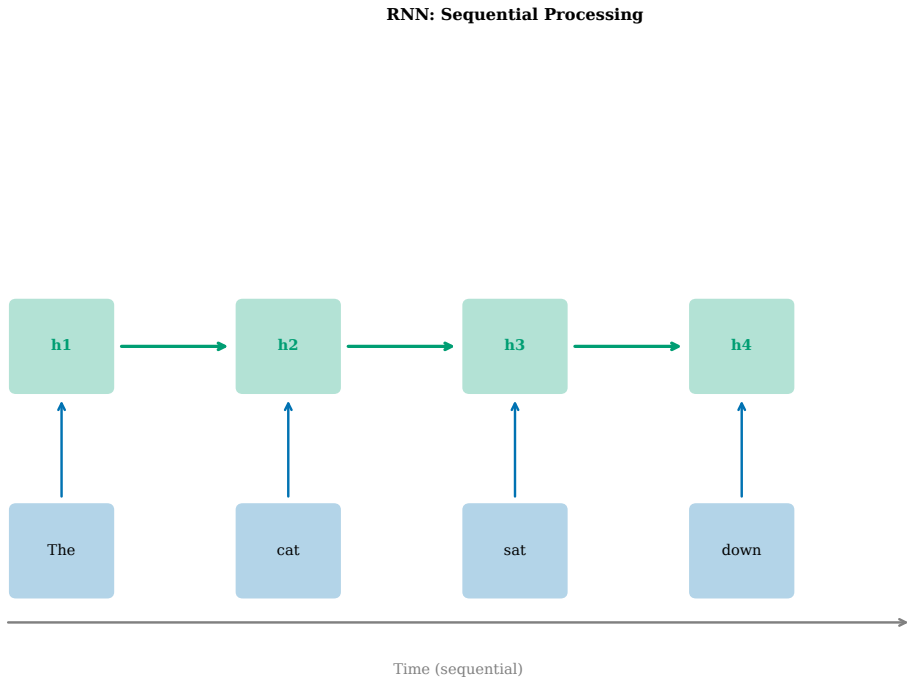


Figure 6.1: Sequential processing in RNNs versus parallel processing in transformers. The RNN (left) computes hidden states sequentially, creating a computational bottleneck where each step depends on the previous. The transformer (right) processes all positions simultaneously, with attention connections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

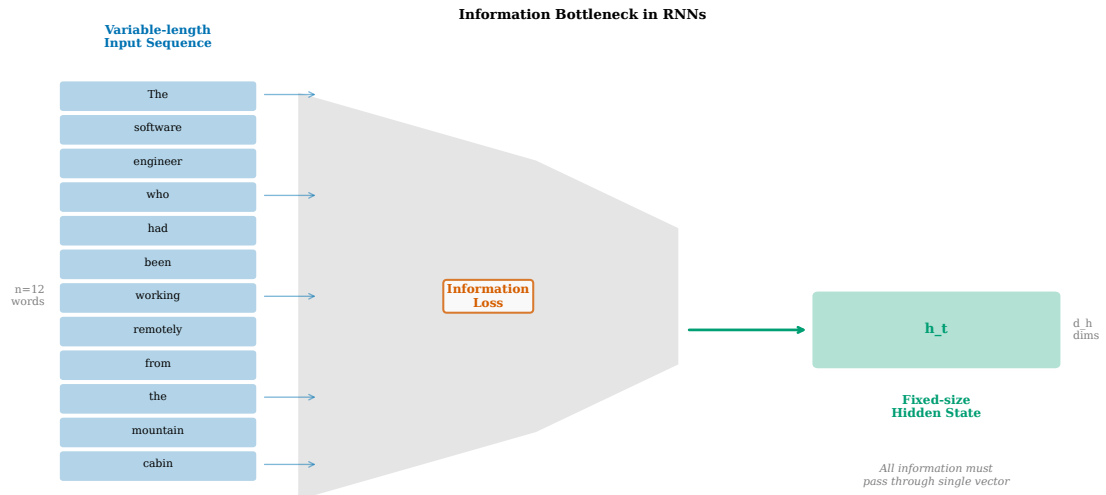


Figure 6.2: The information bottleneck in RNNs. As sequence length increases (x-axis), the amount of information that must pass through the fixed-size hidden state $\mathbf{h}$ grows, but the state's capacity remains constant. Transformers avoid this bottleneck by maintaining separate representations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

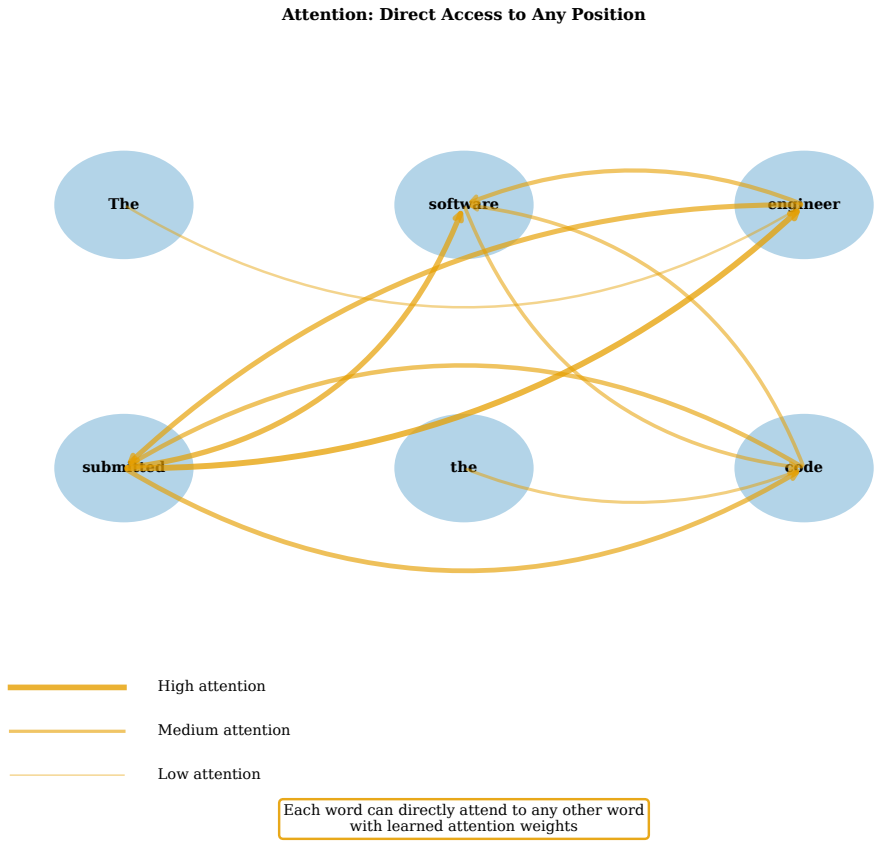


Figure 6.3: Intuitive view of attention as a soft lookup mechanism. Given a query (what we're trying to predict), we compute similarity scores with all keys (previous positions), convert these to weights via softmax, and return a weighted combination of values. The weights are shown as connection thickness.

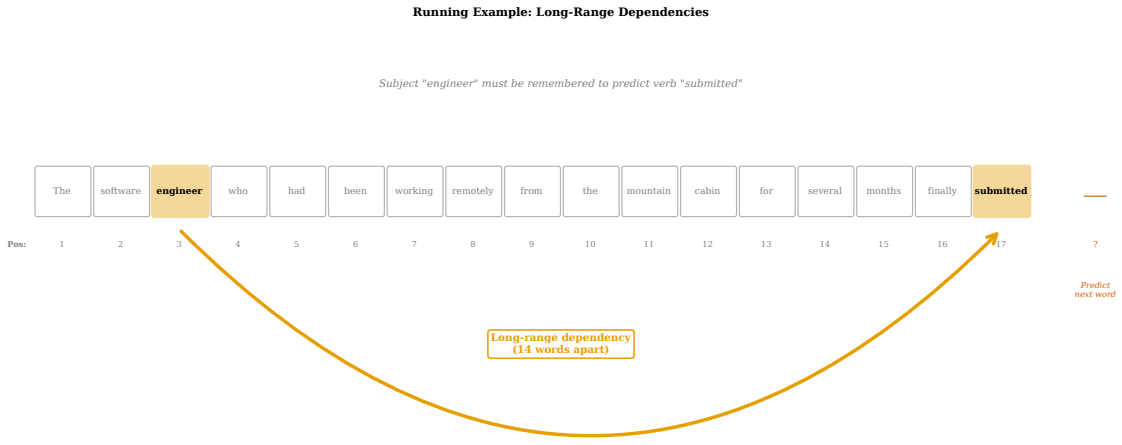


Figure 6.4: Our running example sentence with 22 context words. When predicting position 23, the model can attend to any previous position. Relevant words for subject-verb agreement ("engineer", "submitted") and semantic constraints ("code", "software") may be scattered throughout the context.

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.

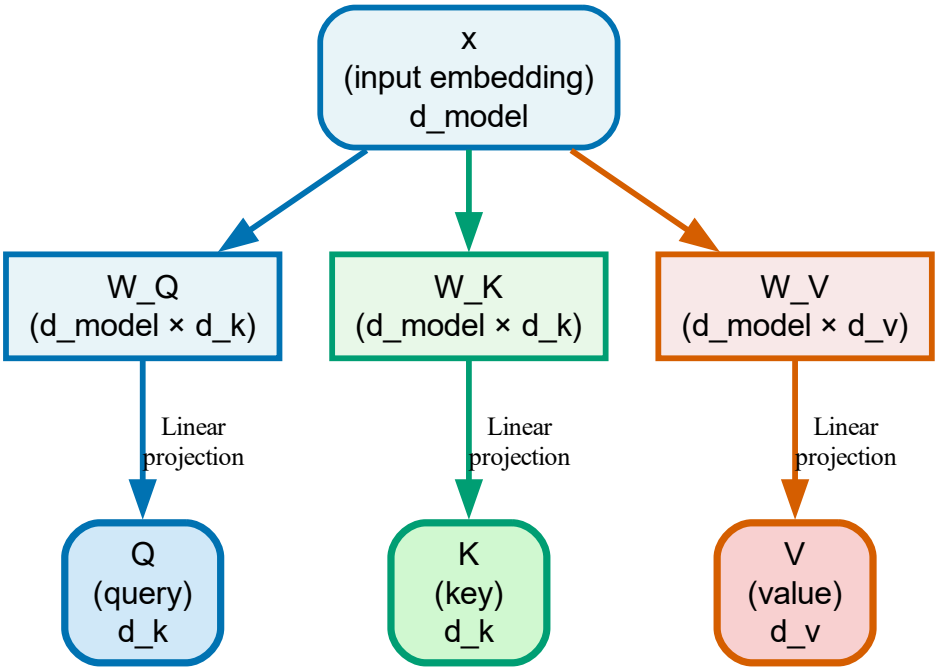


Figure 6.5: Projecting token embeddings into query, key, and value spaces. Each embedding $\mathbf{e}[i]$ is multiplied by three learned weight matrices to produce $\mathbf{q}[i]$, $\mathbf{k}[i]$, and $\mathbf{v}[i]$. These projections decouple the roles of matching (Q-K) and information retrieval (V).

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-

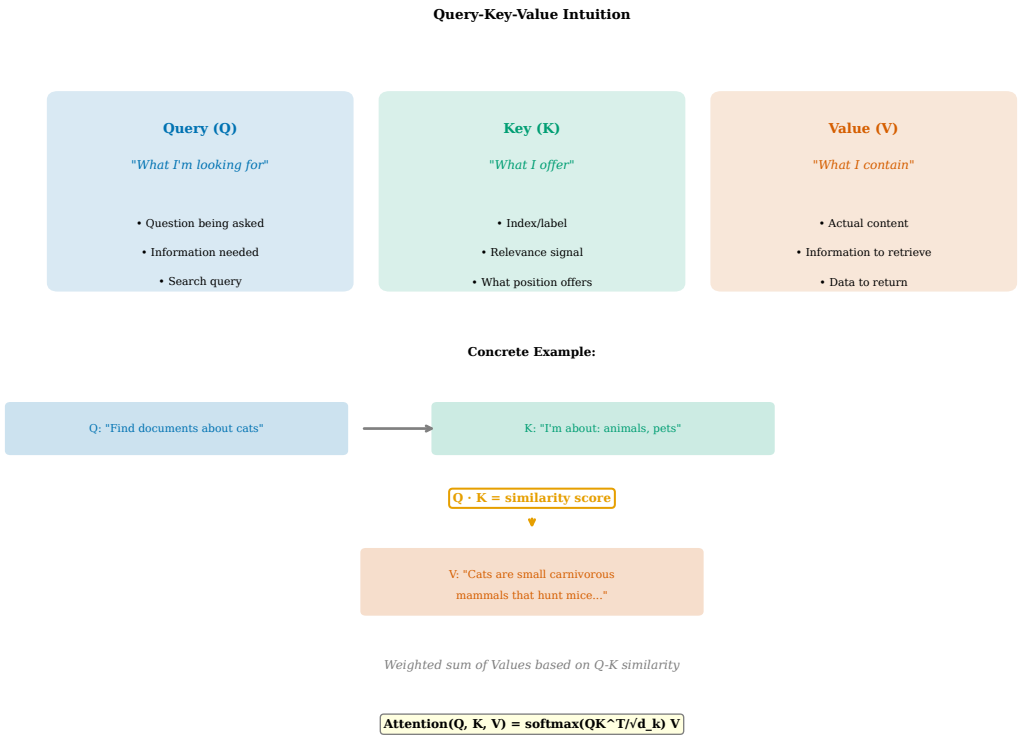


Figure 6.6: Computing attention scores via dot products. For each position i, we compute the dot product of $\mathbf{q}[i]$ with all keys $\mathbf{k}[j]$ for $j \leq i$. Higher dot products indicate stronger alignment between the query and key, suggesting that position j is relevant for position i.

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

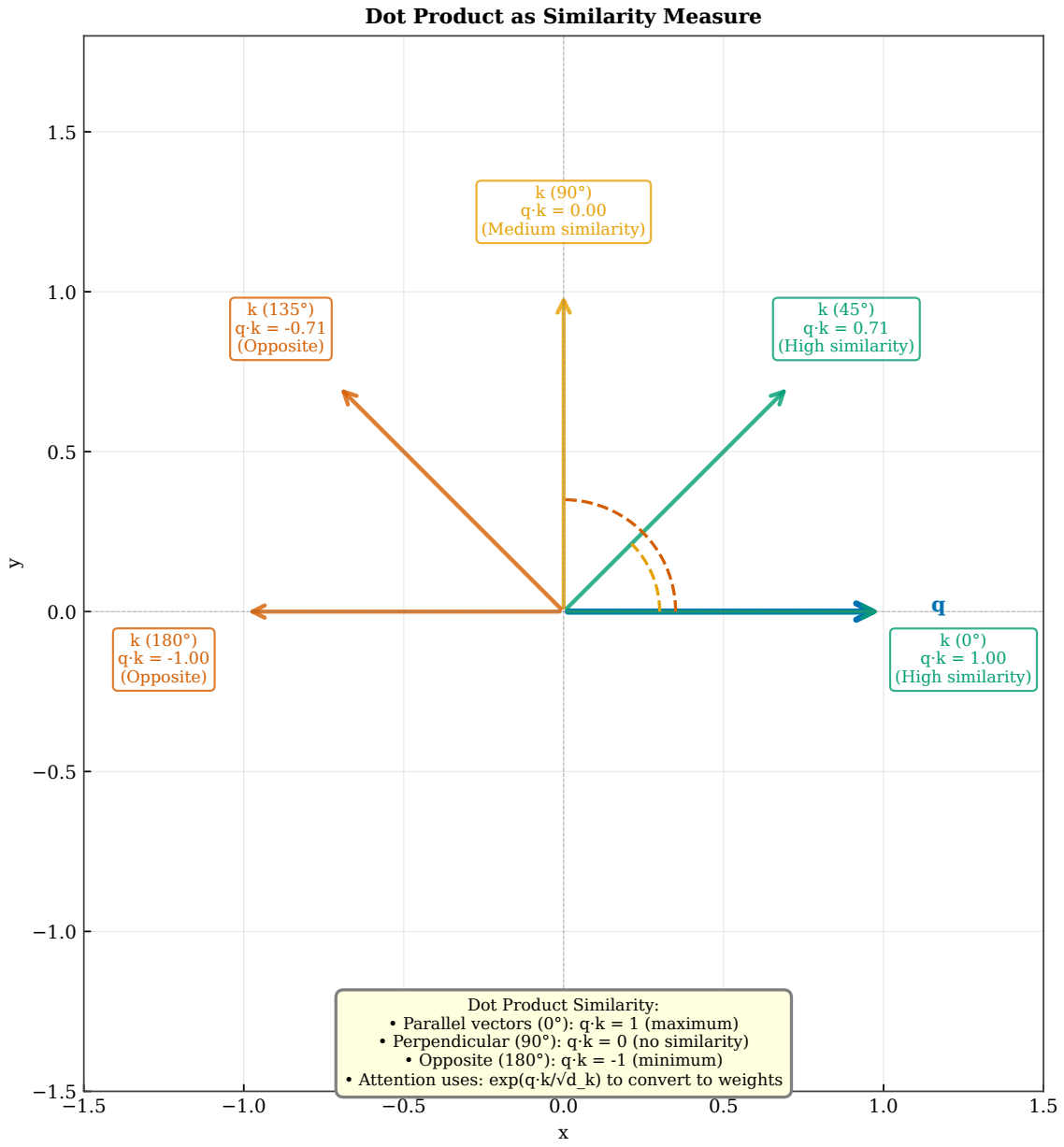


Figure 6.7: Converting attention scores to weights via softmax. The raw dot-product scores (which can be any real number) are exponentiated and normalized to sum to 1, producing a valid probability distribution over previous positions. Higher scores correspond to higher weights after normalization.

enter the softmax.

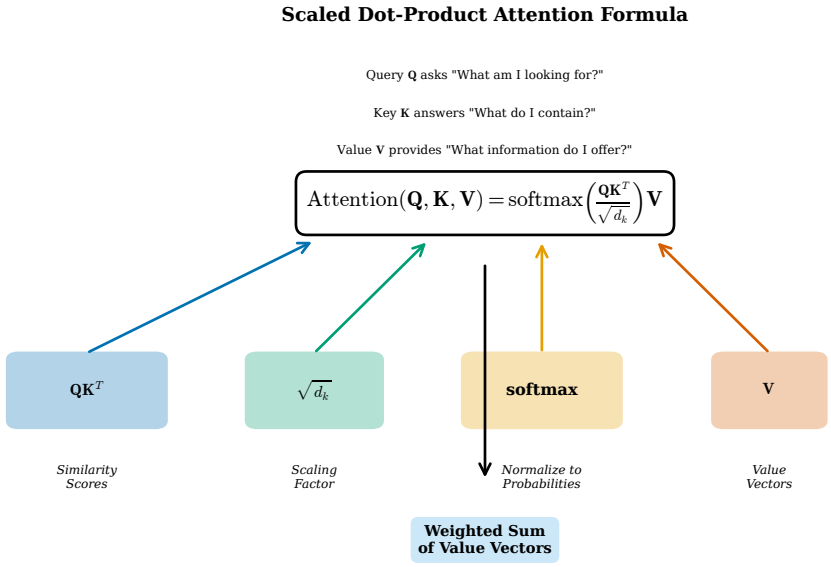


Figure 6.8: Computing the attention output as a weighted sum of values. Each value vector $\mathbf{v}[j]$ is scaled by its attention weight $\alpha[ij]$ and summed to produce the final output. Positions with higher weights contribute more to the output representation.

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-

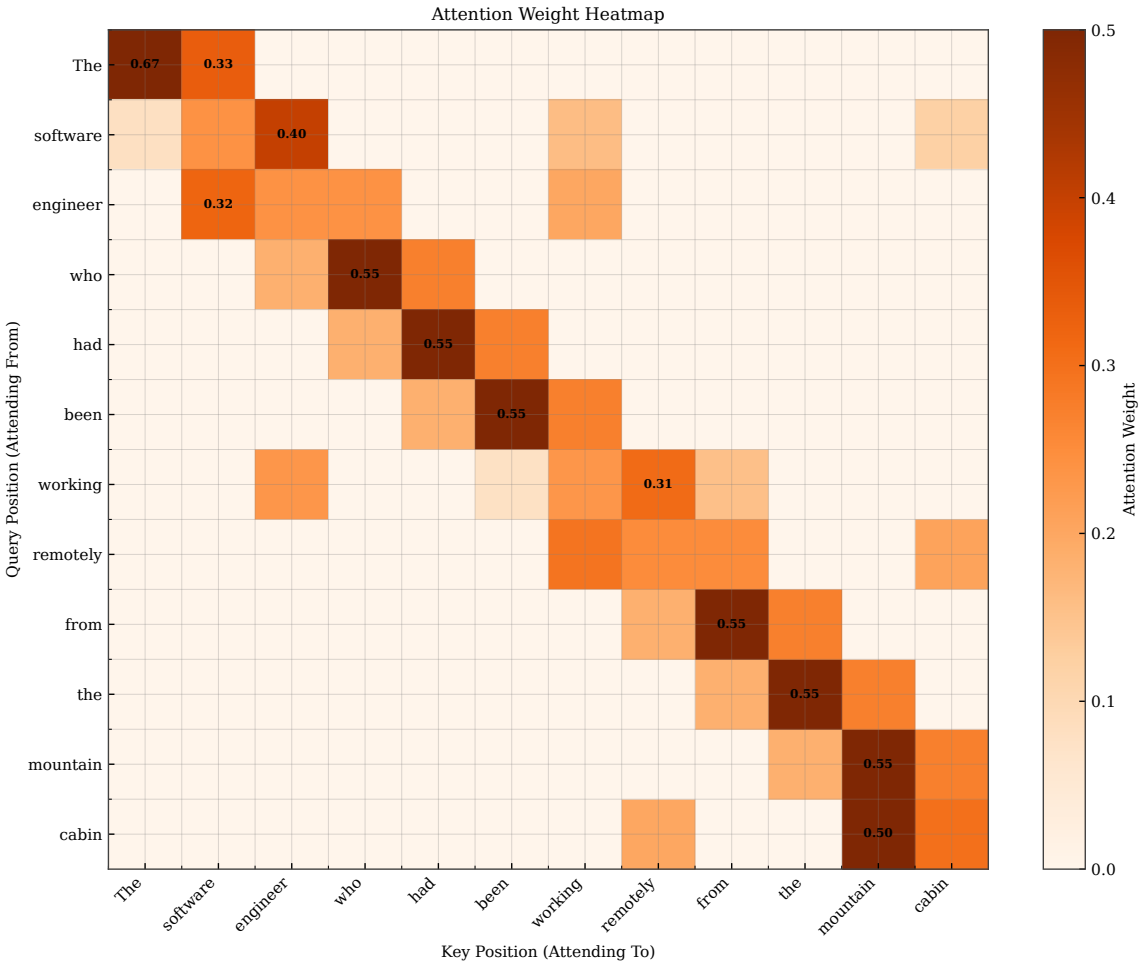


Figure 6.9: Attention weight matrix α for an entire sequence. Rows correspond to query positions (where we predict), columns correspond to key positions (where we look). Each entry $\alpha[ij]$ shows how much position i attends to position j. The matrix is lower-triangular due to causal masking (discussed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

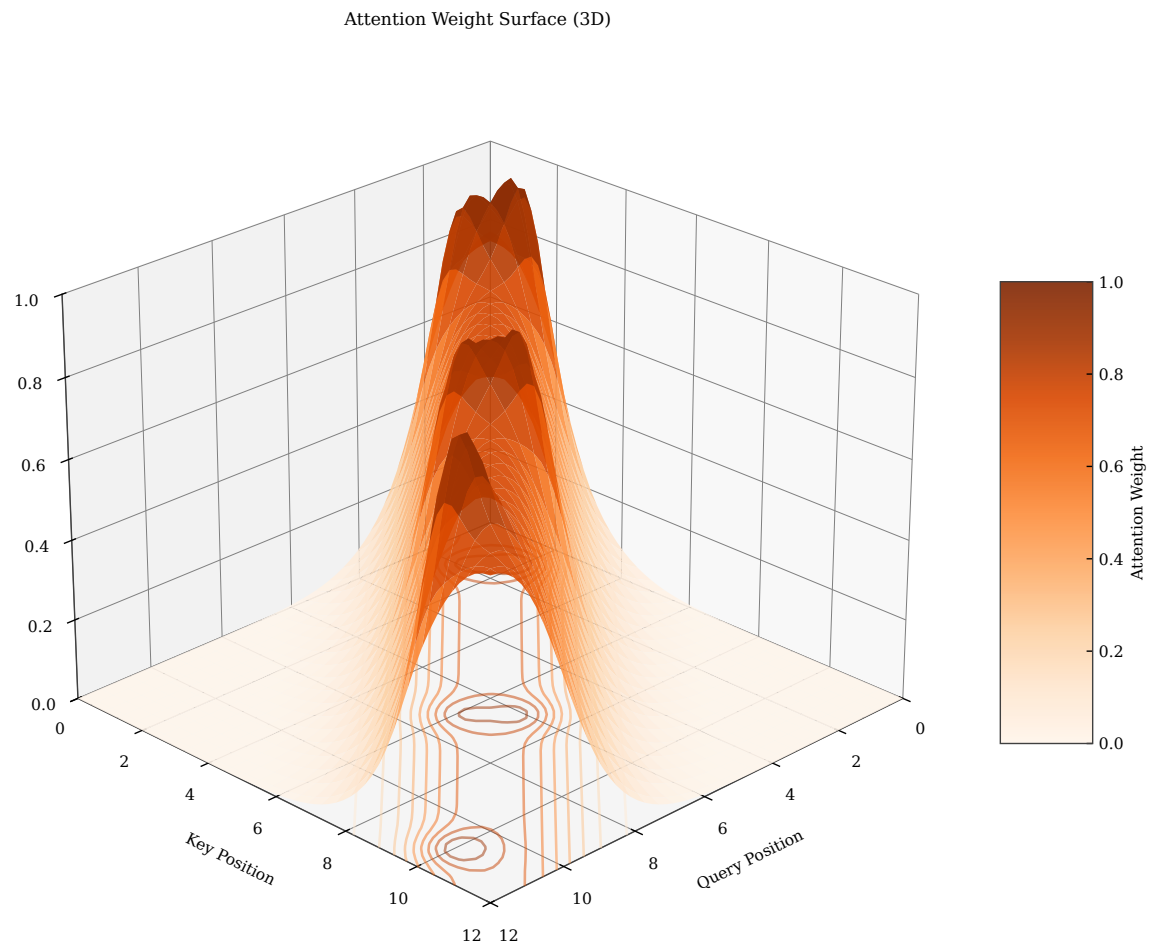


Figure 6.10: Attention weights for position 23 (predicting the next word) in our running example. The model attends most strongly to "code" (position 21), "submitted" (position 19), "engineer" (position 3), and "software" (position 2), which are semantically and syntactically relevant for predicting 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 $\sqrt{d_{\text{model}}}$

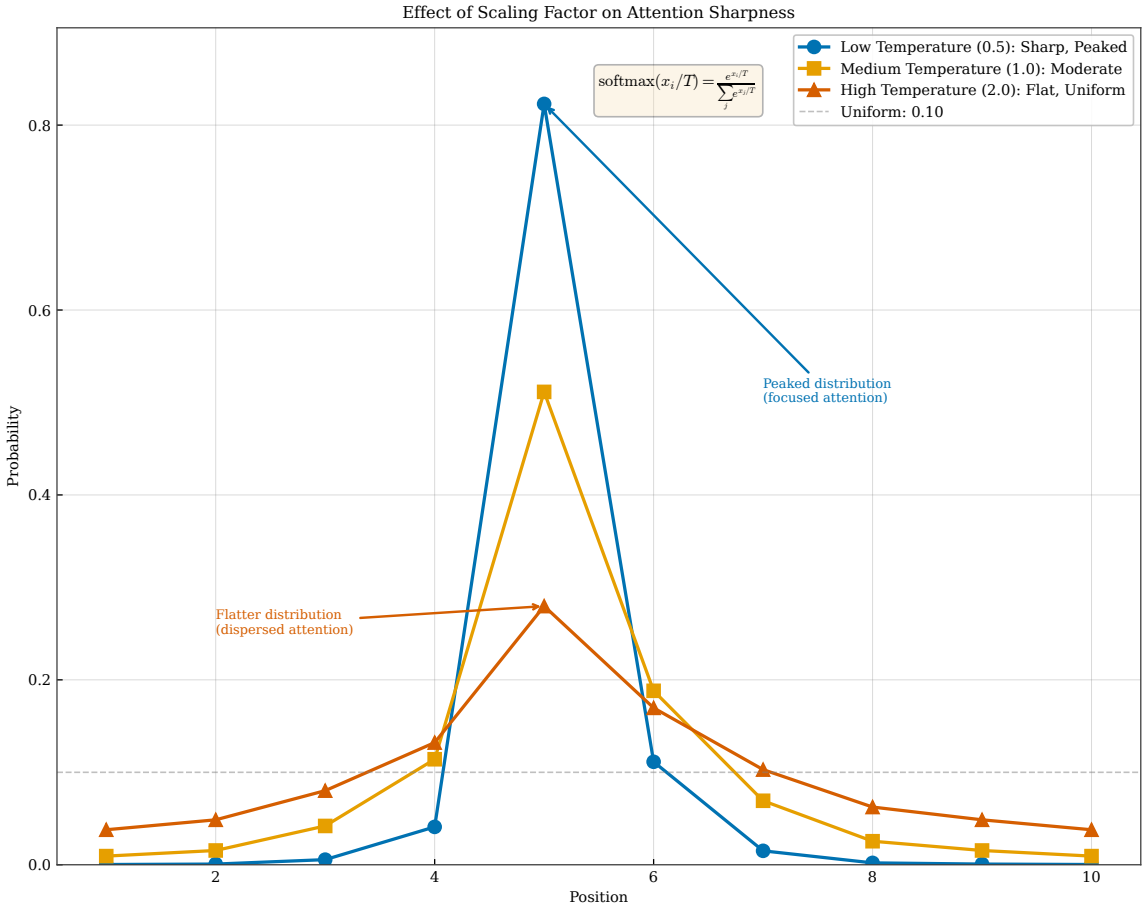


Figure 6.11: Three-dimensional visualization of attention weights as a surface. The height at position (i, j) represents $\alpha[ij]$. Peaks indicate strong attention, while valleys indicate low attention. The surface shows how attention evolves across the sequence.

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.

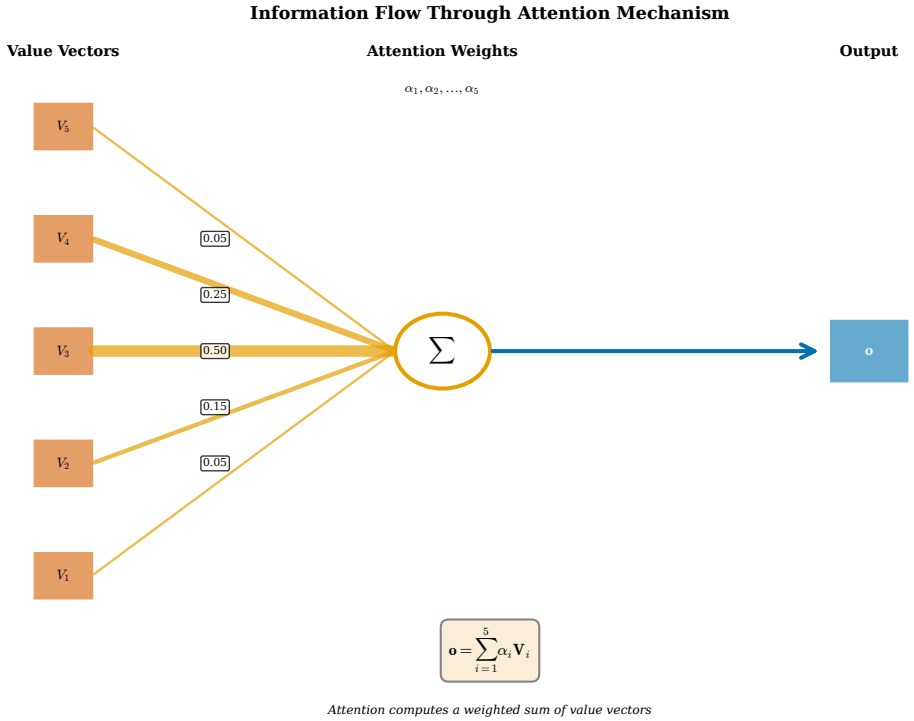


Figure 6.12: Effect of the scaling factor $1/\sqrt{d_{\text{model}}}$ on attention distributions. Without scaling (top), large dot products cause the softmax to produce near-one-hot distributions with negligible gradients. With scaling (bottom), the distribution is smoother, allowing gradients to flow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

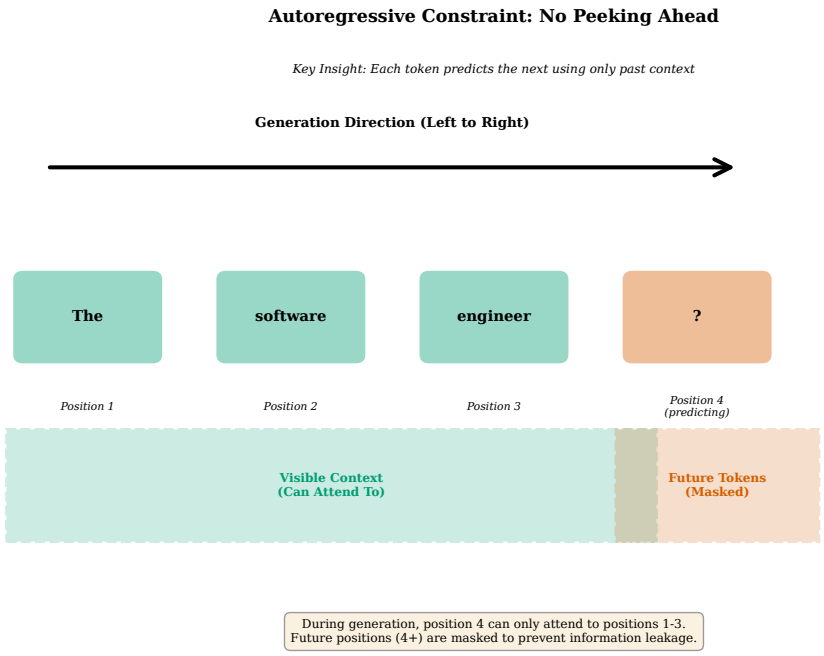


Figure 6.13: Causal mask structure for a sequence of length 8. White cells indicate allowed attention (score is computed normally), black cells indicate forbidden attention (score set to $-\infty$). The lower-triangular pattern ensures that each position can only attend to itself and earlier positions.

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

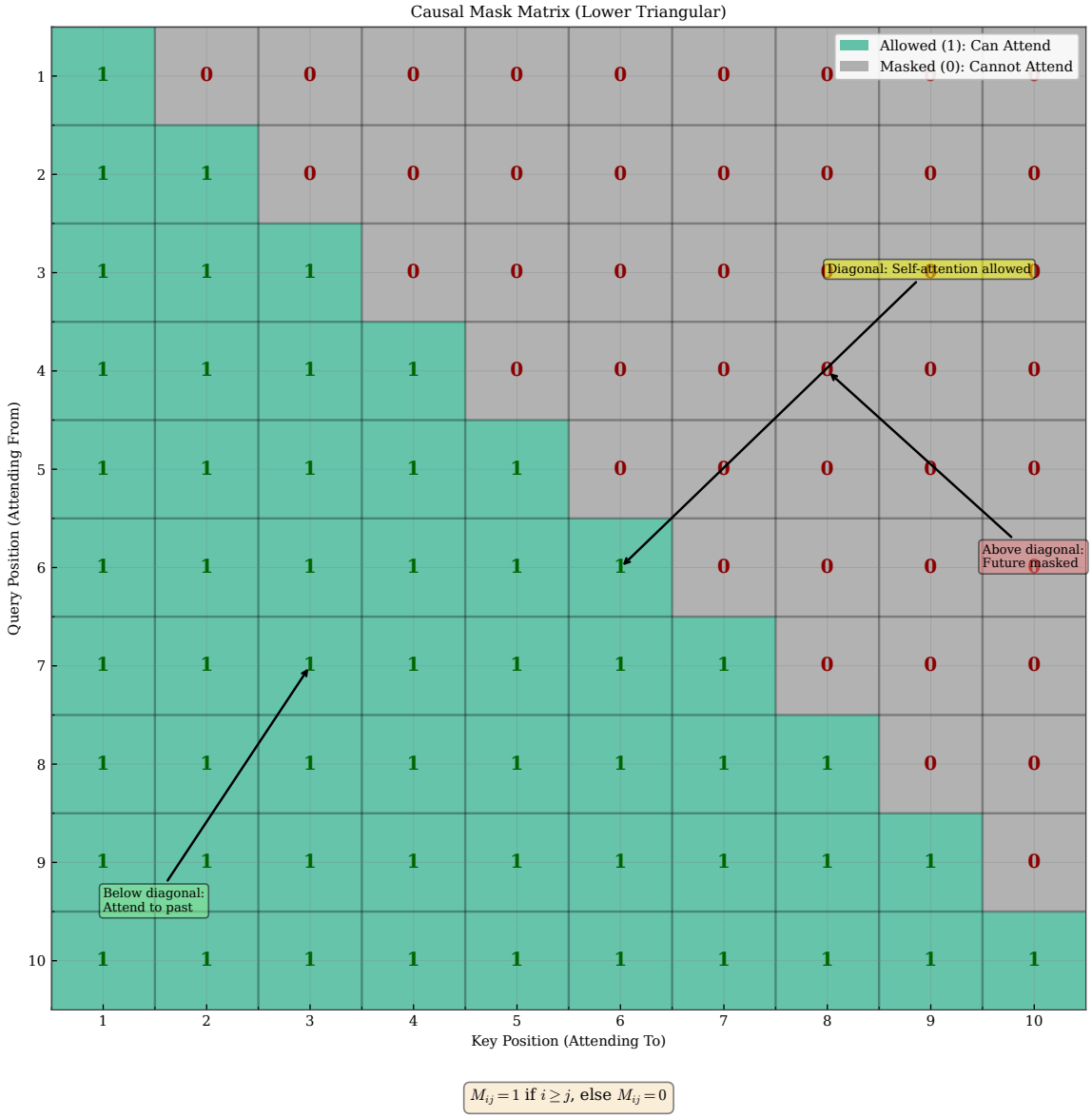


Figure 6.14: Attention weight matrix with causal masking applied. Compare to Figure 6.9: the upper triangle is now strictly zero. Each row i represents a valid probability distribution over positions 1 through i.

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

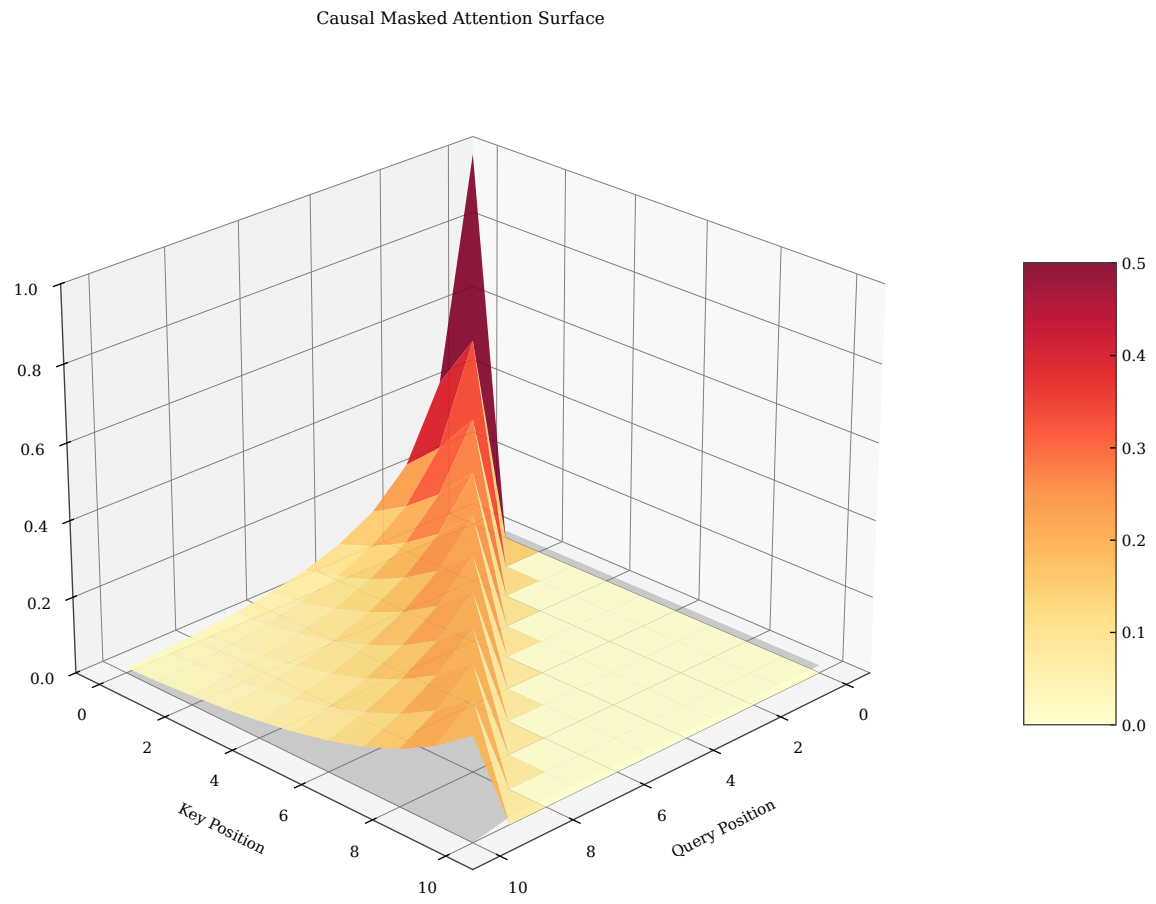


Figure 6.15: Applying causal masking to our running example. When predicting position 23, the model can attend to positions 1 through 22 (shown in color) but not to any future positions (grayed out). This matches the generation setting where future tokens are unknown.

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

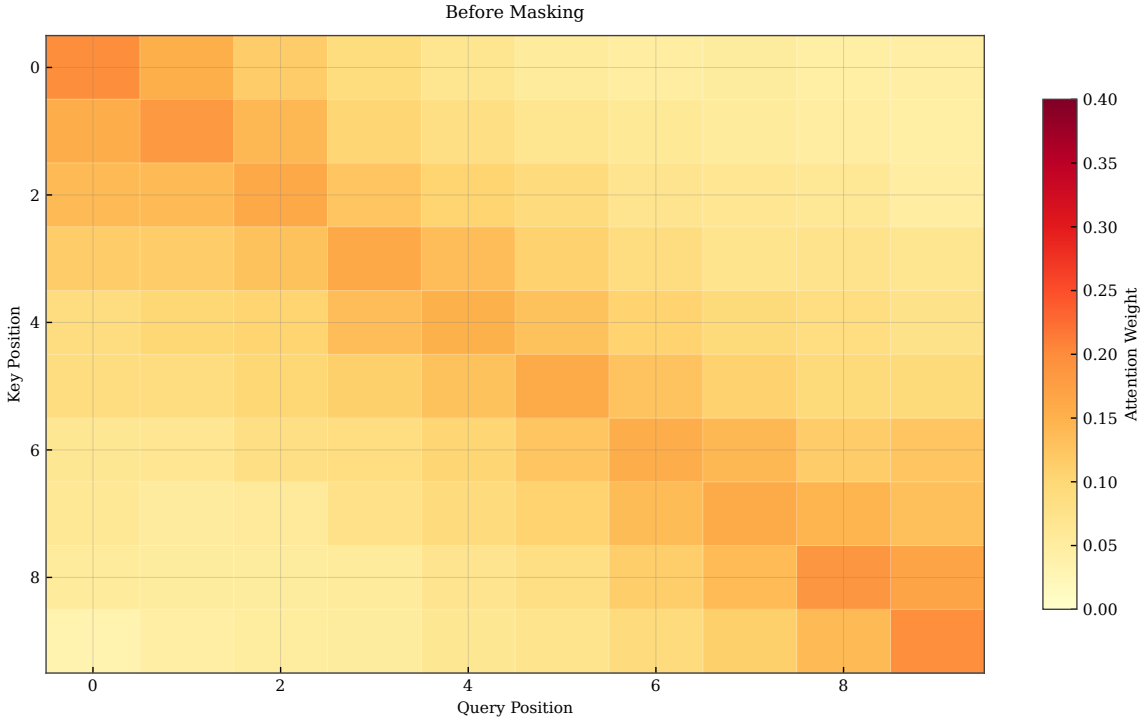


Figure 6.16: Three-dimensional view of causal masked attention. The surface is nonzero only in the lower triangle, forming a stepped pyramid shape. The sharp drop to zero at the diagonal boundary enforces the autoregressive constraint.

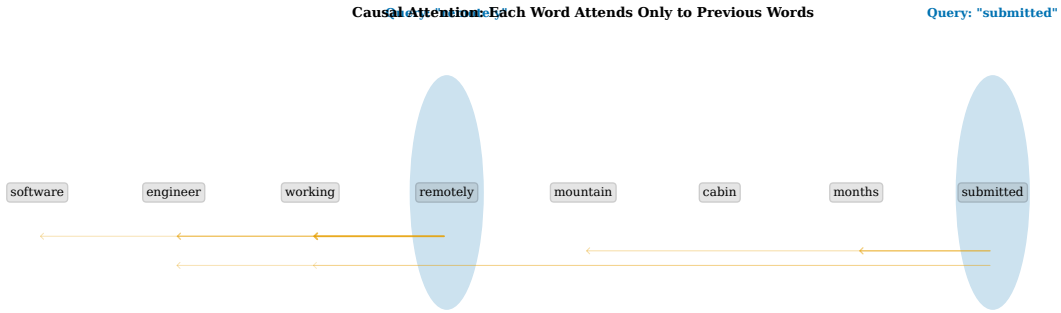


Figure 6.17: Autoregressive generation with causal attention. At step 1, the model attends only to the initial prompt. At step 2, it can attend to the prompt plus the first generated token. At step 3, the context includes all previously generated tokens. Each step uses causal masking to ensure consistency.

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

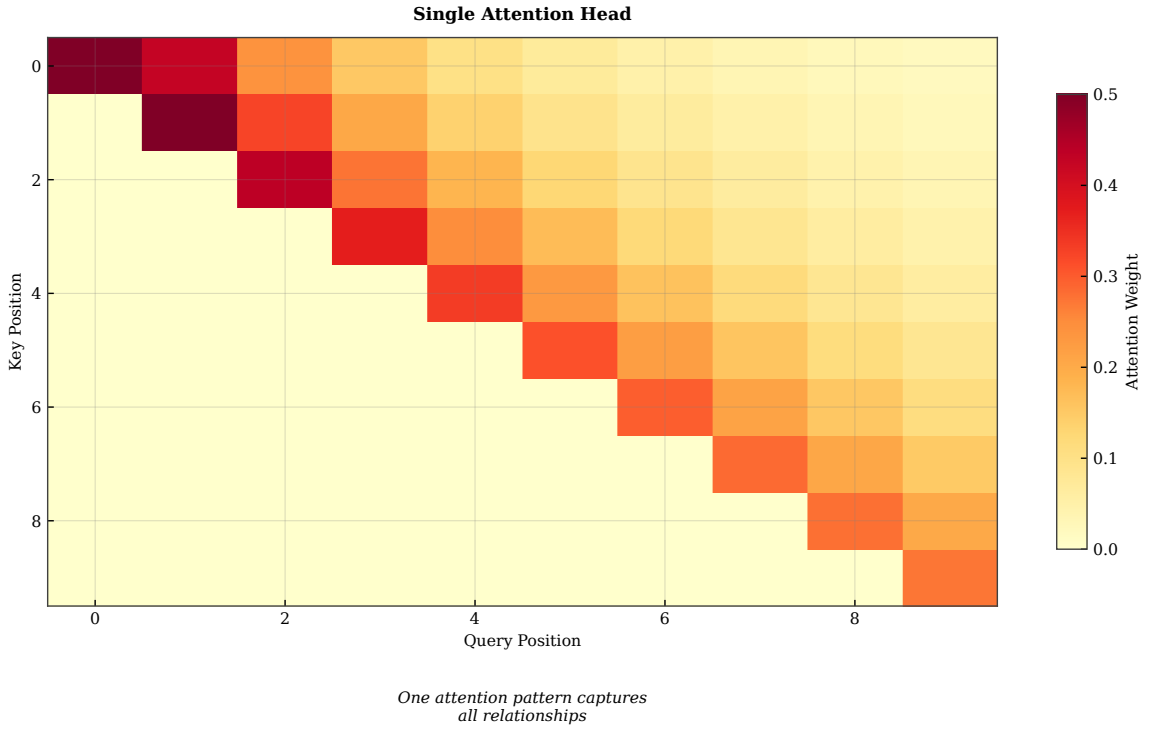


Figure 6.18: Multi-head attention architecture with $n_{\text{heads}} = 4$. The input embeddings are projected into queries, keys, and values for each head independently. Each head computes attention in its own d_{head}-dimensional subspace, producing a head output. All head outputs are concatenated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

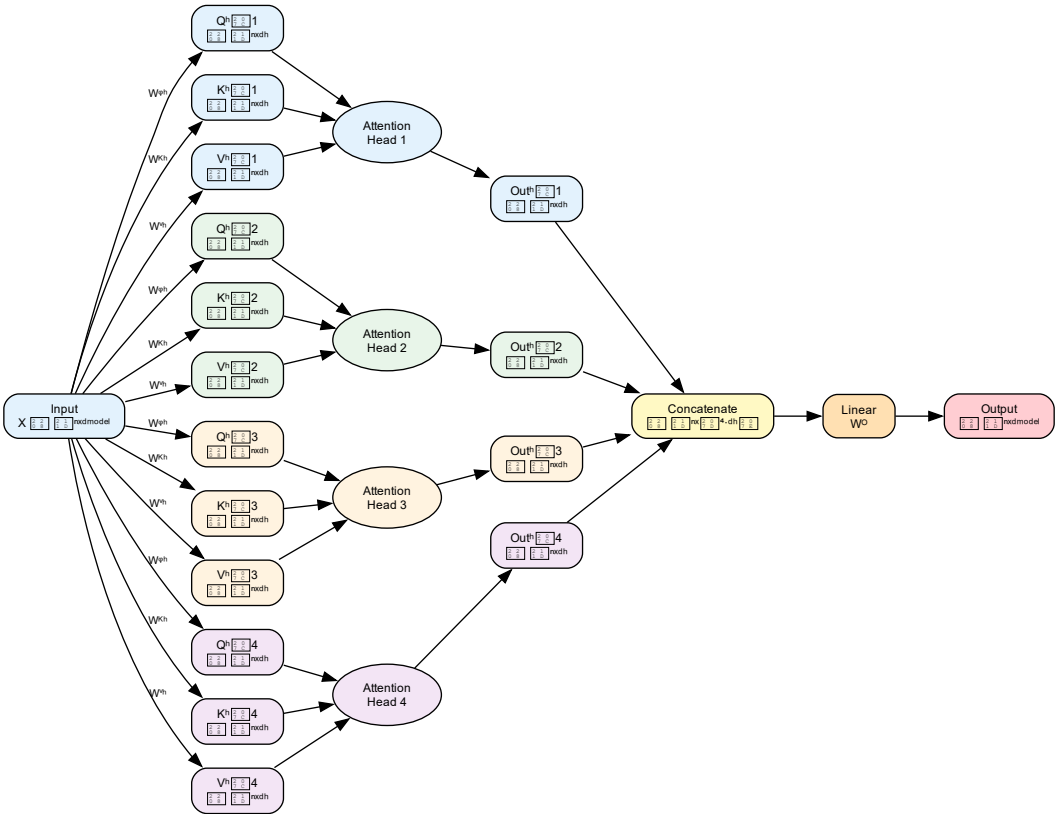
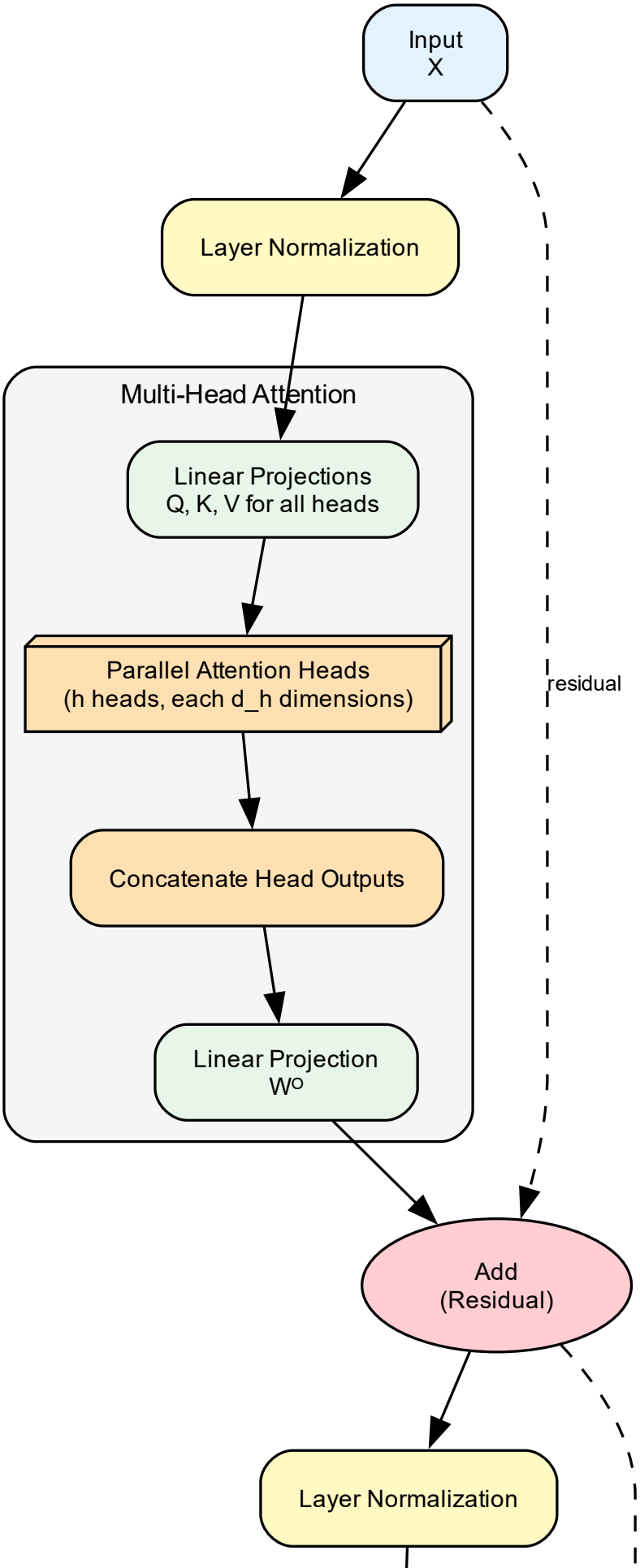


Figure 6.19: Example of head specialization in a trained transformer. Head 1 focuses on adjacent tokens (local context), Head 2 attends to the subject of the sentence (long-range syntactic dependency), Head 3 attends to semantically similar words (thematic coherence), and Head 4 shows a diffuse pattern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

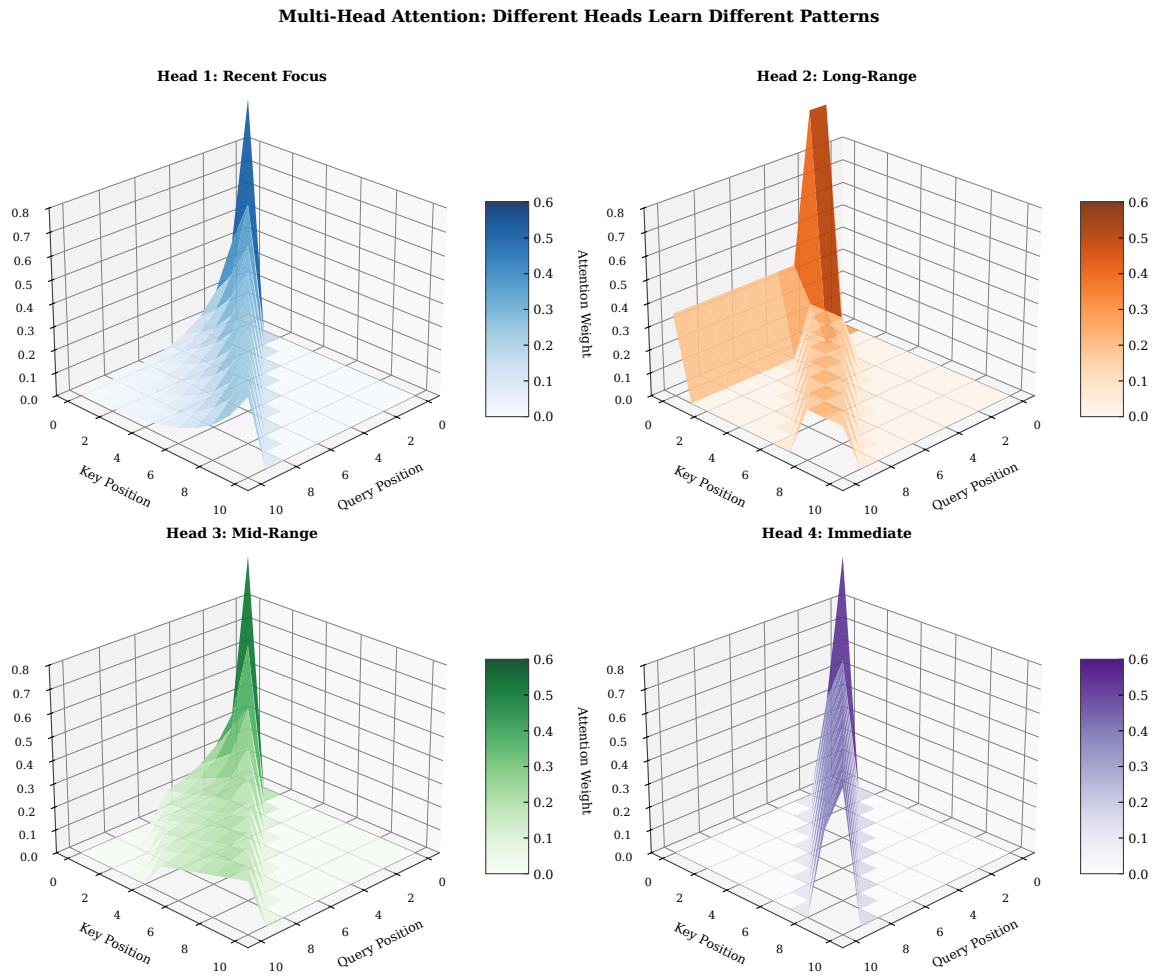


Figure 6.21: Multi-head attention applied to our running example. Different heads attend to different aspects of the context. Head 1 focuses on "code" and "submitted" (semantic and temporal cues), Head 2 focuses on "engineer" and "software" (domain and subject), demonstrating complementary views of relevance.

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.

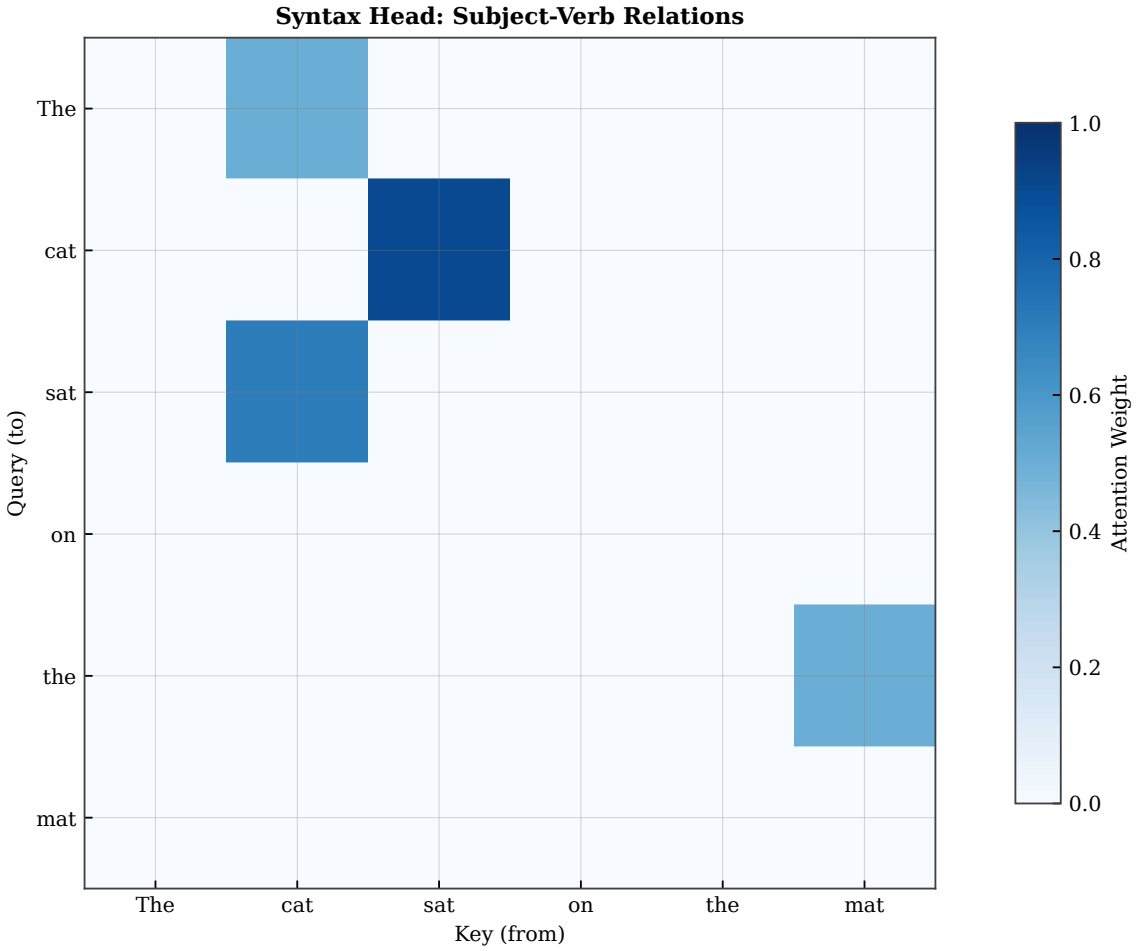


Figure 6.22: Three-dimensional visualization of attention patterns across four heads. Each surface represents the attention weights $\alpha[ij]$ for one head. The diversity in surface shapes reflects the different roles each head has learned, from sharp peaks (focused attention) to broad plateaus (diffuse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

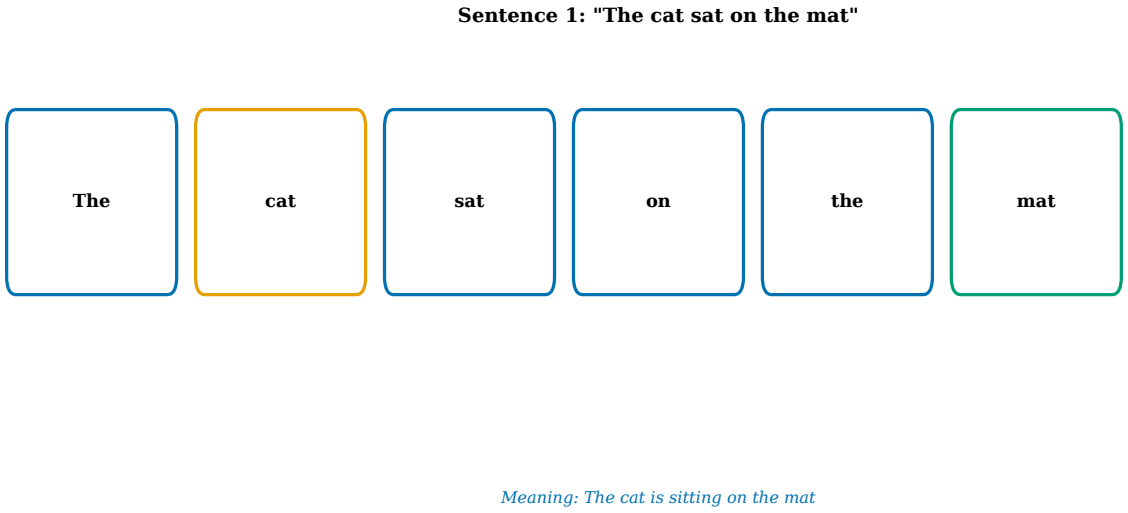


Figure 6.23: Sinusoidal positional encodings for positions 0 to 50 across 128 dimensions. Each row represents a position, each column a dimension. The color intensity indicates the value of the encoding (ranging from -1 to 1). Low dimensions oscillate rapidly, high dimensions oscillate slowly, creating 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 $\mathbf{e}[t] +$

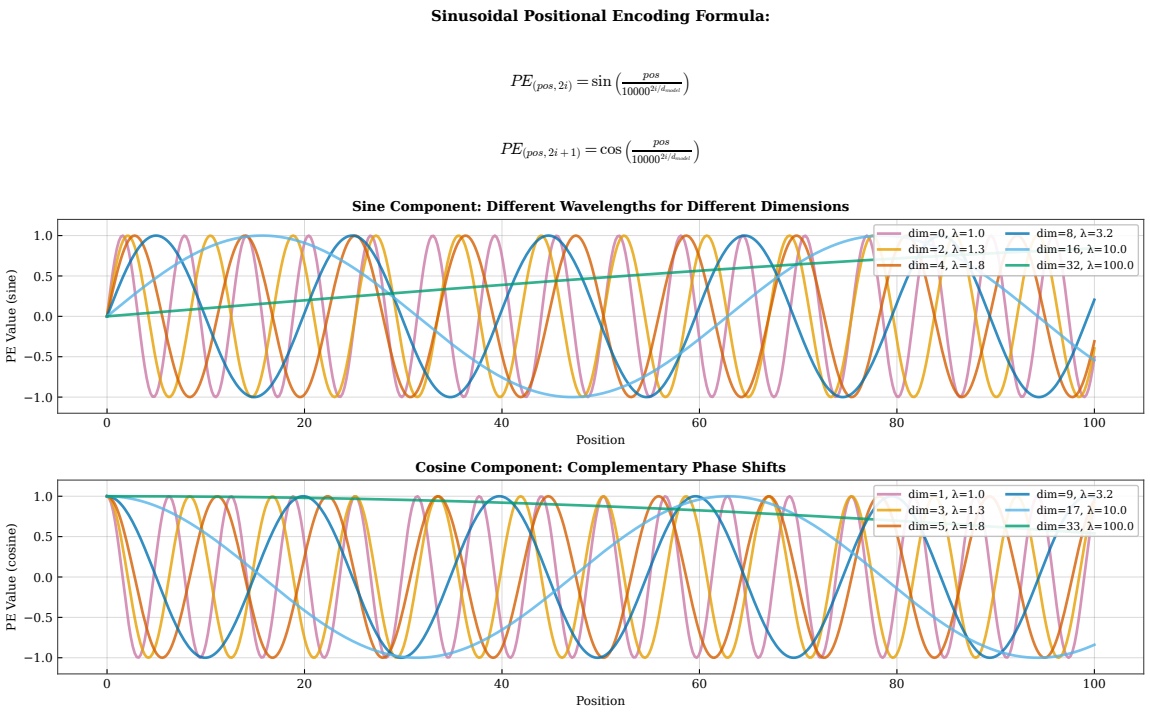


Figure 6.24: Adding positional encodings to token embeddings. Each token embedding $\mathbf{e}[t]$ (representing word identity) is combined element-wise with the positional encoding PE(t) (representing position in the sequence). The resulting vector contains both content and position information.

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

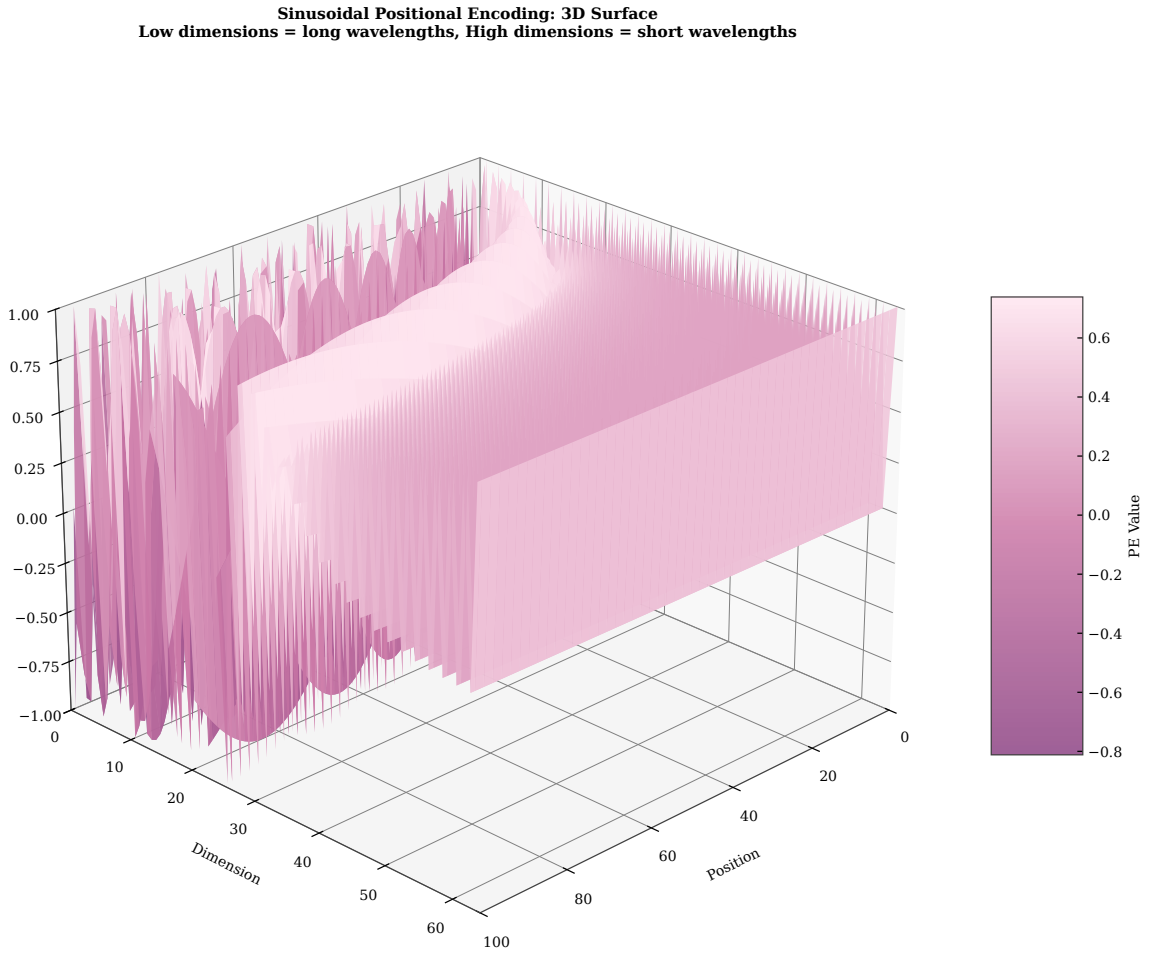


Figure 6.25: Three-dimensional view of sinusoidal positional encodings. The surface shows how the encoding value varies across position (x-axis) and dimension (y-axis). The rippling pattern reflects the sinusoidal structure, with wavelength decreasing as dimension increases.

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 $\cos(\theta_i - \theta_j)$.

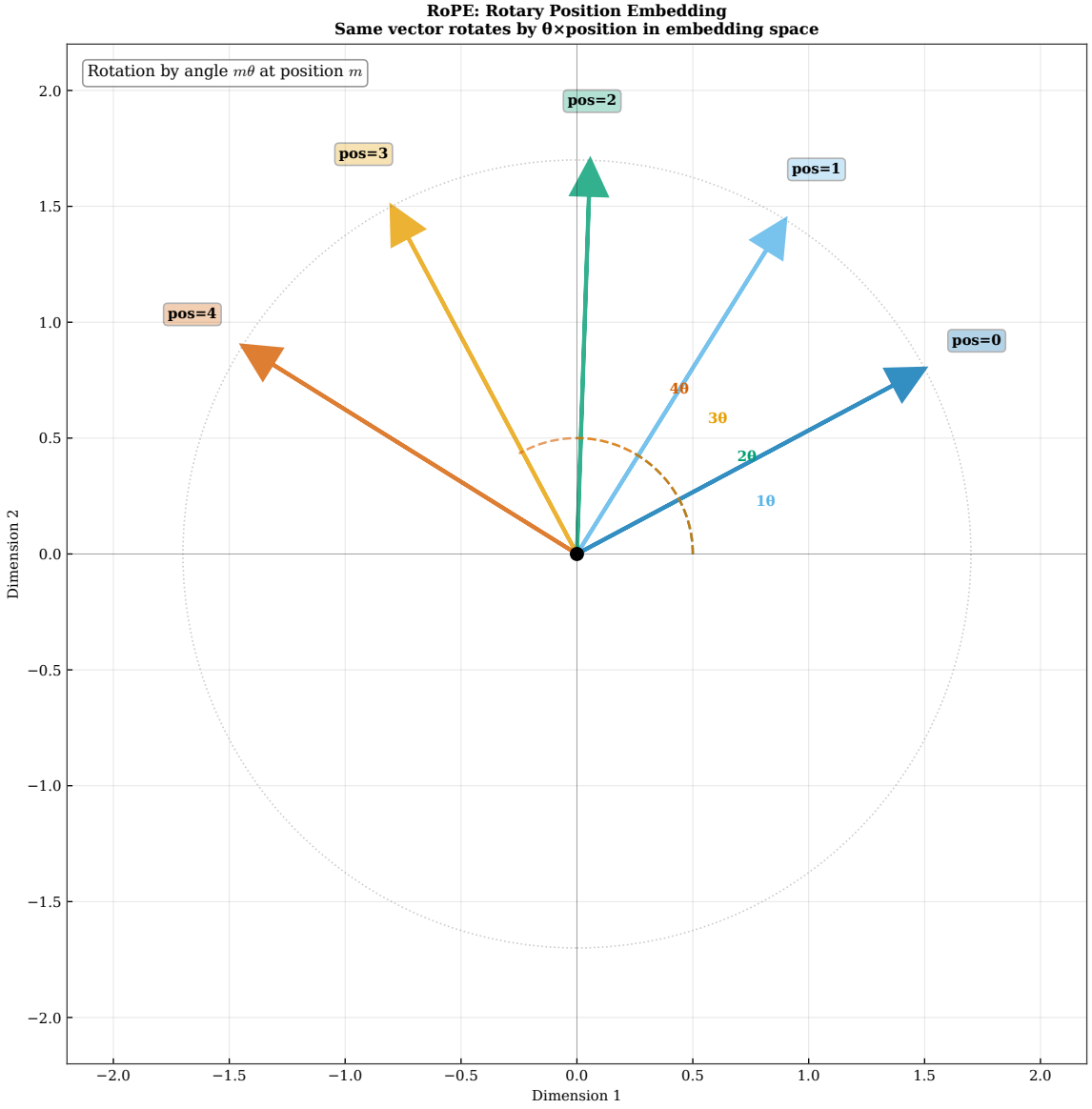


Figure 6.26: Comparison of sinusoidal (left) and learned (right) positional encodings. Learned encodings are treated as parameters and optimized via gradient descent. Both approaches yield similar performance in practice, but sinusoidal encodings generalize better to sequence lengths not seen during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

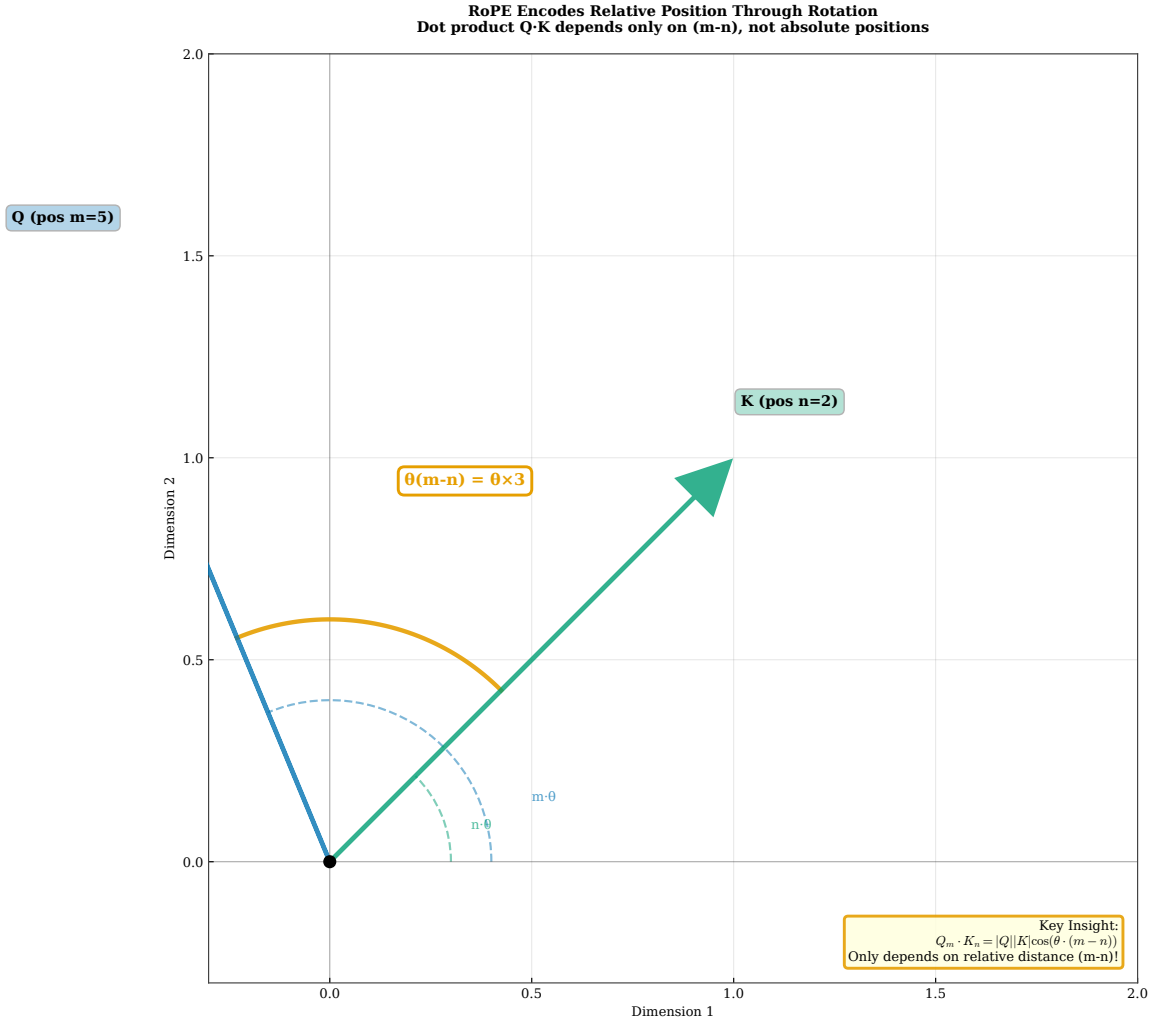


Figure 6.27: Rotary Position Embedding (RoPE) applies position information via rotation in the complex plane. Instead of adding positional encodings to embeddings, RoPE rotates the query and key vectors by an angle proportional to their positions. The relative rotation between positions i and j depends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

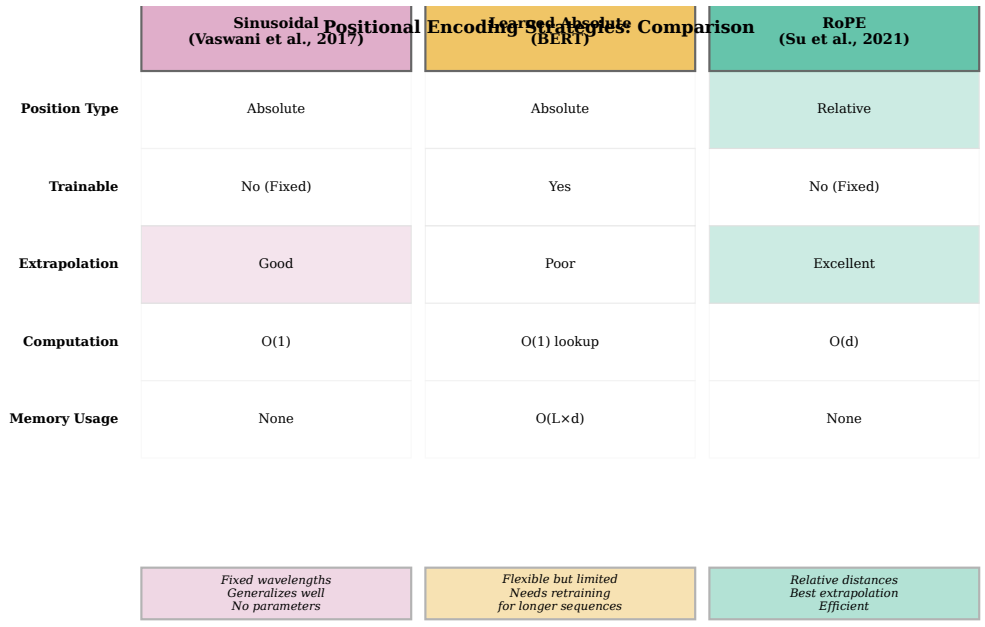


Figure 6.28: Performance comparison of different positional encoding schemes on a language modeling task. Sinusoidal, learned, and RoPE achieve similar perplexity on in-distribution sequence lengths, but RoPE generalizes better when evaluated on longer sequences than seen during training (extrapolation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

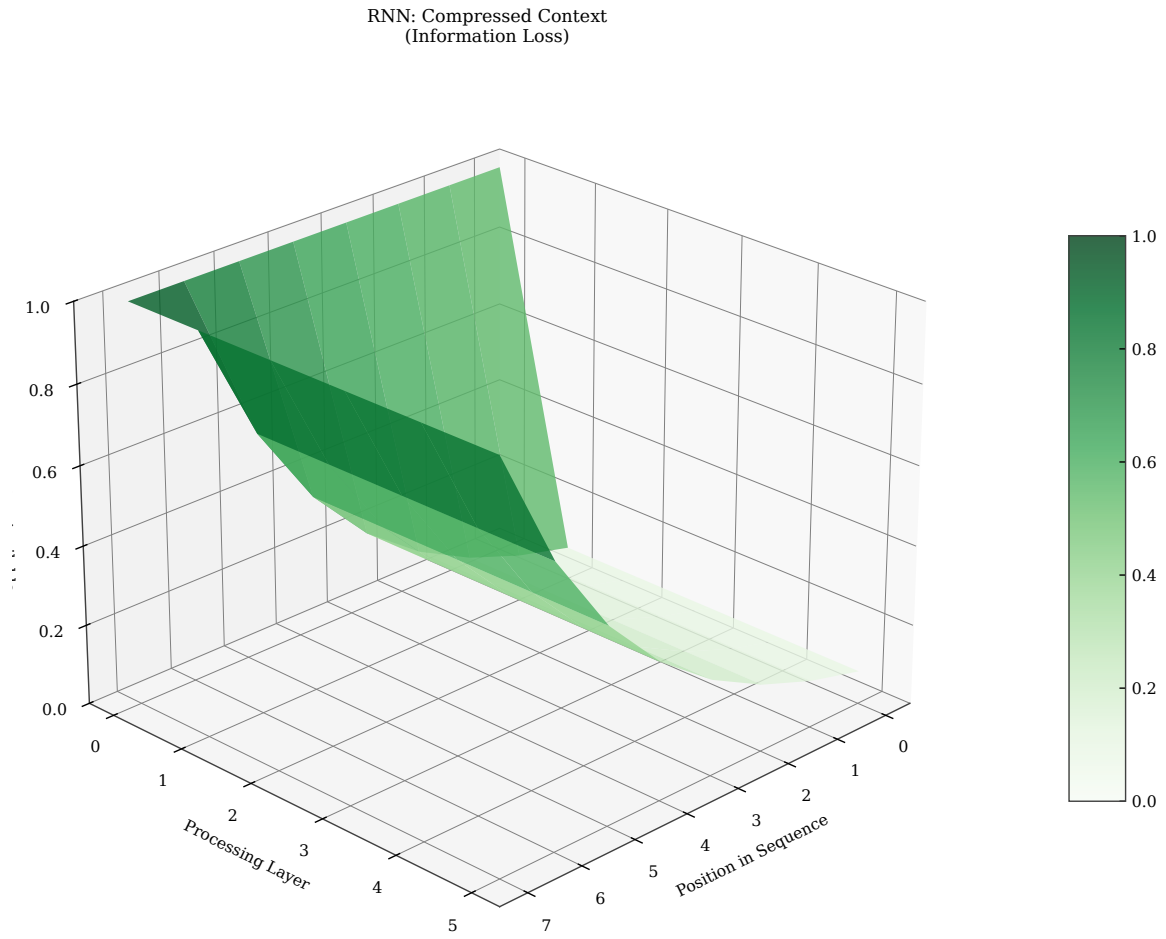
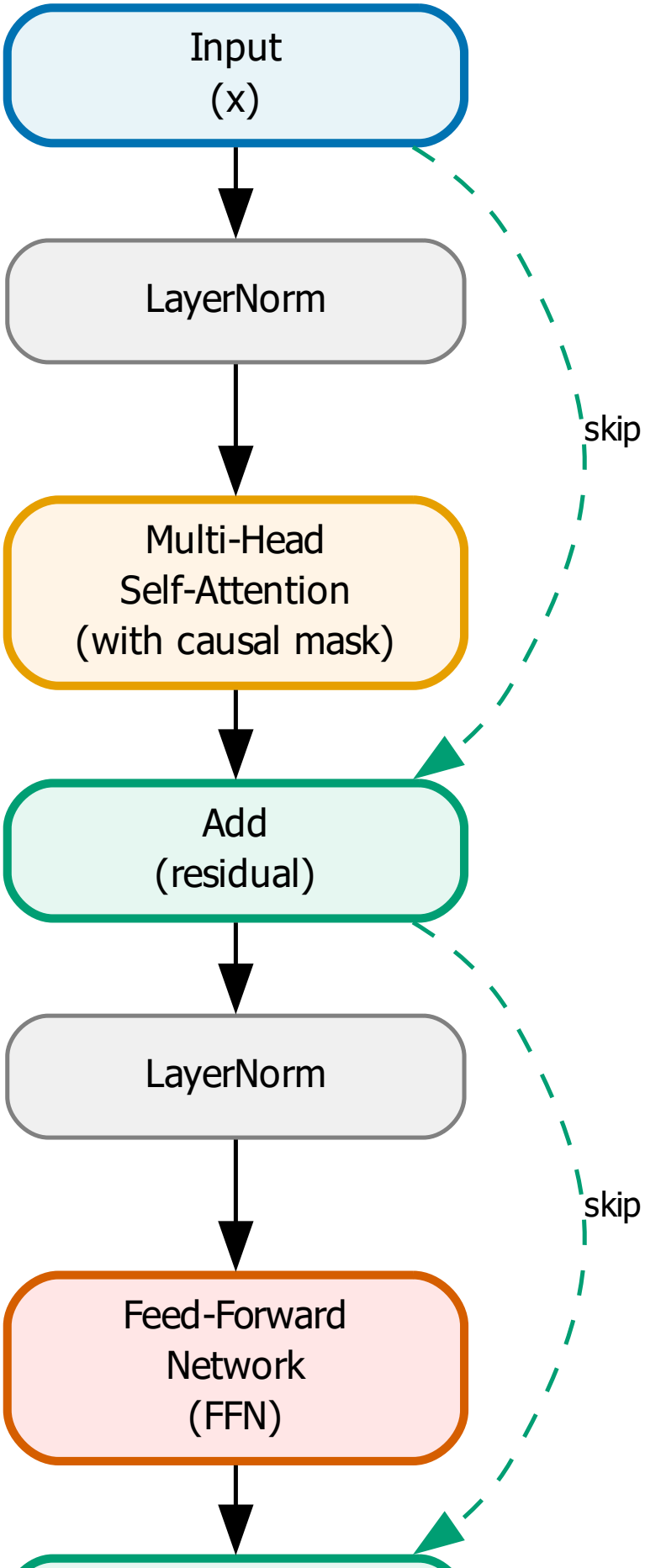


Figure 6.29: A single transformer decoder block. The input passes through multi-head self-attention (with causal masking), residual connection, and layer normalization, then through a position-wise feed-forward network, residual connection, and layer normalization. The output is passed to the next block.

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)

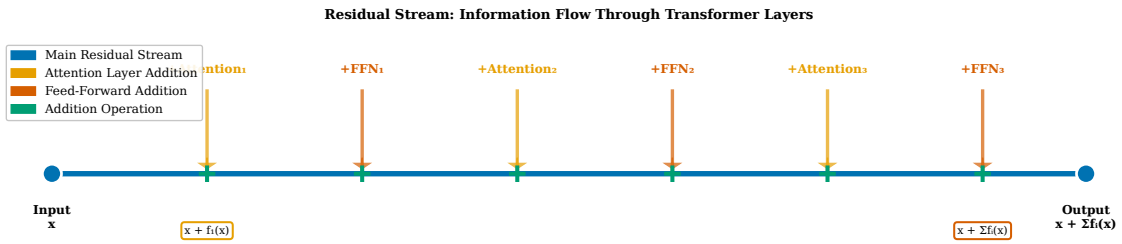


Figure 6.31: Three-dimensional comparison of context representations. The RNN (left surface) compresses all context into a single hidden state trajectory that evolves over time. The transformer (right surface) maintains a separate representation for each position, creating a grid of context vectors.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

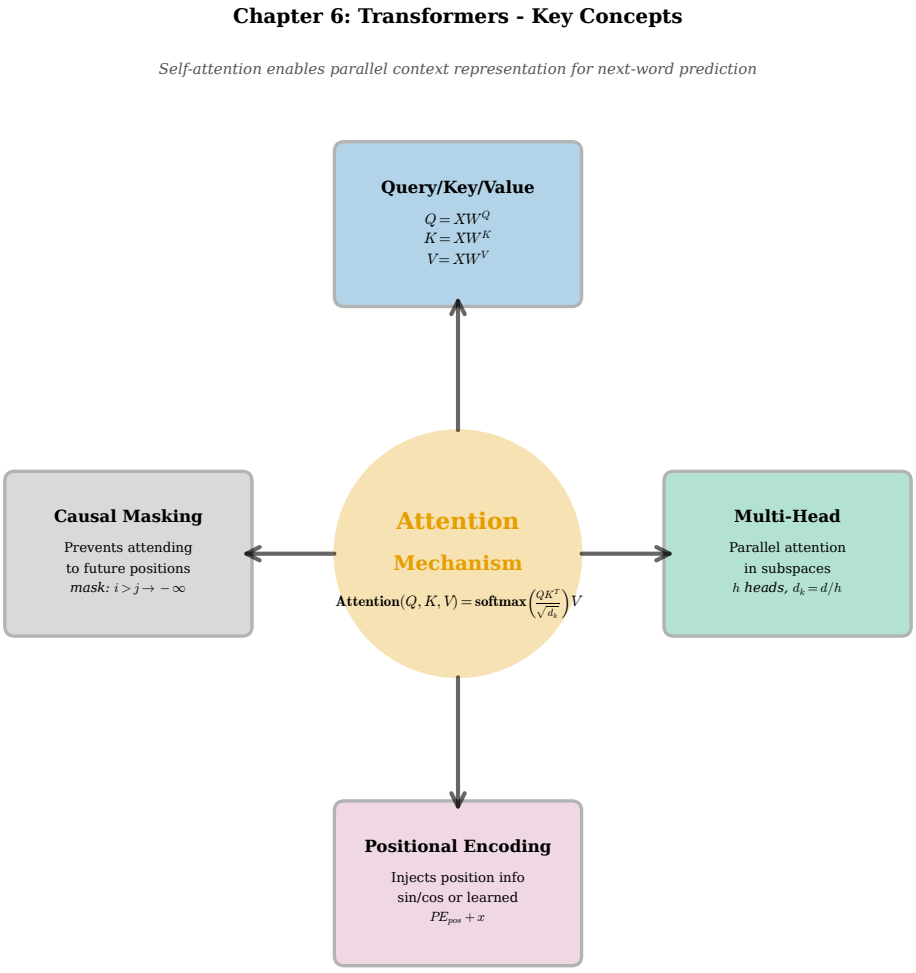


Figure 6.32: Final representation of our running example in a transformer. Each position has a context-enriched representation produced by multiple layers of attention. Position 23 (where we predict the next word) has attended to relevant words like “code”, “engineer”, and “submitted”, integrating their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

Predicting the Next Word

From Shannon to ChatGPT

Test Compilation - Chapter 7

Contents

Chapter 7

Decoding Strategies: From Probabilities to Text

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

- Converting probability distributions to actual text sequences
- Balancing quality and diversity in generated output
- Handling the search space explosion of autoregressive decoding
- Adapting generation strategies to different task requirements

7.1 The Decoding Problem

The transformer architecture developed in Chapter **??** produces a probability distribution $P(w)$ over the vocabulary $\mathscr{V}$ at each position, but language models ultimately must generate discrete text sequences that humans can read and understand. This transformation from continuous probability distributions to discrete token sequences is the fundamental decoding problem that defines how models actually produce useful output in practice. The challenge arises because autoregressive language models generate one token at a time in a sequential process: after predicting and selecting token $w[t]$, this choice becomes an immutable part of the context for predicting $w[t+1]$, creating a cascade of dependent decisions where early mistakes propagate through and contaminate the entire remaining sequence. With a vocabulary size of approximately 50,000 tokens and a target sequence length of 100 tokens, the space of possible outputs grows exponentially to 50000^{100} distinct sequences, a number so astronomically large that it exceeds comprehension and makes exhaustive search computationally impossible by any conceivable technology. The decoding strategy we choose determines not only the quality and coherence of individual outputs but also the diversity of generations across multiple samples from the same prompt, influencing whether the model produces creative variety or repetitive sameness.

Consider our running example: when asked to continue "Once upon a time in a distant kingdom, there lived a young," the model computes a conditional probability distribution $P(w\,|\,\text{context})$ over all possible next words given everything that came before. Figure 7.1 shows this distribution, with "prince" having probability 0.15, "girl" at 0.12, "wizard" at 0.09, and so on through thousands of vocabulary items with monotonically decreasing probability that eventually approaches zero for rare or contextually inappropriate tokens. The decoding problem asks: how should we select from this rich probability distribution to produce compelling text? Should we always pick the most probable word using greedy decoding, should we sample according to the exact probabilities through pure stochastic sampling, or should we use some intermediate strategy that carefully balances exploitation of high-probability tokens with exploration of diverse alternatives? Each choice leads to qualitatively different outputs, from safe but potentially boring and repetitive text to creative but potentially incoherent or nonsensical generations, representing a fundamental trade-off in text generation that has no

universally optimal solution.

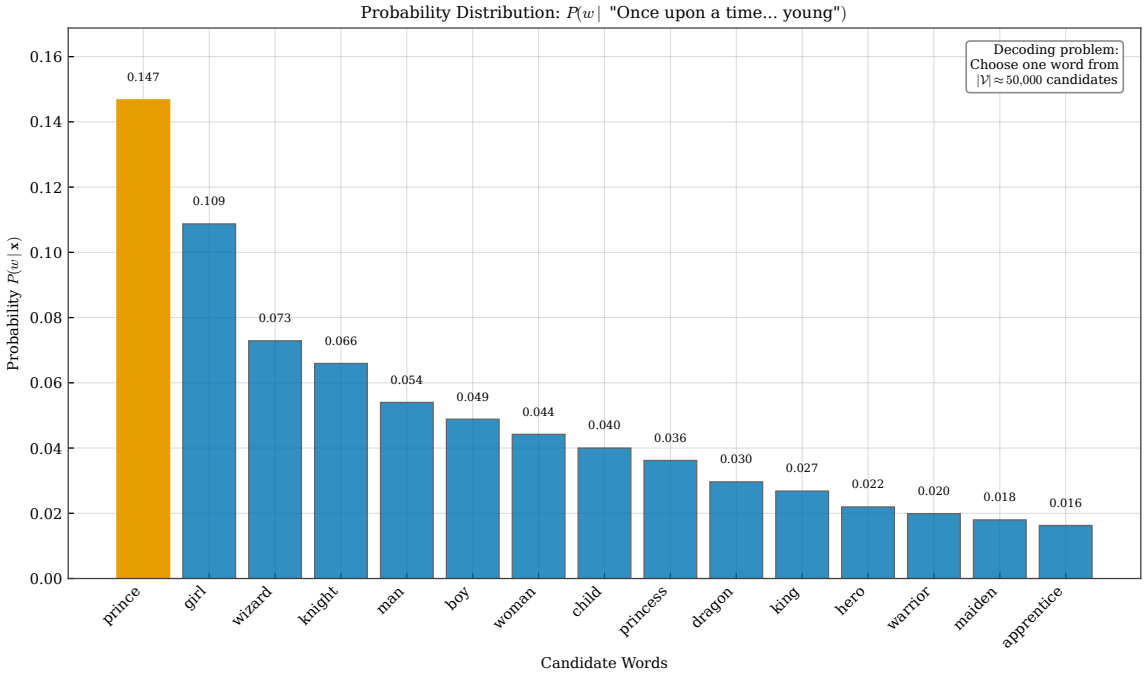


Figure 7.1: Probability distribution over next words after "Once upon a time in a distant kingdom, there lived a young." The model assigns highest probability to "prince" (0.15), followed by "girl" (0.12), "wizard" (0.09), and many other candidates with decreasing probability. Decoding strategies determine how we select from this distribution.

The search space explosion illustrated in Figure 7.2 shows why naive enumeration of all possibilities fails catastrophically. At each decoding step, we face $|\mathcal{V}|$ choices representing every token in our vocabulary, and these choices multiply combinatorially across positions because each decision affects all subsequent distributions. For a vocabulary of 50,000 tokens and sequence length T, the total number of possible sequences is $|\mathcal{V}|^T$, which grows with terrifying speed as sequence length increases. Even for a modest sequence of just 20 tokens, this equals $50000^{20} \approx 10^{94}$, a number that far exceeds not only our computational capabilities but the estimated number of atoms in the observable universe, which is approximately 10^{80}. This combinatorial explosion means we cannot simply enumerate all sequences and pick the best one according to some quality criterion; instead, we must use heuristic strategies that explore only a vanishingly tiny fraction of the search space while hopefully finding outputs that are sufficiently high-quality for our purposes. Different decoding strategies represent fundamentally different trade-offs between computational cost, output quality as measured by coherence and relevance, and output diversity as measured by variation across multiple generations.

Decoding strategies fall into two broad categories: deterministic and stochastic, as shown in Figure 7.3. Deterministic methods like greedy decoding and beam search always produce the same output for a given input, selecting tokens based purely on probability rankings without any randomness whatsoever, making their behavior completely reproducible and predictable. These methods are valuable when consistency matters: running the same prompt twice yields identical results, which simplifies debugging, testing, and deployment of language model systems. Deterministic methods often produce coherent, high-probability sequences that sound grammatical and contextually appropriate, but they can lack diversity and creativity, sometimes falling into repetitive patterns or generating generic, uninspiring text that reads like it was assembled from the most common phrases in the training data. Stochastic methods like temperature sampling, top-k, and nucleus sampling incorporate controlled randomness, sampling tokens according to modified probability distributions that can be tuned to balance exploration and exploitation. These methods can generate diverse, creative outputs by occasionally exploring lower-probability regions of the distribution where surprising but contextually valid tokens reside, but they risk producing incoherent or low-quality text if unlikely tokens are selected at critical junctures where the

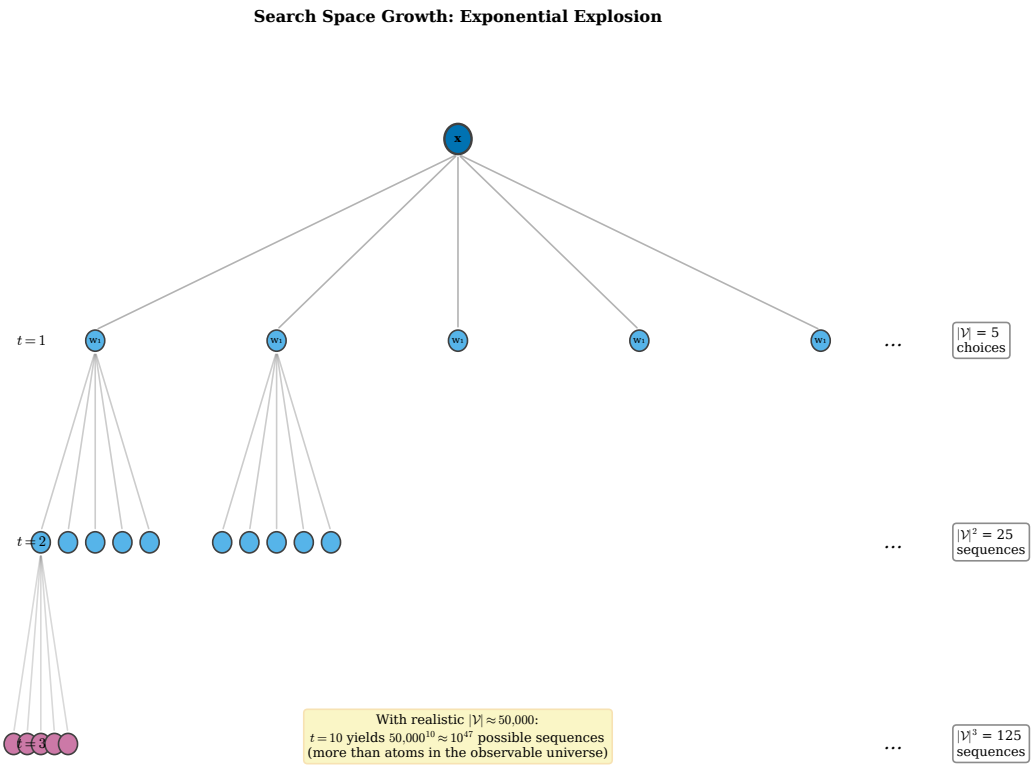


Figure 7.2: The exponential growth of the search space in autoregressive decoding. Each node represents a partial sequence, and each edge represents choosing a next token from the vocabulary $\mathcal{V}$. After just three steps, we have $|\mathcal{V}|^3$ possible sequences. For $|\mathcal{V}| = 50{,}000$ and sequence length 20, the total space contains approximately 10^{94} sequences.

wrong choice derails the entire narrative. Modern practice often combines elements of both approaches, using deterministic pruning to eliminate clearly bad options followed by stochastic sampling among the remaining candidates.

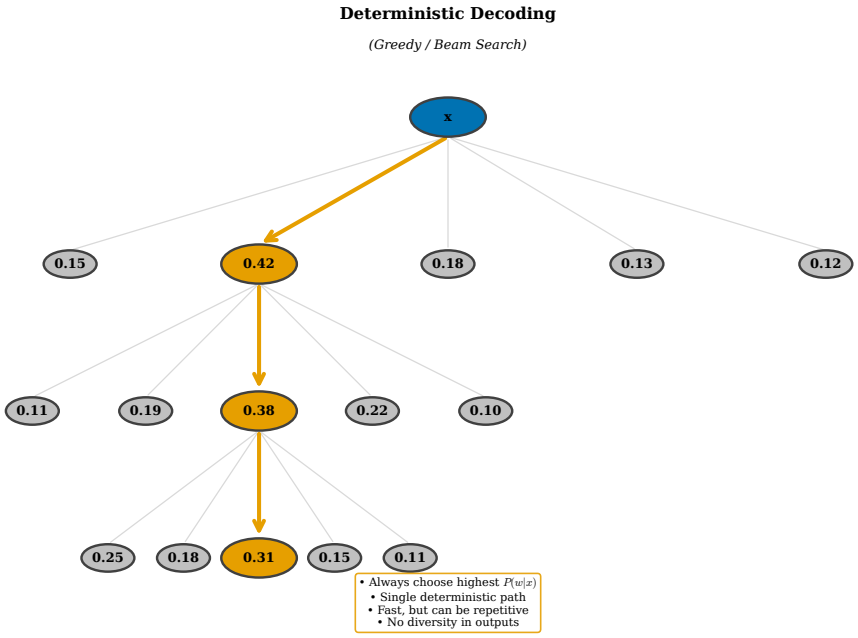


Figure 7.3: Deterministic versus stochastic decoding paradigms. Deterministic methods (left) always follow the same path through the search space, typically selecting highest-probability options. Stochastic methods (right) can explore different paths across multiple runs, potentially discovering more diverse or creative sequences.

7.2 Greedy Decoding and Its Failures

The simplest decoding strategy is greedy decoding, which selects the highest-probability token at each position without considering alternatives: $w[t] = \arg\max_{w \in \mathcal{V}} P(w \mid w[1], \ldots, w[t-1])$. This approach is computationally efficient, requiring only $O(T \cdot |\mathcal{V}|)$ operations to generate a sequence of length T, since we simply take the argmax over vocabulary probabilities at each step without maintaining multiple hypotheses or performing any sampling operations. Greedy decoding produces deterministic output: running it twice on the same prompt yields identical results, which can be advantageous for reproducibility and debugging but limits variety when multiple outputs are desired. The method implicitly assumes that locally optimal choices lead to globally optimal sequences, that picking the best word at each step will produce the best overall text when the sequence is considered as a whole. This assumption, while computationally convenient and intuitively appealing, is fundamentally flawed for language generation, where context dependencies spanning many tokens mean that a slightly less probable word now might enable much more probable and more interesting continuations later, leading to a globally superior sequence despite the locally suboptimal initial choice.

Figure 7.4 illustrates greedy decoding on our running example, showing the decision tree and the single path that greedy decoding selects. Starting with "Once upon a time in a distant kingdom, there lived a young," greedy decoding examines the probability distribution and selects "prince" (probability 0.15 as the highest), then examines the new distribution conditioned on this choice and selects "who" (probability 0.22), then "lived" (probability 0.18), and so on through the generation process. At each step, we commit fully and irrevocably to the highest-probability option without considering alternatives, even when the second-highest option has nearly equal probability and might lead to more interesting continuations. The path through the probability space is entirely deterministic: given the model weights and the input prompt, greedy decoding will always produce exactly the same sequence regardless of when or how many times we run it. The local optimization at each step ignores the global structure of the text and the complex dependencies between distant tokens, potentially missing much better overall sequences that would have required accepting a slightly less probable early choice to unlock superior later options.

The most severe failure mode of greedy decoding is the repetition loop, where the model falls into generat-

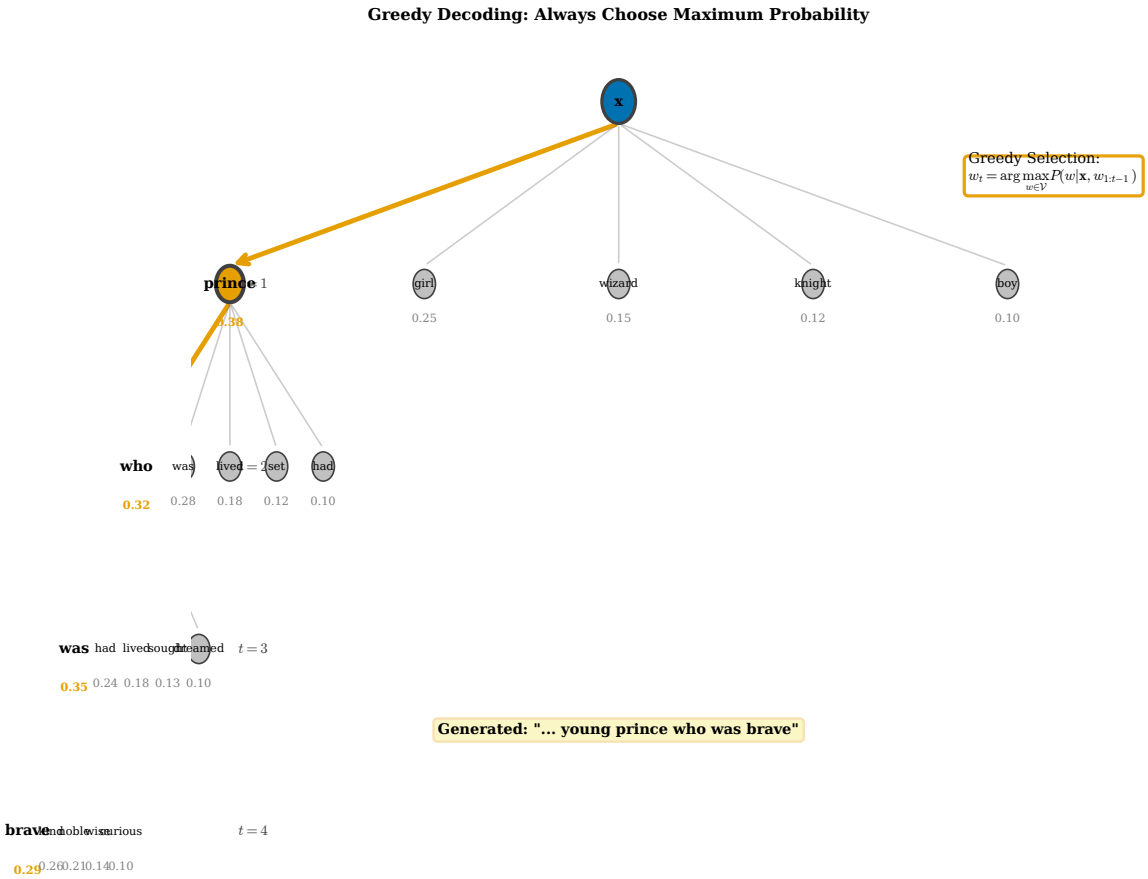


Figure 7.4: Greedy decoding selects the highest-probability token at each step (shown in orange). Alternative tokens with lower probabilities are discarded immediately. The greedy path may not be globally optimal: a different choice at an early step might have enabled higher probabilities later.

ing the same phrase or sentence repeatedly without any mechanism to escape. Figure 7.5 shows this catastrophic failure on our fantasy prompt, where greedy decoding produces: “Once upon a time in a distant kingdom, there lived a young man. The man was a man who was a man who was a man...” This degeneration occurs through a self-reinforcing feedback loop: once the model generates “man,” the phrase “was a man” becomes a high-probability continuation because similar patterns appear frequently in the training data, and after generating “was a man,” the same phrase remains the highest-probability continuation. The greedy strategy follows this local maximum repeatedly, completely unable to escape because doing so would require selecting a lower-probability token. Beyond these catastrophic repetition loops, greedy decoding also produces generic, uninspiring text even when it does not degenerate completely: it gravitates relentlessly toward common, expected continuations rather than surprising or distinctive ones, producing clichéd text filled with overused phrases like “the beautiful princess,” “the brave knight,” and “happily ever after” for creative writing, or safe, bland responses like “I’m doing well, thank you” and “That’s interesting” for dialogue systems. The probability distribution learned by language models reflects the frequency of patterns in their training data, so the highest-probability options are mathematically destined to be the most common and therefore least distinctive tokens, causing greedy decoding to systematically sacrifice diversity and interestingness for a narrow notion of quality that equates probable with good.

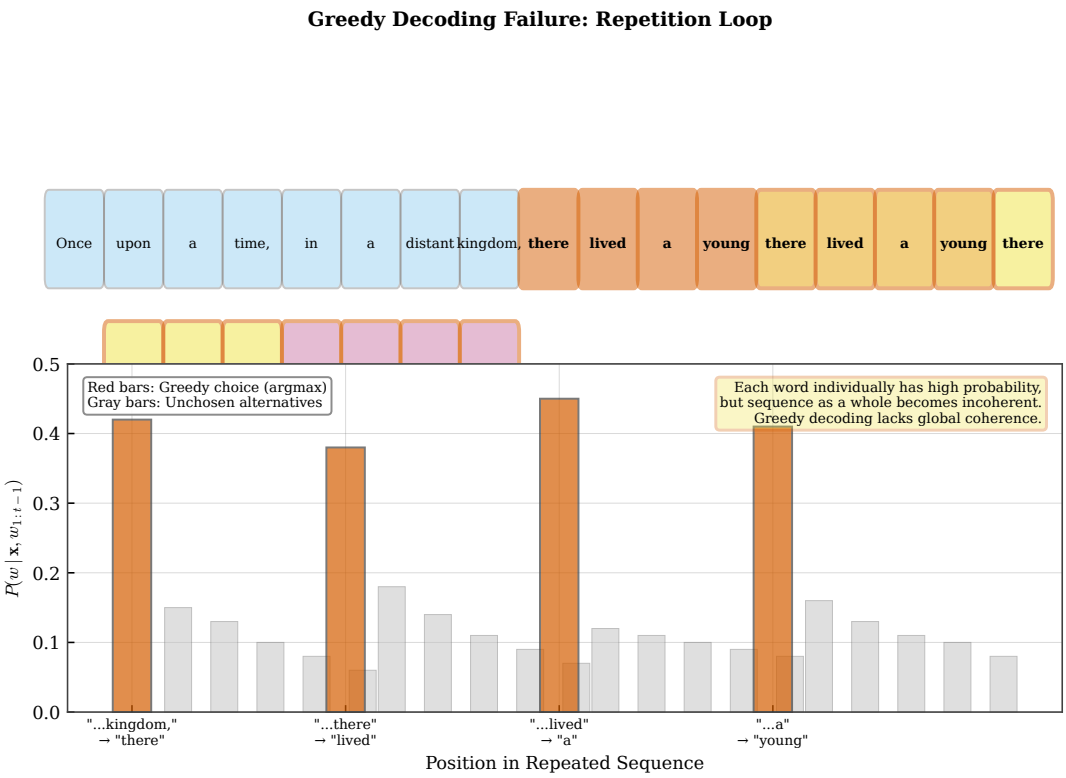


Figure 7.5: Greedy decoding failure: repetition loop. The model falls into repeating "was a man" because this phrase is always the highest-probability continuation of the previous "man." The red highlighting shows where degeneration begins. Greedy decoding cannot escape such loops without external intervention.

7.3 Temperature Scaling

Temperature scaling modifies the probability distribution before sampling by dividing the logits by a temperature parameter τ before applying the softmax function: $P(w) = \frac{\exp(z_w/\tau)}{\sum_{w' \in \mathcal{V}} \exp(z_{w'}/\tau)}$, where z_w is the raw logit (pre-softmax score) for word w. The temperature controls the sharpness or peakedness of the resulting distribution through a simple but powerful mechanism: values less than 1 make the distribution peakier by amplifying differences between logits, while values greater than 1 make it flatter by compressing those differences. At $\tau = 1$, the original distribution is preserved exactly as the model produced it. As $\tau \to 0$, the distribution approaches a one-hot vector concentrated entirely on the highest-probability token, mathematically equivalent to greedy decoding. As $\tau \to \infty$, the distribution approaches uniform over all vocabulary items, giving every token equal probability regardless of the model's preferences. Temperature scaling is motivated by the observation that greedy decoding produces boring text precisely because it always selects the mode: by flattening the distribution through increased temperature, we allow stochastic sampling to explore alternative tokens while still maintaining a preference for more probable options over less probable ones.

Figure 7.6 shows how different temperature values dramatically affect our probability distribution for the next word after "young," illustrating the trade-off between peaked and flat distributions. At $\tau = 0.3$, nearly all probability mass concentrates sharply on "prince," making all other options almost impossible to sample and effectively reducing sampling to greedy selection. At $\tau = 1.0$, the original distribution shows a clear preference for "prince" at 15% but gives meaningful probability to alternatives like "girl" at 12% and "wizard" at 9%, creating genuine randomness while respecting the model's learned preferences. At $\tau = 2.0$, the distribution is much flatter with many tokens having similar probabilities, allowing creative but potentially incoherent choices

because even low-probability tokens now have reasonable sampling chances. The three-dimensional surface in Figure 7.7 visualizes how token probabilities vary jointly with temperature and token rank across the entire distribution: at low temperatures, the surface shows a dramatic sharp peak at the first-ranked token with probabilities dropping precipitously for lower-ranked tokens, while as temperature increases this peak gradually flattens and spreads outward, with probability mass redistributing from the top tokens to the middle and tail, compressing the differences so that the 10th-ranked token becomes nearly as likely as the 2nd-ranked token. This global reshaping explains why temperature simultaneously affects both the expected quality of generated text (by changing which tokens are likely to be sampled) and its diversity (by changing how much variation occurs across multiple samples).

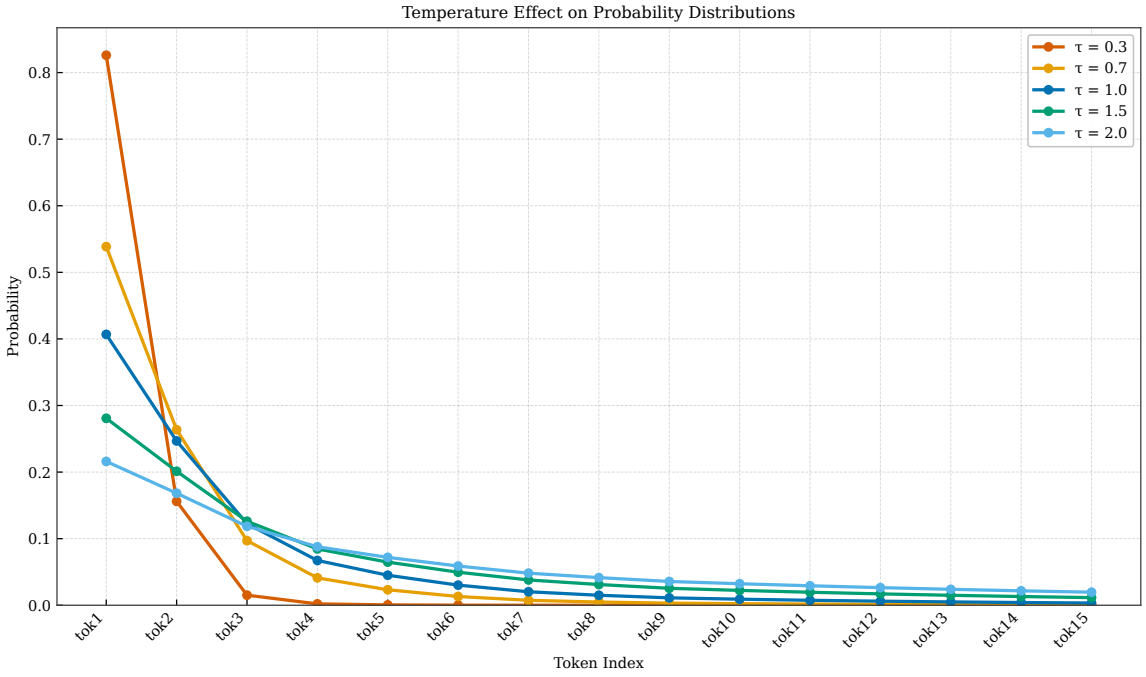


Figure 7.6: Effect of temperature on probability distributions. At $\tau = 0.3$ (red), probability concentrates sharply on the highest-probability token. At $\tau = 1.0$ (blue), the original distribution is preserved. At $\tau = 2.0$ (green), the distribution flattens significantly, giving many tokens similar probability.

Figure 7.8 quantifies the fundamental relationship between temperature and entropy using information-theoretic measures. Entropy, defined as $H(P) = -\sum_{w \in \mathcal{V}} P(w) \log P(w)$, measures the uncertainty or randomness inherent in the distribution—higher entropy means more unpredictability in which token will be sampled. At $\tau \to 0$, entropy approaches zero because all probability mass concentrates on a single token (the argmax), making the sampling outcome completely predictable and deterministic with no uncertainty whatsoever. As temperature increases, entropy increases monotonically because probability mass spreads across more tokens, each contributing to the overall uncertainty. The maximum possible entropy (achieved as $\tau \to \infty$) equals $\log |\mathcal{V}|$, corresponding to a uniform distribution where all tokens are equally likely and each sample is maximally unpredictable. Practitioners often think of temperature as controlling "creativity" or "randomness": higher entropy means more variation in sampling outcomes, which can produce more surprising and original outputs but also increases the risk of sampling contextually inappropriate tokens that damage coherence. The entropy curve provides a principled way to understand and select temperature values based on the desired level of generation variability. However, temperature scaling alone does not solve all the problems of greedy decoding, despite its intuitive appeal and widespread use, because it does not prevent sampling very low-probability tokens that might produce nonsensical, ungrammatical, or contextually inappropriate output—at $\tau = 1.5$, a token with original probability 0.001 might become probable enough to sample occasionally, and such rare tokens often represent typographical errors, proper nouns that do not fit the context, or grammatically invalid continuations. A single such problematic token sampled at a critical position can derail an entire generation, making the output

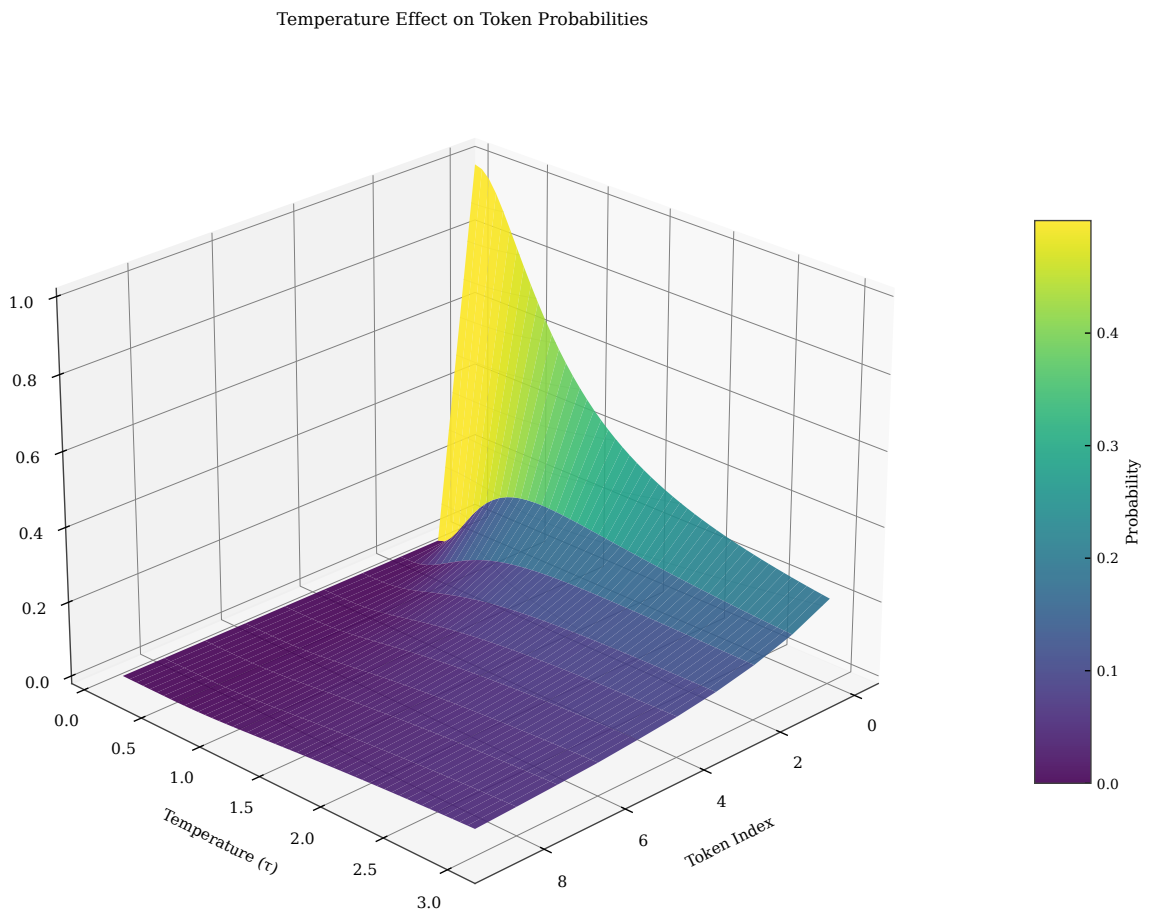


Figure 7.7: Three-dimensional surface showing probability as a function of token rank and temperature. At low temperatures (front), probability concentrates sharply on top-ranked tokens. At high temperatures (back), probability spreads more uniformly across ranks. The surface shape reveals how temperature systematically reshapes the distribution.

unusable despite all other tokens being reasonable, which motivates truncation strategies like top-k and nucleus sampling that combine temperature-like smoothing with hard cutoffs preventing sampling from the extreme tail.

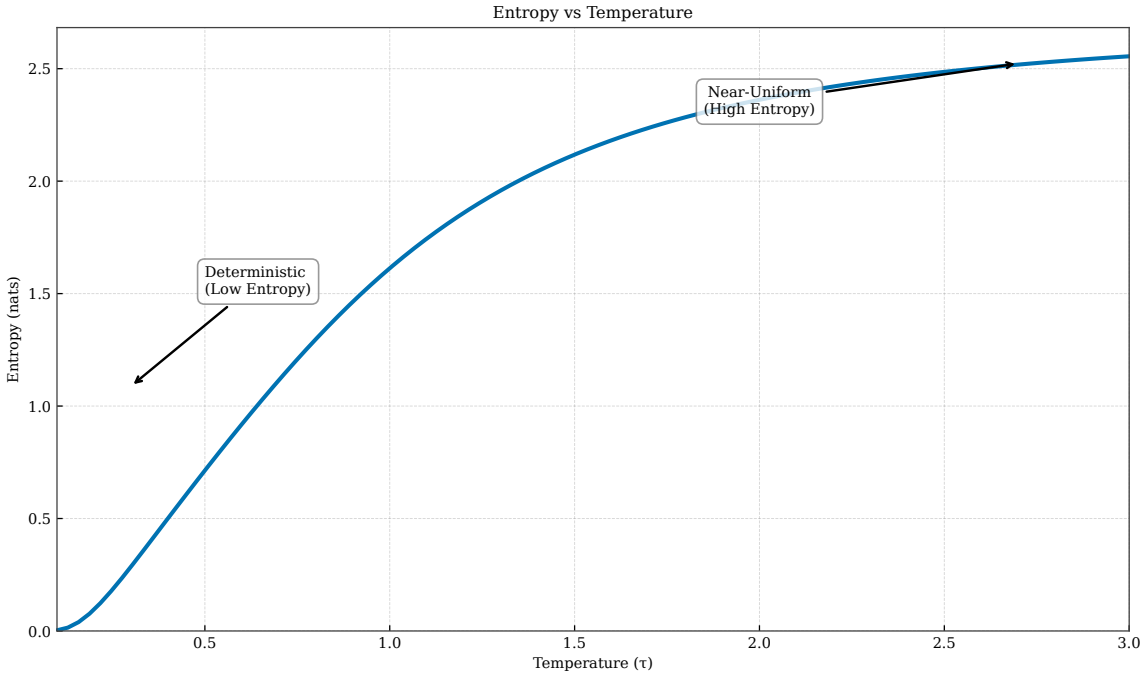


Figure 7.8: Entropy as a function of temperature. Entropy increases monotonically from near-zero (deterministic, equivalent to greedy decoding) to the maximum $\log|\mathcal{V}|$ (uniform distribution). Temperature controls the trade-off between predictable, safe outputs and diverse, creative outputs.

7.4 Top-k Sampling

Top-k sampling addresses the tail risk of pure temperature sampling by truncating the distribution to the k most probable tokens before sampling, providing a simple but effective guard against sampling contextually inappropriate tokens. Given the probability distribution $P(w)$, we first identify the k tokens with highest probability by sorting the distribution, set the probabilities of all other tokens (those ranked below k) to exactly zero, and then renormalize the remaining probabilities to sum to one so we have a valid probability distribution. Mathematically, if V_k denotes the set of k most probable tokens, the modified distribution is $P(w)' = P(w)/\sum_{w' \in V_k} P(w')$ for $w \in V_k$ and $P(w)' = 0$ otherwise. This hard truncation prevents sampling from the long tail of low-probability tokens that might produce nonsensical or jarring text, while still allowing meaningful diversity among the top candidates that the model considers contextually appropriate. The parameter k controls how much diversity is permitted: small k restricts choices severely and approaches greedy decoding at $k = 1$, while large k allows more exploration of the distribution but with greater risk of occasionally sampling mediocre options.

The three-dimensional visualization in Figure 7.9 shows how top-k truncation reshapes the probability landscape across different values of k, revealing the interaction between rank and truncation threshold. For $k = 5$, only the five highest-probability tokens receive nonzero probability after truncation, with all others masked to exactly zero and shown as a gray plane at the bottom representing the excluded portion of the vocabulary. As k increases progressively to 10, 20, and 50, more tokens become available for sampling, gradually restoring portions of the original distribution's shape. The surface reveals that truncation has two distinct effects: it completely eliminates tokens beyond rank k, preventing any possibility of sampling them, and it amplifies the relative probabilities of remaining tokens through renormalization so that a token originally at 10% of total probability mass becomes a substantially larger share when competing only against other top-ranked tokens. Top-k sampling was popularized by Fan et al. [2018] for story generation, where experiments found that k values between 10 and 50 produced good results balancing coherence with creativity, though the optimal k depends heavily on the model architecture, domain, and task requirements, with smaller models potentially needing smaller k to avoid sampling mediocre options.

Figure 7.10 compares the original untruncated distribution with the $k = 5$ filtered distribution on our run-

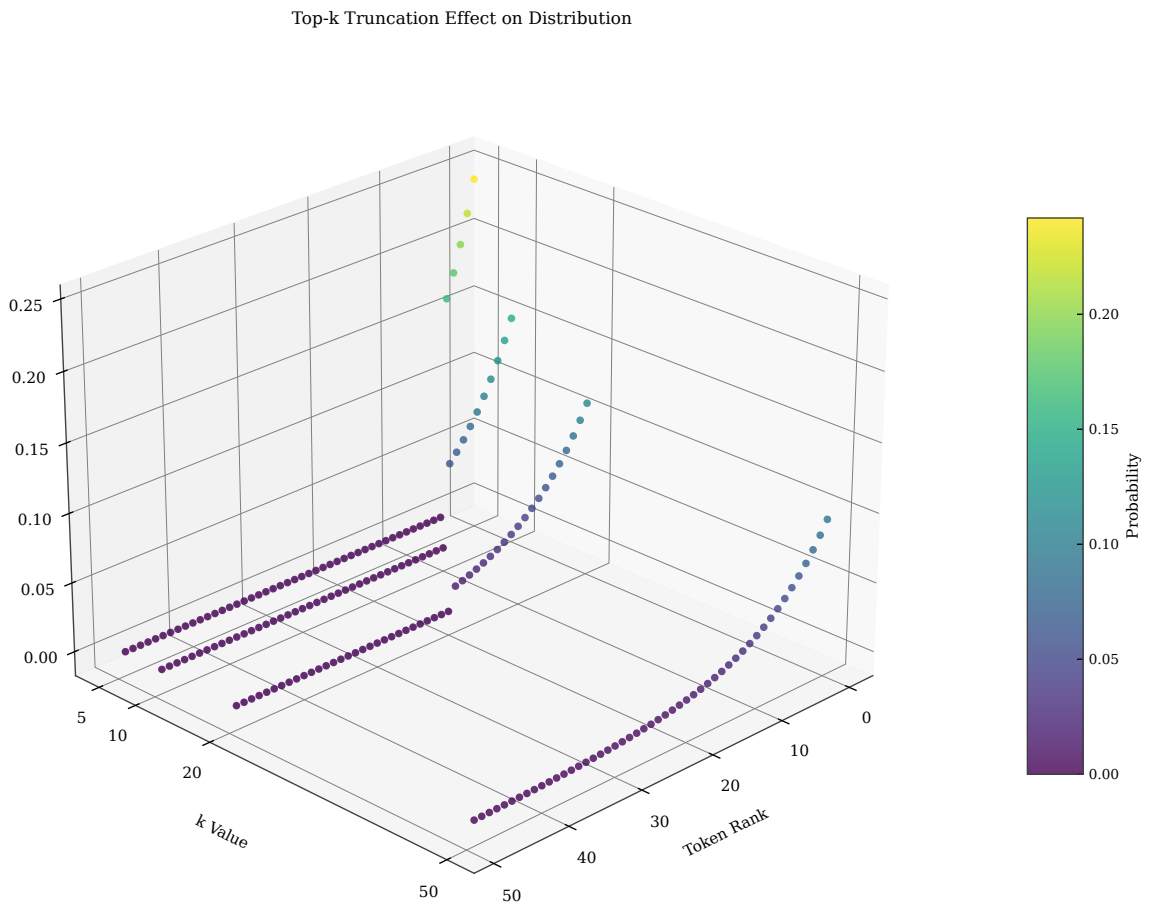


Figure 7.9: Three-dimensional visualization of top-k truncation. The surface shows probability as a function of token rank and k value. Tokens beyond rank k are masked (gray region at zero probability). Remaining tokens have their probabilities amplified through renormalization.

ning fantasy example, illustrating concretely what information is preserved and what is discarded. The original distribution assigns nonzero probability to many tokens across the vocabulary, including rare words like "cobbler," "turnip," and various grammatically awkward options that would sound strange continuing "there lived a young." After top-5 filtering, only "prince," "girl," "wizard," "knight," and "man" remain as candidates for sampling, with their probabilities renormalized to sum to one—"prince" now has approximately 30% probability rather than 15%. The filtering eliminates tail risk entirely: we cannot accidentally sample an unlikely word that would derail the narrative or break the fantasy genre conventions. However, this safety comes at the cost of diversity: if the context genuinely supports an unusual but contextually perfect word at position 6 or beyond in the original ranking, top-5 sampling will never select it regardless of how appropriate it might be. This reveals the fundamental limitation of top-k sampling: the fixed cutoff k ignores the shape of the probability distribution, applying the same rigid truncation regardless of whether the model is confident or uncertain. When the model is highly confident with probability concentrated in just a few tokens, $k = 50$ might wastefully include many nearly-zero-probability tokens that clutter the candidate set without providing meaningful diversity; when the model is genuinely uncertain with probability spread uniformly across many reasonable tokens, $k = 50$ might aggressively exclude perfectly valid options that should legitimately be considered.

Figure 7.11 illustrates the fixed-k problem with concrete examples that demonstrate why no single k value works well across all distribution shapes. For a peaked distribution where the top token has 80% probability, $k = 50$ wastefully includes 40+ tail tokens with negligible individual probability, cluttering the candidate set without adding meaningful diversity and slightly increasing the chance of sampling a mediocre option. For a flat distribution where many tokens share similar probability around 4-5% each, $k = 5$ aggressively excludes tokens

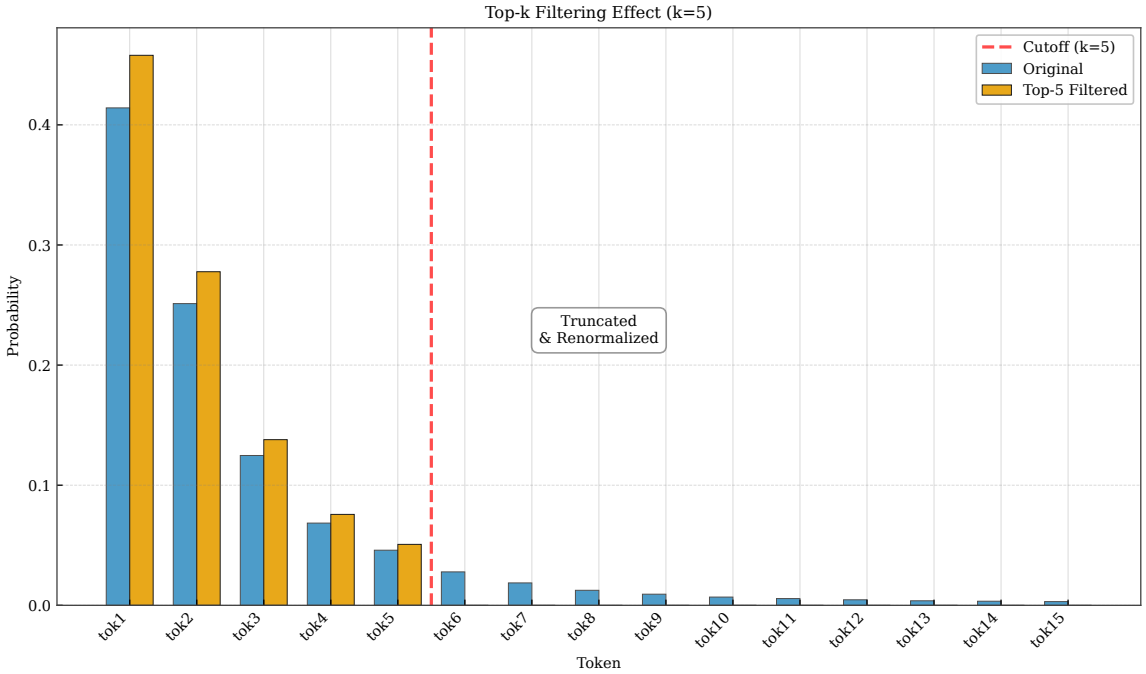


Figure 7.10: Comparison of original distribution (blue) with top-5 filtered distribution (orange). The vertical line marks the truncation point. Tokens beyond rank 5 are eliminated entirely, and remaining tokens have amplified probabilities after renormalization.

that are nearly as probable as those included, arbitrarily cutting off valid options that the model considers almost equally appropriate. The ideal cutoff depends on the distribution shape, which varies dramatically from position to position based on context, making any fixed k suboptimal for at least some positions during generation.

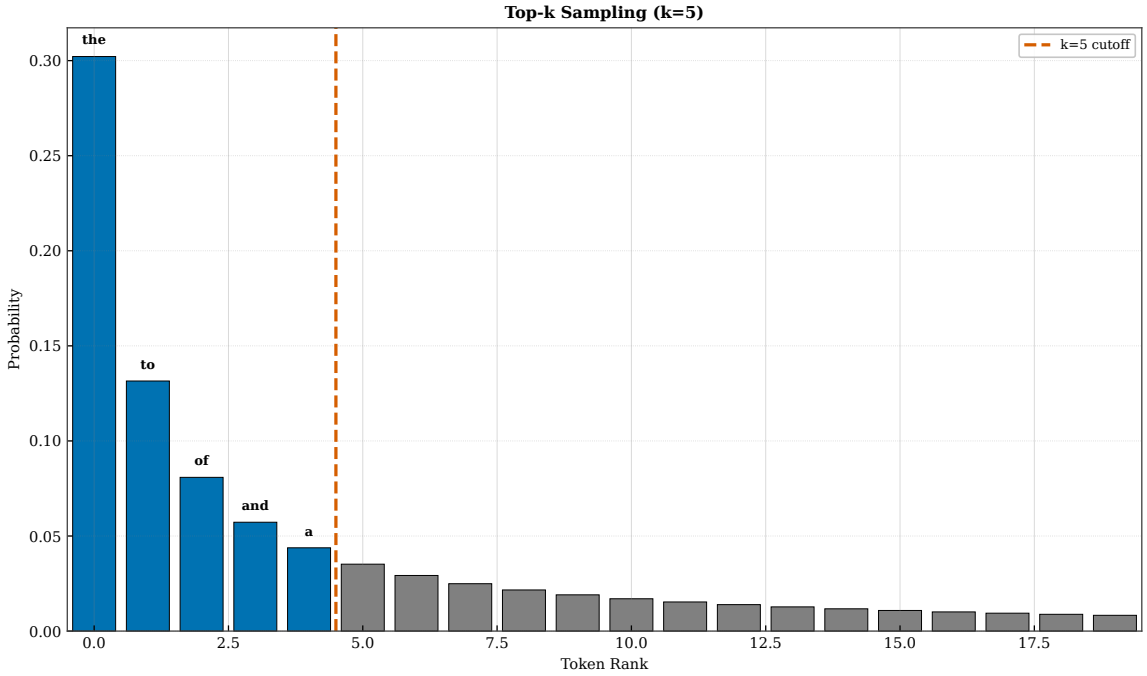


Figure 7.11: The fixed-k problem: the same k value behaves differently on peaked versus flat distributions. Left: for a peaked distribution, $k = 50$ includes many near-zero-probability tokens. Right: for a flat distribution, $k = 5$ excludes tokens nearly as probable as those included. Optimal k depends on distribution shape.

7.5 Nucleus (Top-p) Sampling

Nucleus sampling, also called top-p sampling, addresses the fixed-cutoff problem of top-k by defining the truncation in terms of cumulative probability rather than rank, allowing the size of the candidate set to adapt automatically to the distribution shape [Holtzman et al., 2020]. Instead of keeping a fixed number of tokens regardless of their probabilities, we keep the smallest set of tokens whose cumulative probability exceeds a threshold p, meaning we include exactly enough tokens to capture at least p fraction of the probability mass. Formally, we sort tokens by decreasing probability and include tokens until their cumulative probability first exceeds p: $V_p = \arg\min_{V \subseteq \mathcal{V}} |V|$ such that $\sum_{w \in V} P(w) \geq p$. This adaptive truncation means that for peaked distributions where few tokens dominate, the nucleus contains few tokens (perhaps just 3-5), while for flat distributions where probability is spread widely, the nucleus expands to include many tokens (perhaps 50-100), automatically adjusting to the model's confidence level at each position without requiring manual tuning of a rank-based cutoff.

The three-dimensional surface in Figure 7.12 visualizes how the nucleus boundary varies jointly with the probability threshold p and token rank, showing the adaptive nature of the truncation across the parameter space. At low p (e.g., 0.7), the nucleus typically contains only the very top tokens because 70% of probability mass is often captured by just a handful of tokens, while at high p (e.g., 0.95), the nucleus extends further into the tail to capture nearly all the probability mass while still excluding the extreme tail where problematic tokens reside. The surface shows that the nucleus boundary is not a simple function of rank like top-k but depends on the cumulative probability structure of each specific distribution: positions where the model is confident have sharp cutoffs at low ranks because few tokens capture most probability, while positions where the model is uncertain have gradual cutoffs extending to higher ranks because probability is distributed more evenly and more tokens are needed to reach the threshold.

Figure 7.13 demonstrates concretely how nucleus sampling adapts to different distribution shapes, contrasting its behavior on peaked versus flat distributions. In the left panel, the model is highly confident with probability concentrated in just 3 tokens that together account for 90% of the probability mass; nucleus sampling with $p = 0.9$ includes exactly these 3 tokens, avoiding the wasteful inclusion of dozens of tail tokens

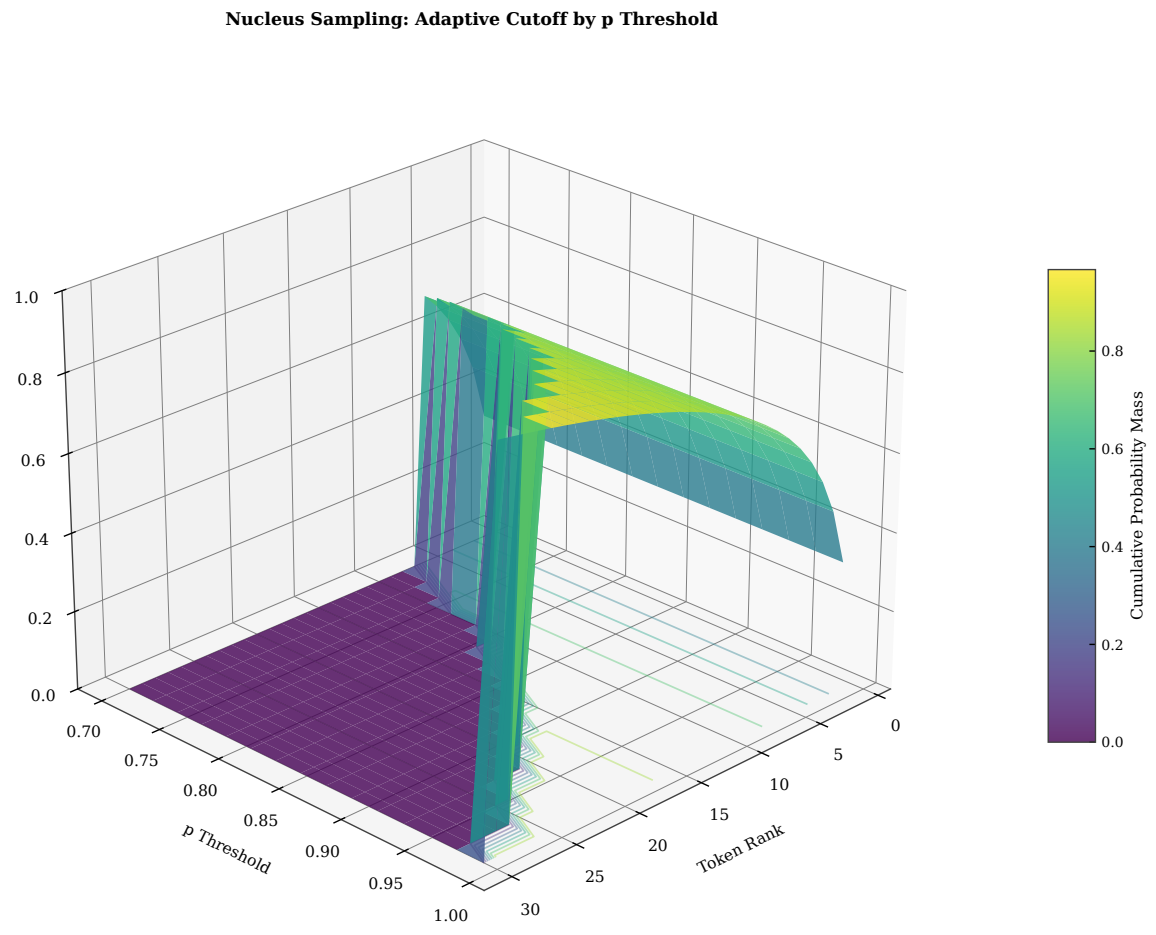


Figure 7.12: Three-dimensional surface showing nucleus membership as a function of token rank and probability threshold p. Higher p values include more tokens in the nucleus. The boundary adapts to the probability distribution: peaked distributions have sharp cutoffs, while flat distributions have gradual cutoffs.

that would occur with a fixed $k = 50$. In the right panel, the model is genuinely uncertain with probability spread across 20 tokens each around 4.5%; nucleus sampling includes all 20 tokens, capturing this genuine uncertainty rather than arbitrarily truncating at $k = 5$. This adaptivity is the key advantage of nucleus sampling over top-k: it respects the model's varying confidence level across positions. The cumulative probability view in Figure 7.14 provides the most direct visualization of how the threshold p determines the nucleus boundary: tokens are sorted by probability from highest to lowest, and the first position where cumulative probability exceeds p marks the boundary. Higher p means including more tokens to capture more probability mass, while lower p means fewer tokens; the step-function shape of the cumulative curve also reveals how peaked distributions have steep early steps and reach 90% quickly, while flat distributions have gradual steps requiring many more tokens.

Nucleus sampling can be productively combined with temperature scaling, applying both techniques in sequence: first adjust the distribution with temperature to control overall sharpness, then apply nucleus truncation to the temperature-scaled distribution to control tail risk. The combined approach gives practitioners two independent knobs to control different aspects of generation behavior: temperature controls the overall entropy and how peaked versus flat the distribution is, while the nucleus threshold controls tail truncation and prevents sampling from the extreme low-probability region regardless of temperature. Common settings in practice include $p = 0.9$ or $p = 0.95$ for the nucleus threshold combined with temperature around 0.7 to 1.0, though optimal values depend on the model and task. The Holtzman et al. [2020] paper that introduced nucleus sampling demonstrated its advantages over top-k for open-ended text generation tasks, showing through both automated

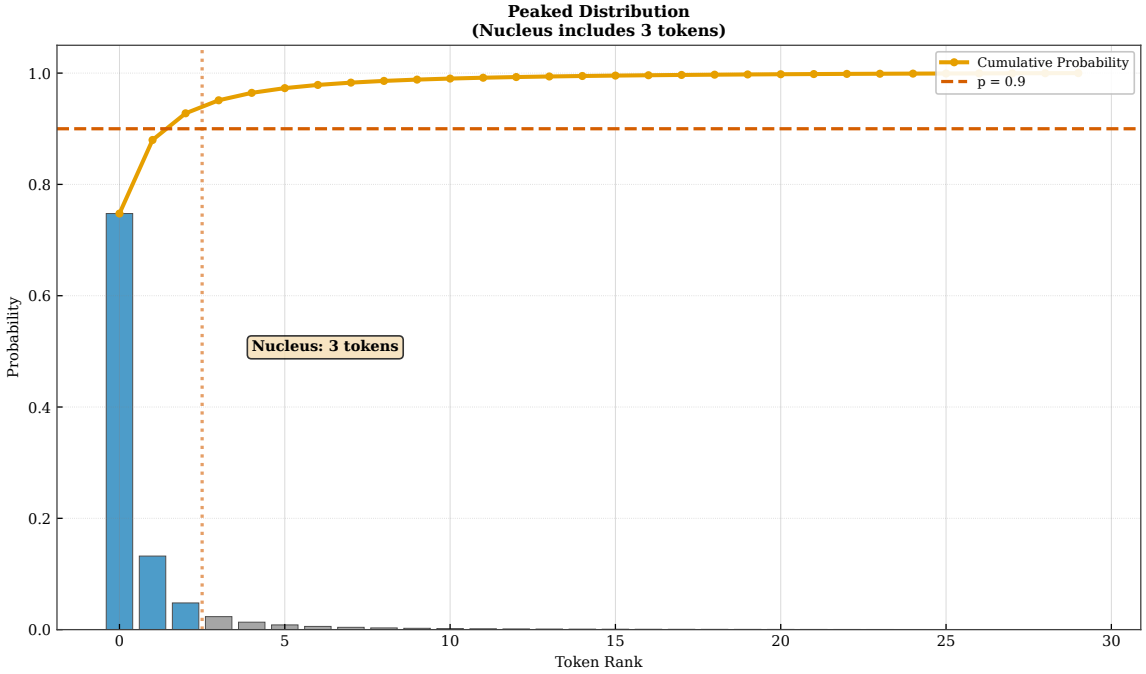


Figure 7.13: Nucleus sampling adapts to distribution shape. Left: peaked distribution, nucleus contains 3 tokens. Right: flat distribution, nucleus contains 20 tokens. Both use $p = 0.9$, but the resulting set size differs based on how probability mass is distributed.

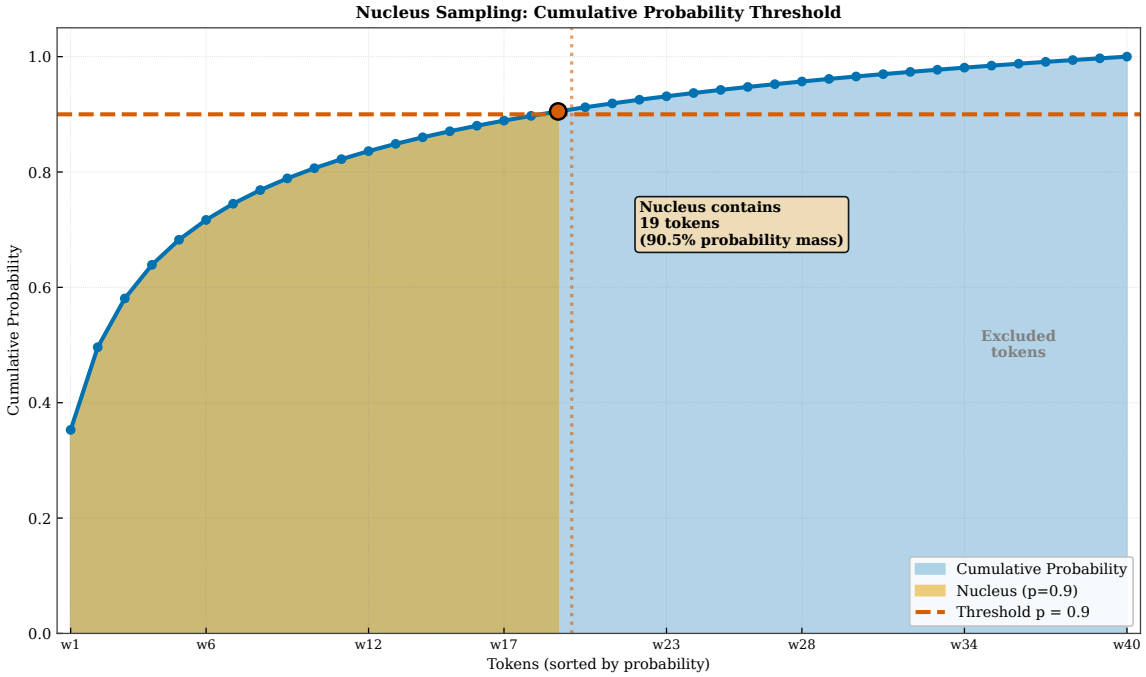


Figure 7.14: Cumulative probability view of nucleus sampling. Tokens are sorted by probability (x-axis) and we plot cumulative probability (y-axis). The nucleus includes all tokens up to where cumulative probability first exceeds p (horizontal line). The shaded region represents tokens in the nucleus.

metrics and human evaluation that nucleus sampling produces more human-like text with fewer degenerate repetition patterns than alternative methods.

7.6 Typical Sampling

Typical sampling takes an information-theoretic approach to the truncation problem, motivated by the observation that humans tend to produce text that is neither too predictable nor too surprising—natural language occupies a middle ground of "typical" information content [Meister et al., 2023]. Rather than including tokens based on their probability rank (as in top-k) or cumulative probability (as in nucleus sampling), typical sampling includes tokens whose information content is close to the expected information content (entropy) of the distribution. The information content of a token is defined as $-\log P(w)$, which is low for high-probability tokens (predictable, few bits of surprise) and high for low-probability tokens (surprising, many bits of information). Typical sampling keeps tokens whose information content falls within a specified range around the entropy $H(P) = \mathbb{E}_{[}P[-\log P(w)]$, excluding both tokens that are too predictable (very high probability, information content below the typical range) and tokens that are too surprising (very low probability, information content above the typical range). This two-sided filtering based on typicality rather than raw probability represents a fundamentally different philosophy of what makes a good token choice.

Figure 7.15 visualizes the typical set in terms of token probability and information content, showing which tokens are included and excluded by the typical sampling criterion. Each point in the visualization represents a token, with position on the x-axis showing its probability and position on the y-axis showing its information content ($-\log p$). Note that these quantities are inversely related: high probability corresponds to low information content, and vice versa. The horizontal band around the entropy value marks the typical set: tokens falling within this band are considered for sampling, while tokens outside are excluded. Critically, note that the typical set excludes both extremes: very high-probability tokens (on the left side of the plot, with low information content) that would make text too predictable and boring, and very low-probability tokens (on the right side, with high information content) that would make text too surprising and potentially incoherent. This two-sided filtering fundamentally distinguishes typical sampling from top-k and nucleus sampling, which only filter the low-probability tail and always include the highest-probability token.

Figure 7.16 compares typical sampling with nucleus sampling on a concrete example distribution, highlighting the different tokens each method selects and the philosophical difference in their approaches. For a given distribution, nucleus sampling (shown in the left panel) includes all tokens above a cumulative probability threshold, which by construction necessarily includes the highest-probability token since cumulative probability starts with that token. Typical sampling (shown in the right panel) may exclude the highest-probability token because its information content is below the typical range—the method considers it "too obvious" a choice that would make the text predictable. The overlap between the two methods, shown in the center, contains tokens that are both highly probable (included by nucleus) and informationally typical (included by typical sampling). Tokens in the nucleus set but not the typical set are "too predictable" by the typicality criterion; tokens in the typical set but not the nucleus are "too surprising" for nucleus sampling's cumulative threshold but informationally appropriate according to the typical sampling philosophy.

Typical sampling is motivated by rate-distortion theory and the concept of typical sequences in information theory, connecting text generation to fundamental principles established by Claude Shannon. In a long sequence of independent samples from a distribution, the law of large numbers implies that the average information content of the sequence converges to the entropy of the distribution. Sequences whose average information content deviates significantly from the entropy are exponentially unlikely to occur—they are "atypical" in a precise mathematical sense. This theoretical insight suggests that human-like text should have information content close to what the language model expects on average, neither too high (overly surprising, containing too much information) nor too low (overly predictable, containing too little information). While typical sampling is theoretically elegant and has strong mathematical foundations, it has seen less widespread adoption than nucleus sampling in practical applications, partly because the information-theoretic motivation is less intuitive to practitioners and partly because the exclusion of high-probability tokens can sometimes hurt coherence in domains where the obvious continuation really is the best one.

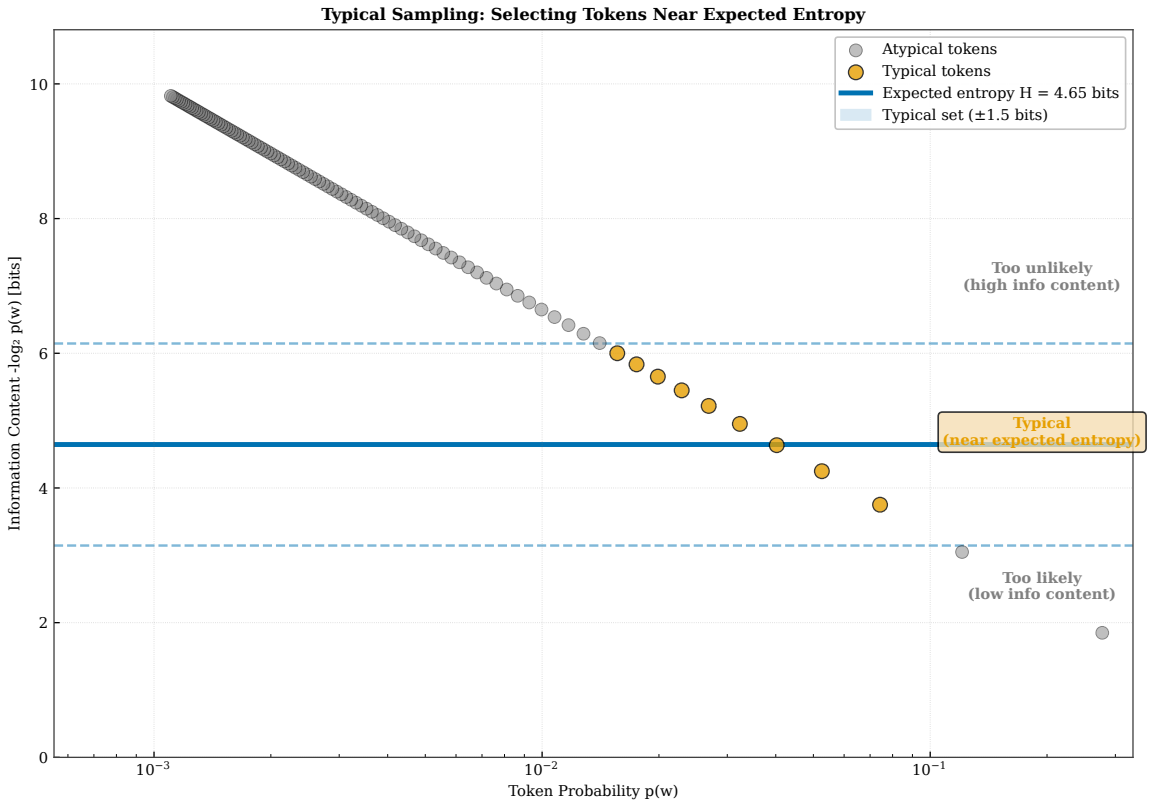


Figure 7.15: The typical set in typical sampling. X-axis shows token probability; y-axis shows information content ($-\log p$). The horizontal band around the entropy marks the typical set. Unlike nucleus sampling, typical sampling excludes both very likely tokens (too predictable) and very unlikely tokens (too surprising).

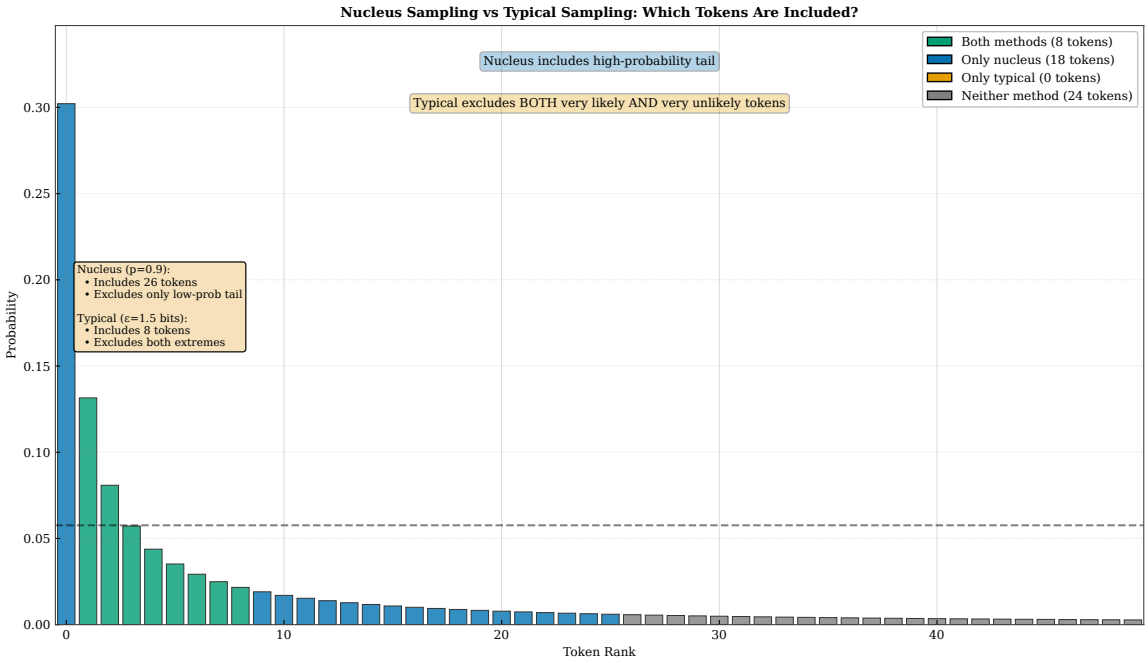


Figure 7.16: Comparison of nucleus and typical sampling. Nucleus sampling includes all tokens above a cumulative probability threshold. Typical sampling includes tokens with information content near the expected value, potentially excluding both very likely and very unlikely tokens.

7.7 Beam Search

While sampling methods generate diverse outputs by incorporating controlled randomness, beam search takes an entirely deterministic approach to finding high-probability sequences by maintaining multiple hypotheses simultaneously throughout the generation process. Beam search keeps the B most probable partial sequences (called "beams") at each step, expanding each beam with all possible next tokens to generate candidate continuations and then pruning back to the top B candidates based on cumulative probability. Unlike greedy decoding, which commits irrevocably to a single choice at each step, beam search explores B alternative paths through the search space simultaneously, allowing recovery from locally suboptimal choices if a different path leads to a higher-scoring sequence overall when considered as a complete unit. This hedging against early mistakes makes beam search particularly valuable for structured generation tasks like machine translation and summarization, where output quality is measured by metrics like BLEU that reward globally coherent sequences rather than locally optimal individual token choices, and where the correct translation of one word often depends on choices made later in the sentence.

The three-dimensional tree in Figure 7.17 visualizes beam search with $B = 3$ on our running fantasy example, showing how the algorithm maintains and updates multiple hypotheses over time. The x-axis represents time steps (token positions), the y-axis represents beam index (which of the 3 hypotheses we are examining), and the z-axis represents cumulative log-probability (the score used to rank beams). At each step, all three beams are expanded by considering all possible next tokens from the vocabulary, generating $3 \times |\mathscr{V}|$ candidate continuations, then pruned back to the three highest-scoring partial sequences based on their total log-probability. Lines connecting nodes show which beams survive to the next step; the vast majority of expansions are pruned away as lower-scoring than the survivors. The final output is typically the beam with the highest cumulative score after reaching the end-of-sequence token or a maximum length limit, though sometimes the top-k final beams are all returned for downstream reranking.

Figure 7.18 shows a two-dimensional view of beam search paths that emphasizes the token sequences and their scores, making it easier to see which words are generated and why some paths survive while others are pruned. Each path represents a different beam (hypothesis), with nodes labeled by the tokens generated at each position and edges weighted by the log-probabilities of those token choices. The cumulative score of a path is simply the sum of log-probabilities along its edges, which represents the log-probability of the entire partial sequence under the model—higher scores mean more probable sequences. At each step t, beam search keeps the B paths with highest cumulative score, discarding all others regardless of how promising they might have seemed earlier. The final step shows the three surviving beams, with the highest-scoring beam typically selected as the output. The figure illustrates a key property of beam search: it can recover from a seemingly suboptimal choice at step 1 by finding that a different choice at step 2 or 3 leads to higher overall probability, justifying the earlier deviation from the greedy path.

A critical issue with naive beam search is that it systematically favors shorter sequences, often terminating prematurely with incomplete outputs. This bias arises because we accumulate log-probabilities (which are negative since probabilities are between 0 and 1), so longer sequences necessarily have lower (more negative) cumulative scores simply by having more terms in the sum, regardless of whether the individual terms represent good choices. This length bias is addressed by length normalization [Wu et al., 2016]: instead of ranking beams by raw log-probability $\sum_{t=1}^{T} \log P(w[t] \mid w[<t])$, we divide by a function of length to get a normalized score that removes the inherent penalty for being longer: $\text{score} = \frac{1}{T^{\alpha}} \sum_{t=1}^{T} \log P(w[t] \mid w[<t])$, where α is the length penalty parameter that controls how aggressively we normalize. Figure 7.19 shows how different values of α affect the ranking of sequences of different lengths: at $\alpha = 0$, there is no length normalization and shorter sequences are strongly preferred; at $\alpha = 1$, we divide by length exactly, normalizing to per-token log-probability; values between 0 and 1, particularly around $\alpha = 0.6$, are commonly used in practice, providing partial correction for length bias while still maintaining some preference for conciseness. For sequence-to-sequence tasks like machine translation, beam search can additionally suffer from under-translation (failing to translate some source content) or over-translation (translating the same source content multiple times), which coverage penalty [Tu et al., 2016] addresses by explicitly penalizing beams that repeatedly attend to the same source positions or fail to attend to some positions at all, as shown in Figure 7.20.

The coverage mechanism is typically computed from the cross-attention weights in encoder-decoder mod-

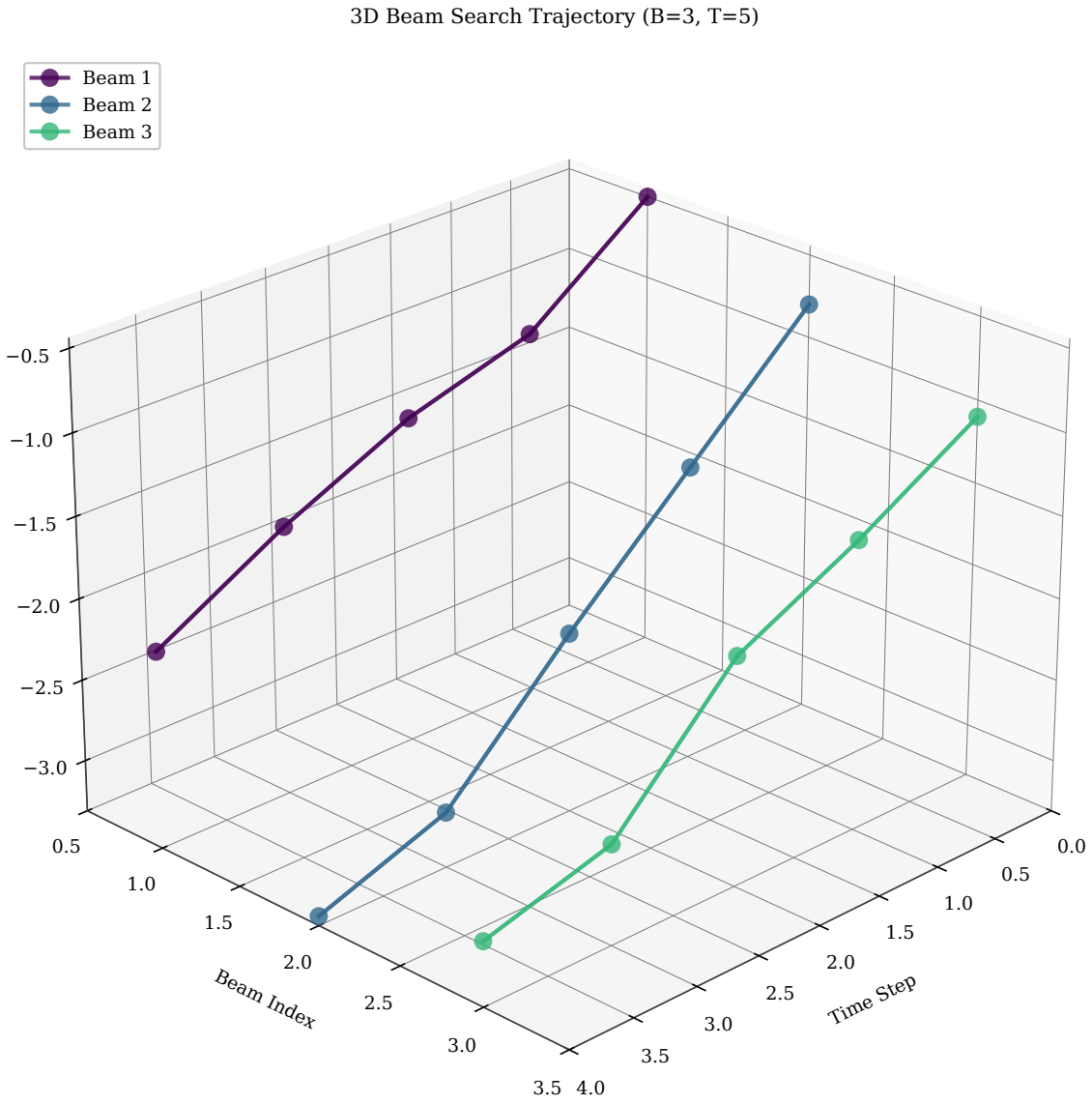


Figure 7.17: Three-dimensional visualization of beam search with $B = 3$. X-axis: time steps. Y-axis: beam index. Z-axis: cumulative log-probability score. Each beam is expanded and pruned at each step, maintaining the top 3 highest-scoring partial sequences.

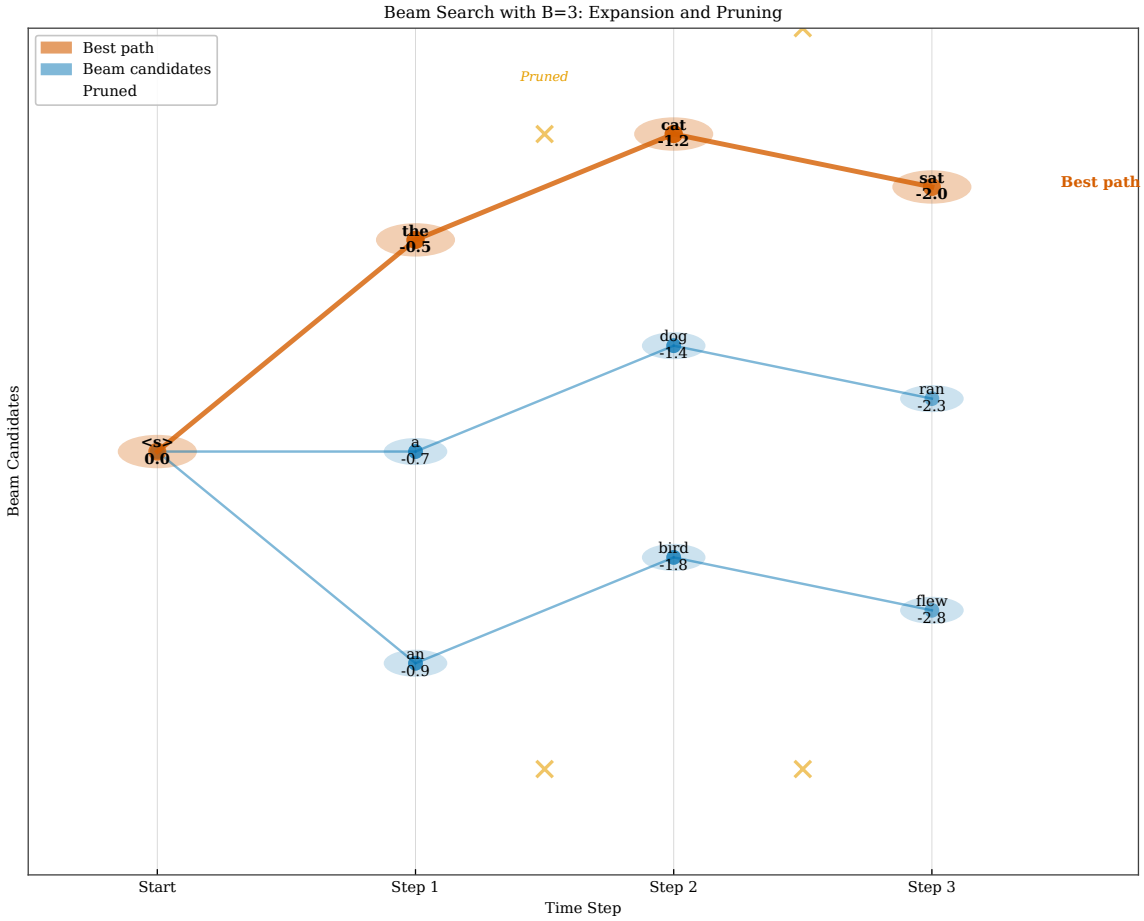


Figure 7.18: Two-dimensional view of beam search with $B = 3$. Each path shows tokens and their log-probabilities. Cumulative scores are shown at each node. Pruning at each step keeps only the highest-scoring beams, potentially dropping a path that started strong but became less probable.

els, tracking how much cumulative attention each source position has received across all decoding steps so far to ensure complete and non-redundant translation. Figure 7.20 visualizes coverage as a heatmap of attention over source positions (columns) and decoding steps (rows), where each cell shows the attention weight from that decoding step to that source position. Positions with too little cumulative attention (under-coverage, shown as cold colors) indicate content that may have been skipped, while positions with too much cumulative attention (over-coverage, shown as hot colors) indicate content that may be repeated. The coverage penalty adds a term to the beam score that encourages balanced attention, producing translations that cover the source content exactly once rather than missing or duplicating elements.

Despite its considerable success in structured generation tasks where a single high-quality output is desired, beam search has significant limitations for open-ended creative text generation. Like greedy decoding, beam search is completely deterministic, always producing the same output for a given input and beam width, which means it cannot generate the diversity of outputs that sampling methods provide. It tends to find high-probability but generic sequences that lack the diversity, creativity, and surprise expected in creative writing, dialogue, or brainstorming applications. Furthermore, beam search can suffer from a well-documented "blandness" or "genericness" problem: because it optimizes purely for probability, it systematically prefers safe, common phrases that appear frequently in training data over distinctive, interesting, or memorable formulations that might have slightly lower probability but much higher value. For these reasons, sampling-based methods like nucleus sampling are typically preferred for open-ended generation tasks, while beam search remains the standard and highly effective choice for translation, summarization, and other structured tasks where finding the single best output matters more than generating diverse alternatives.

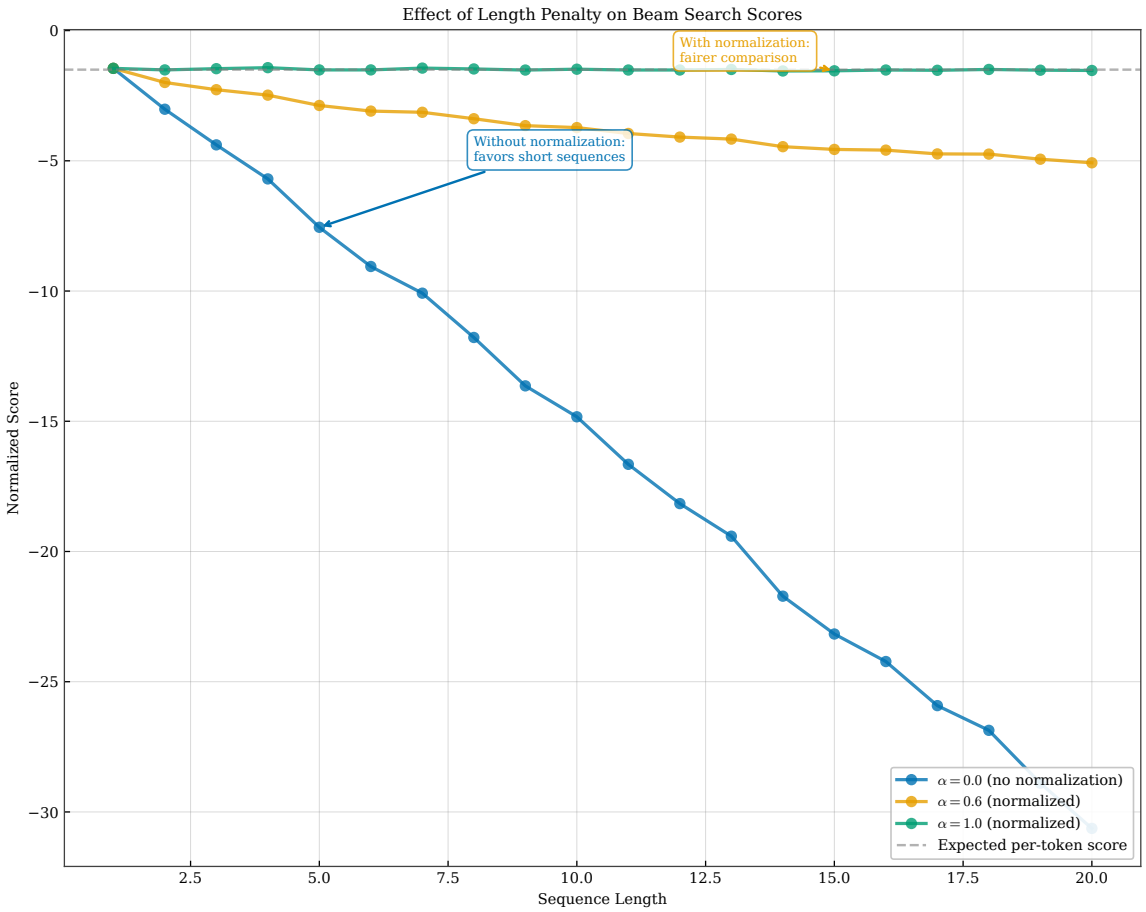


Figure 7.19: Effect of length normalization parameter α on sequence scores. Without normalization ($\alpha = 0$), longer sequences have lower scores. With full normalization ($\alpha = 1$), we compute per-token log-probability. Intermediate values like $\alpha = 0.6$ partially correct for length bias.

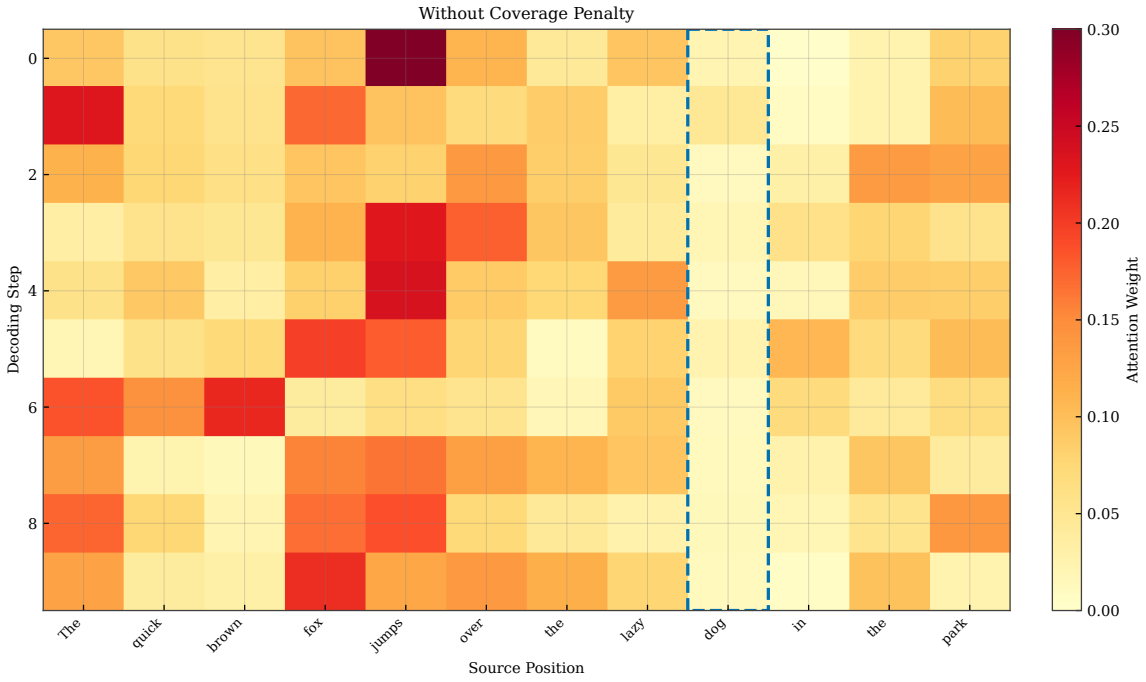


Figure 7.20: Coverage penalty visualization. Heatmap shows attention weights over source positions (columns) and decoding steps (rows). Coverage penalty penalizes under-attended positions (indicating potential missing content) and over-attended positions (indicating potential repetition).

7.8 Contrastive Decoding

Contrastive decoding improves generation quality by contrasting the predictions of a strong "expert" model with those of a weaker "amateur" model, exploiting the observation that failure modes are often more pronounced in less capable models [Li et al., 2023]. The key insight motivating this approach is that common failure modes like repetition, blandness, incoherence, and generic phrasing are more severe in smaller, less well-trained models—they lack the capacity to learn subtle preferences that distinguish high-quality text from mediocre text. By subtracting the amateur model's log-probabilities from the expert's, we mathematically amplify tokens that the expert prefers but the amateur does not, which empirically tend to be more interesting, distinctive, and less prone to degeneration. The contrastive score for token w is computed as $\log P_{\text{expert}}(w) - \lambda \log P_{\text{amateur}}(w)$, where λ controls the strength of the contrast between the two models. Additionally, a plausibility constraint ensures we only consider tokens where $P_{\text{expert}}(w)$ exceeds a minimum threshold, preventing the pathological case of selecting tokens that only appear good because the amateur model happens to strongly dislike them for idiosyncratic reasons unrelated to quality.

Figure 7.21 shows expert and amateur probability distributions for our running fantasy example, illustrating how the two models differ in their preferences. For completing "Once upon a time... young," both models assign relatively high probability to "prince" as the obvious and common continuation, but the amateur model also assigns substantial probability to repetitive or generic continuations that tend to lead to degeneration, while the expert model more strongly prefers diverse and distinctive continuations like "cartographer," "herbalist," or "tinker" that are unusual but contextually appropriate. The contrastive score amplifies these distinctive tokens through the subtraction: they have moderately high expert probability combined with low amateur probability, yielding high contrastive scores. Figure 7.22 demonstrates the qualitative improvement in practice: standard nucleus sampling produces serviceable but unremarkable text like "there lived a young prince who dreamed of adventure," while contrastive decoding produces "there lived a young cartographer whose maps revealed paths between worlds"—more specific with the unusual profession choice, more imaginative with the fantastical element, and more engaging overall by avoiding the generic patterns that appear countless times in training data.

Contrastive decoding requires running two models during inference rather than just one, which approximately doubles the computational cost and memory requirements compared to standard single-model decoding. However, the amateur model can be substantially smaller than the expert (e.g., a 125M parameter model serving as amateur contrasted with a 13B parameter expert), which limits the additional overhead to a manageable fraction of the total cost. The key requirement is that the amateur model be trained on similar data and capture similar patterns as the expert, just less well—this ensures that the difference between them captures quality-relevant distinctions rather than arbitrary differences in capability. Contrastive decoding is particularly effective for open-ended generation tasks like story writing and dialogue where the failure modes of smaller models (repetition, incoherence, blandness) are well-documented and where distinctiveness is valued. For structured tasks like translation where smaller models perform adequately and where the goal is accuracy rather than creativity, the benefits of contrastive decoding are less pronounced and the computational overhead may not be justified.

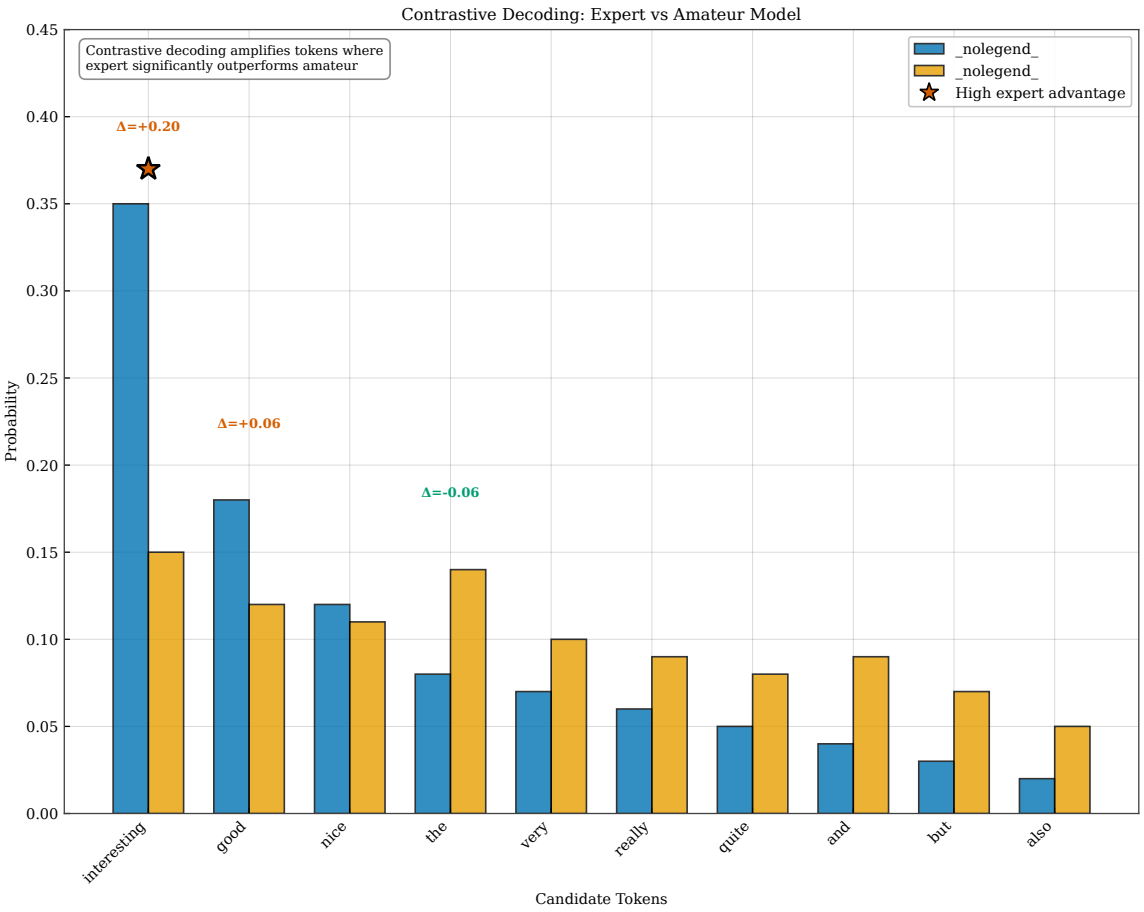


Figure 7.21: Expert and amateur probability distributions for contrastive decoding. Both models agree on common choices like “prince,” but the expert more strongly prefers distinctive options. Contrastive decoding amplifies tokens that the expert likes but the amateur does not.

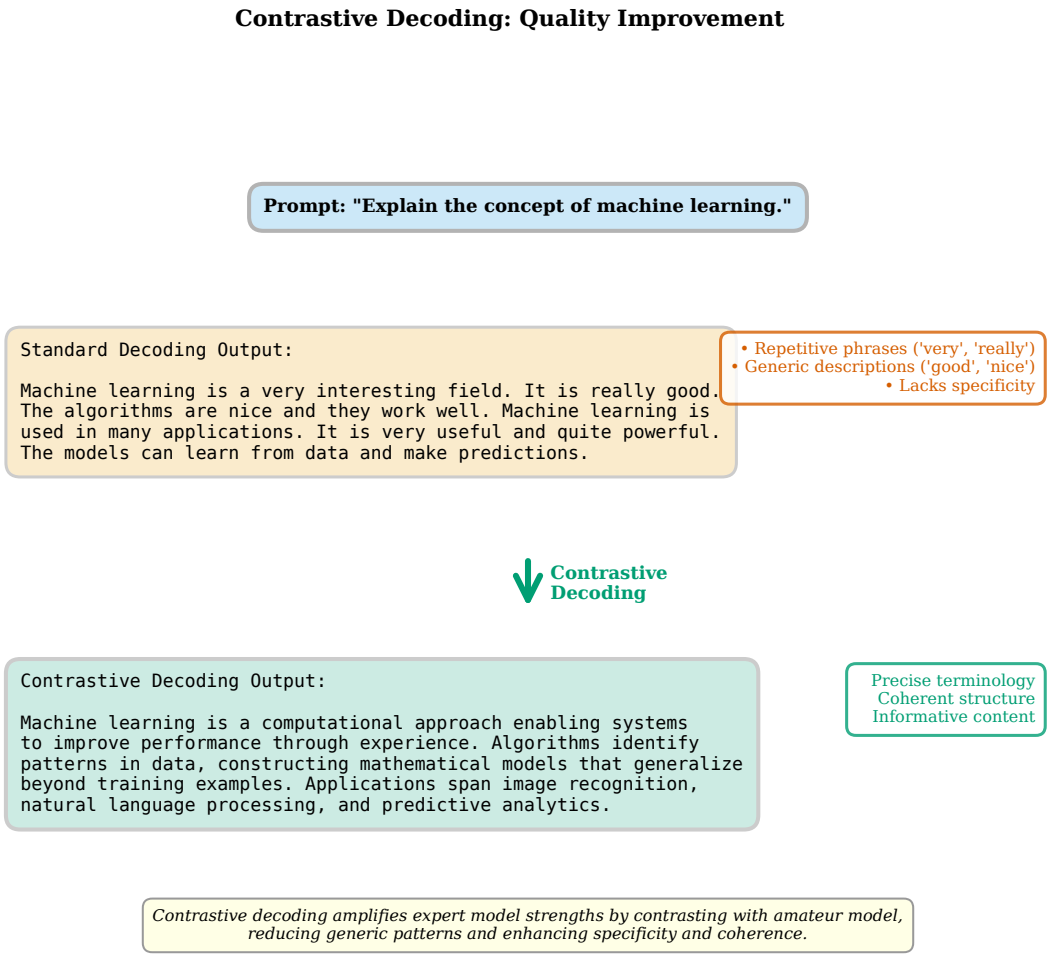


Figure 7.22: Comparison of standard versus contrastive decoding outputs. Standard decoding produces generic text following common patterns. Contrastive decoding produces more distinctive, creative text by amplifying tokens that differentiate the expert from the amateur model.

7.9 Constrained Decoding

Constrained decoding enforces that generated text must satisfy specific requirements, such as containing certain words, following particular formats, or adhering to grammatical structures, providing a way to control generation beyond what prompt engineering alone can achieve. Unlike the free-form decoding strategies discussed so far that allow any grammatically valid continuation, constrained decoding modifies the search process to guarantee constraint satisfaction, ensuring that the output definitely includes required elements rather than merely hoping the model produces them. The most common form is lexical constraint: requiring that specific words or phrases appear somewhere in the output regardless of where the model's probability distribution would naturally place them. For example, we might require that a story about dragons must include the word "dragon" at least once, or that a product description must include specific feature keywords like "waterproof" and "lightweight." Grid beam search [Hokamp and Liu, 2017] and NeuroLogic decoding [Lu et al., 2022] are sophisticated algorithms that efficiently search for high-probability sequences satisfying such constraints while avoiding the combinatorial explosion that naive approaches would encounter.

Figure 7.23 illustrates grid beam search for generating our fantasy story with the constraint that the output must include both "dragon" and "treasure," demonstrating the two-dimensional structure of the constrained search. The grid has decoding steps on the x-axis (progress through the sequence) and constraint satisfaction level on the y-axis (how many required constraints have been satisfied). Beams move rightward by generating tokens and upward when they generate a token satisfying a pending constraint; the algorithm ensures only complete sequences at the top level (all constraints satisfied) are considered as final outputs. Figure 7.24 shows the tree of valid sequences, with paths successfully including both words highlighted in green as valid candidates, while paths reaching end-of-sequence without satisfying all constraints are grayed out and discarded. The constrained decoding algorithm actively prunes invalid paths during search rather than generating freely and filtering afterward, which is crucial for efficiency since rejection sampling would be extremely slow for rare constraints that most unconstrained generations would fail to satisfy. The algorithm might produce output like "there lived a young knight who sought the dragon's treasure hidden in the mountain cave," where both constraint words appear naturally in a coherent narrative.

Beyond simple lexical constraints requiring specific words, constrained decoding can enforce a wide variety of structural requirements that would be difficult or impossible to achieve through prompting alone. These include JSON formatting where the output must be valid parseable JSON with specific required fields, grammatical patterns where certain syntactic structures must appear, length limits where the output must fall within a specific word or token count range, and rhyme or meter constraints for poetry generation. NeuroLogic decoding extends the basic grid beam search framework with lookahead heuristics that estimate the probability of satisfying remaining constraints from any partial sequence, improving efficiency by prioritizing beams that are likely to succeed and deprioritizing beams that would require low-probability token sequences to satisfy their remaining constraints. These techniques are particularly valuable in applications like information extraction (where outputs must follow specific schemas to be parsed by downstream systems), code generation (where outputs must be syntactically valid in the target programming language), and controlled text generation (where outputs must match specific style or content requirements specified by users).

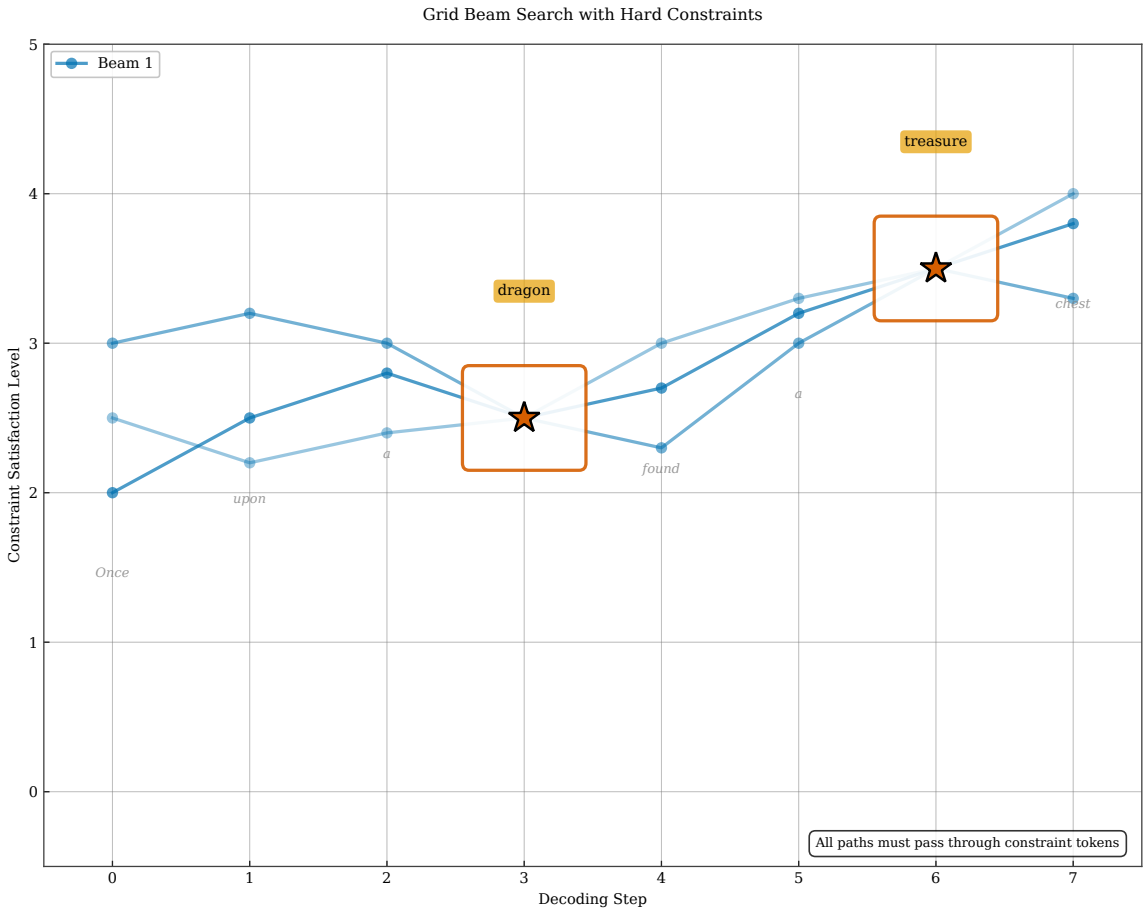


Figure 7.23: Grid beam search for constrained decoding. X-axis: decoding steps. Y-axis: number of constraints satisfied. Beams progress rightward and upward, with the final output required to reach the top level (all constraints satisfied). Red markers indicate where constraint words are generated.

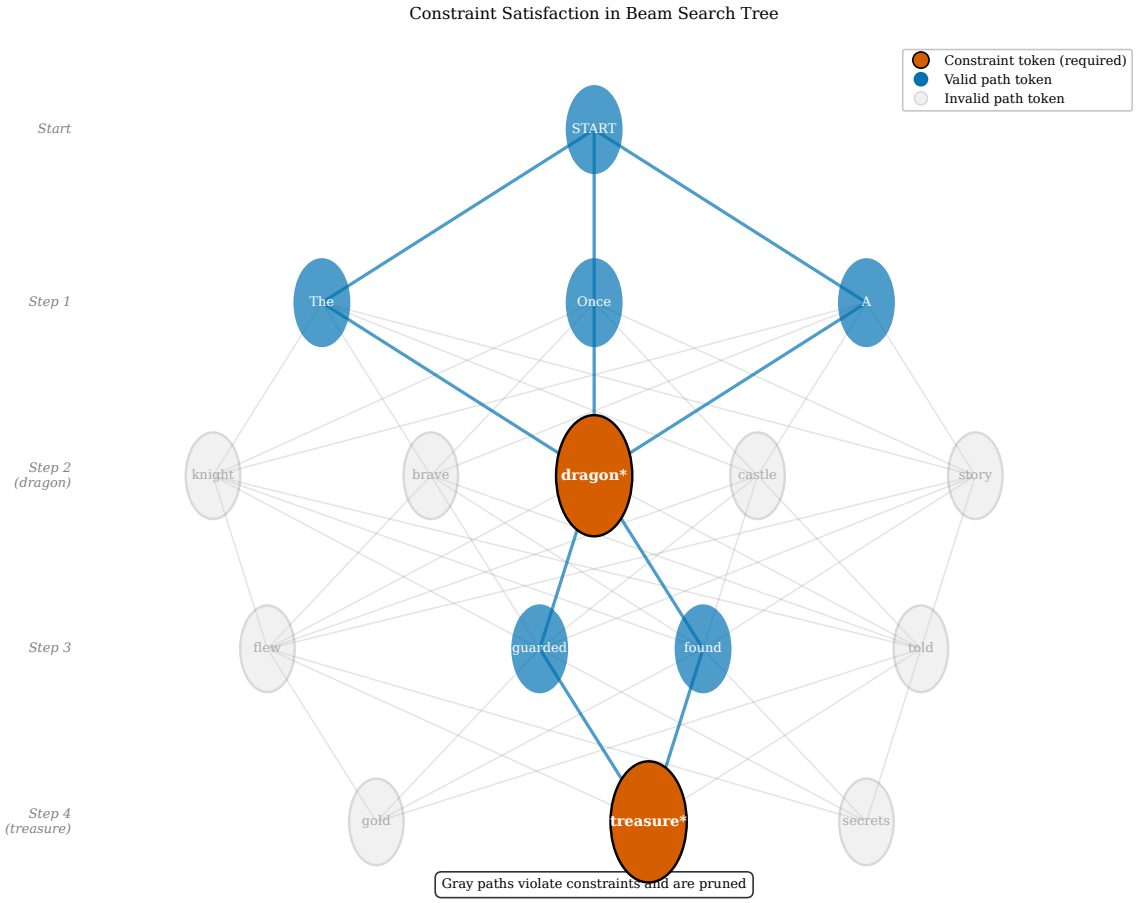


Figure 7.24: Constraint satisfaction paths in the decoding tree. Valid paths (green) include all required words (“dragon” and “treasure”). Invalid paths (gray) fail to satisfy constraints. Constrained decoding prunes invalid paths during search.

7.10 Speculative Decoding

Speculative decoding addresses a fundamental inefficiency inherent in autoregressive generation: because each token depends on all previous tokens through the attention mechanism, we cannot parallelize the generation of consecutive positions using the main model, leaving valuable GPU parallelism unexploited [Leviathan et al., 2023, Chen et al., 2023]. The standard approach to autoregressive generation computes tokens one at a time, running a full forward pass of the large model for each token, making complete use of the model's capacity but achieving only sequential throughput regardless of available parallel compute resources. Speculative decoding breaks this serialization bottleneck by using a small, fast "draft" model to generate multiple candidate tokens speculatively, exploiting parallelism in the draft model, and then using the large target model to verify these candidates in a single batched forward pass that processes all candidates simultaneously. When the draft model's predictions match what the large model would have produced through standard sequential decoding, we accept multiple tokens at once and advance by several positions; when predictions differ, we reject from the point of divergence and fall back to the large model's prediction for that position.

The sampling trajectory visualization in Figure 7.25 shows multiple speculative paths through the probability space, illustrating how the draft model explores possible futures that the target model will later evaluate. The draft model generates candidate sequences shown as branching colored paths, each representing a possible continuation of the current prefix based on the draft model's probability distribution, extending for several tokens into the future to create a tree of speculative continuations. The large target model then evaluates all these candidates simultaneously in a single forward pass by using appropriate causal attention masks, computing what it would have sampled at each position. For each path, we determine the longest prefix where the draft model's tokens match the target model's preferences according to a probabilistic acceptance criterion. This parallel verification is the key source of speedup: instead of running the large model k times for k tokens sequentially, we run it once to verify all k candidates, achieving up to k-fold speedup when the draft model's predictions are accepted. The draft-then-verify pipeline shown in Figure 7.26 operates in two phases: in the draft phase, the small fast draft model (perhaps 10-100x fewer parameters than the target) generates a candidate sequence of k tokens autoregressively; in the verify phase, the target model computes probabilities for all k positions simultaneously using a single forward pass with appropriate attention masking, accepting tokens that match its preferences according to a carefully designed probabilistic acceptance criterion and resampling from the rejected position onward when predictions differ, with acceptance rates of 70-90% achievable when the models are well-matched.

Speculative decoding provides practical speedups of 2-3x on real systems without changing the output distribution: accepted sequences are provably exactly what the target model would have generated through standard sequential decoding, maintaining perfect distributional equivalence with no quality degradation. This property distinguishes speculative decoding from approximation methods like quantization or distillation that trade quality for speed. The method is particularly effective when the draft model is a smaller version of the target model trained on the same data, or a model that has been distilled specifically to match the target's distribution, because such models achieve high acceptance rates with minimal distribution mismatch. The primary cost is running the draft model, which should be fast enough that its computational overhead is more than offset by the parallelism gains from accepting multiple tokens per target model forward pass. Speculative decoding has become increasingly important as large language models grow to hundreds of billions of parameters, offering a practical way to reduce inference latency and cost without sacrificing the quality that large models provide.

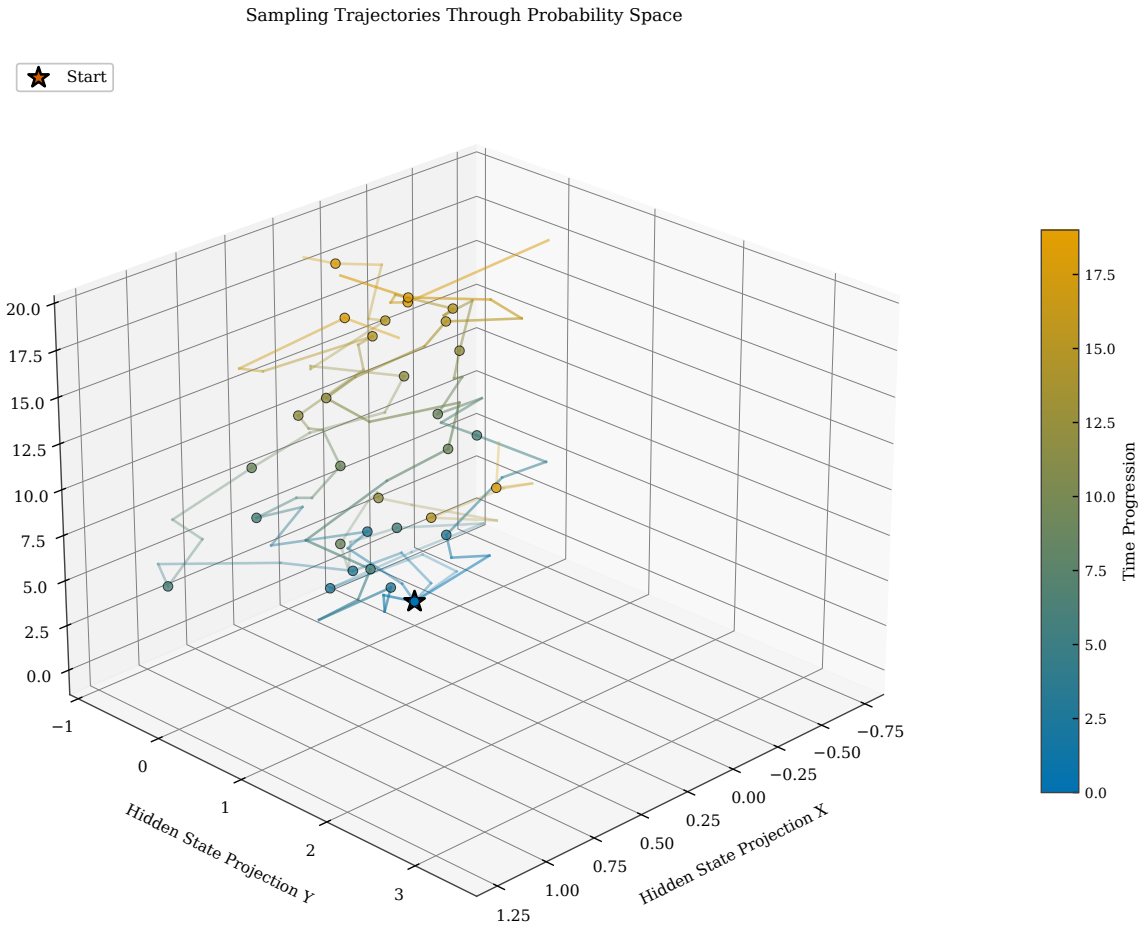


Figure 7.25: Speculative decoding trajectories in probability space. The draft model generates multiple candidate paths (colored trajectories). The large model verifies all paths in parallel, accepting matching prefixes. Accepted paths advance generation by multiple tokens at once.

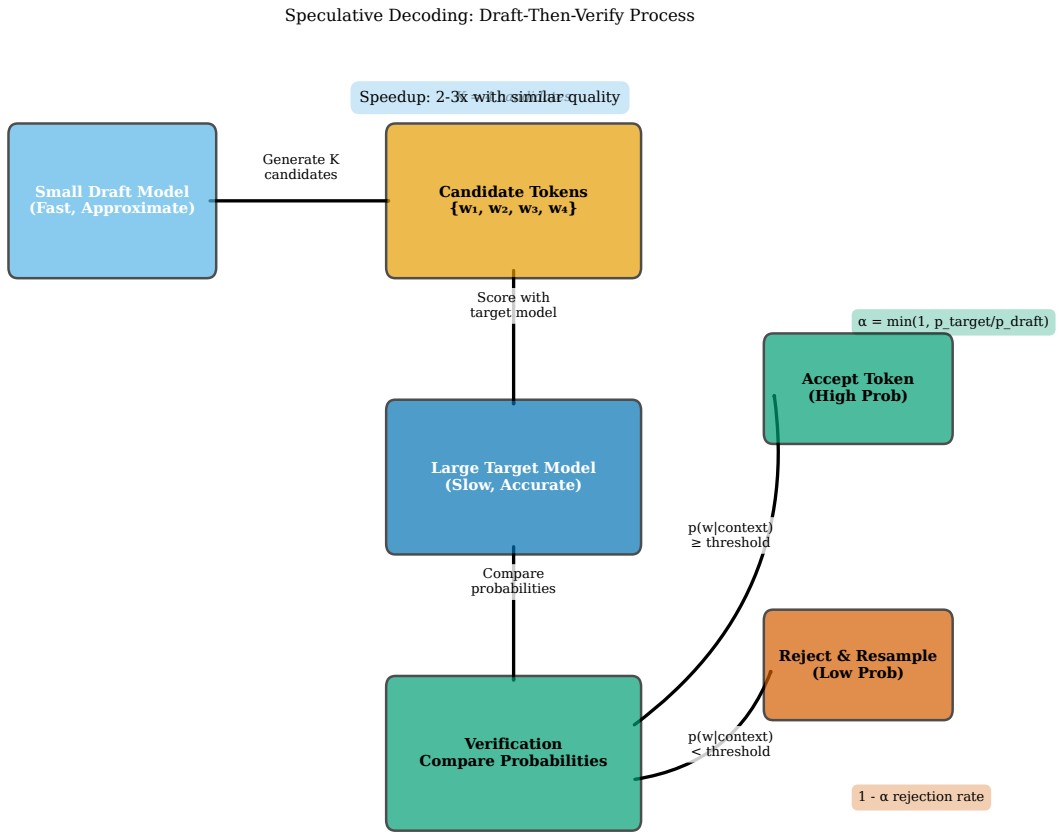


Figure 7.26: The speculative decoding pipeline. The draft model generates candidate tokens quickly. The target model verifies all candidates in parallel. Matching tokens are accepted; mismatches trigger resampling from the target model. Speedup depends on acceptance rate.

7.11 Context Representation in Decoding

Decoding strategies convert the transformer's rich context representations into discrete token sequences, but the efficiency and quality of this conversion depend critically on how context is accessed, maintained, and updated during the generation process. Chapter **??** explained how transformers represent context through the key-value (KV) cache: instead of recomputing attention over all previous positions from scratch at each step, we cache the key and value projection vectors and append only the new position's KV pair after each token is generated, reducing the complexity of each decoding step from $O(T^2)$ to $O(T)$ while maintaining access to the full context representation. The probability distribution at each decoding step reflects all context learned by the model through the entire training process and the specific content of the current generation: the KV cache encodes the complete prefix including both the user's prompt and all tokens generated so far, and the attention mechanism allows the current position to access any previous position with learned attention weights. Decoding strategies navigate this rich probability landscape using different algorithmic approaches: greedy decoding follows the steepest gradient of probability, always moving deterministically toward the mode; sampling methods explore the probability surface more broadly, visiting lower-probability regions with frequency proportional to their probability mass, enabling diversity and creativity; beam search maintains multiple paths simultaneously, hedging against locally optimal but globally suboptimal choices. Each strategy extracts different information from the same underlying context representation, producing characteristically different outputs from identical prompts and revealing the fundamental trade-offs between quality, diversity, and computational cost that practitioners must navigate.

Figure 7.27 visualizes the quality-diversity trade-off surface that different decoding strategies navigate, showing how the choice of decoding method determines which region of this trade-off space we occupy. Quality (measured by metrics like perplexity on held-out data, BLEU score against references, or human evaluation of coherence and fluency) and diversity (measured by metrics like self-BLEU across multiple generations, count of distinct n-grams, or entropy of the output distribution) are often in tension: strategies that maximize quality tend to produce repetitive, similar outputs across multiple runs, while strategies that maximize diversity risk incoherence or low fluency. Greedy decoding and beam search occupy the high-quality, low-diversity corner of this surface—they produce coherent text but always the same text. Pure sampling with high temperature occupies the high-diversity, variable-quality region—outputs differ across runs but quality is inconsistent. Nucleus sampling, top-k, and contrastive decoding find different trade-off points on this surface, with the optimal choice depending on whether the application values consistency or variety more.

How This Chapter Represents Context:

- The KV cache stores context as key-value pairs for efficient attention during generation
- Each decoding step produces a probability distribution reflecting the full context
- Decoding strategies navigate the probability landscape defined by the context representation
- Trade-offs between quality and diversity reflect different ways of exploiting context

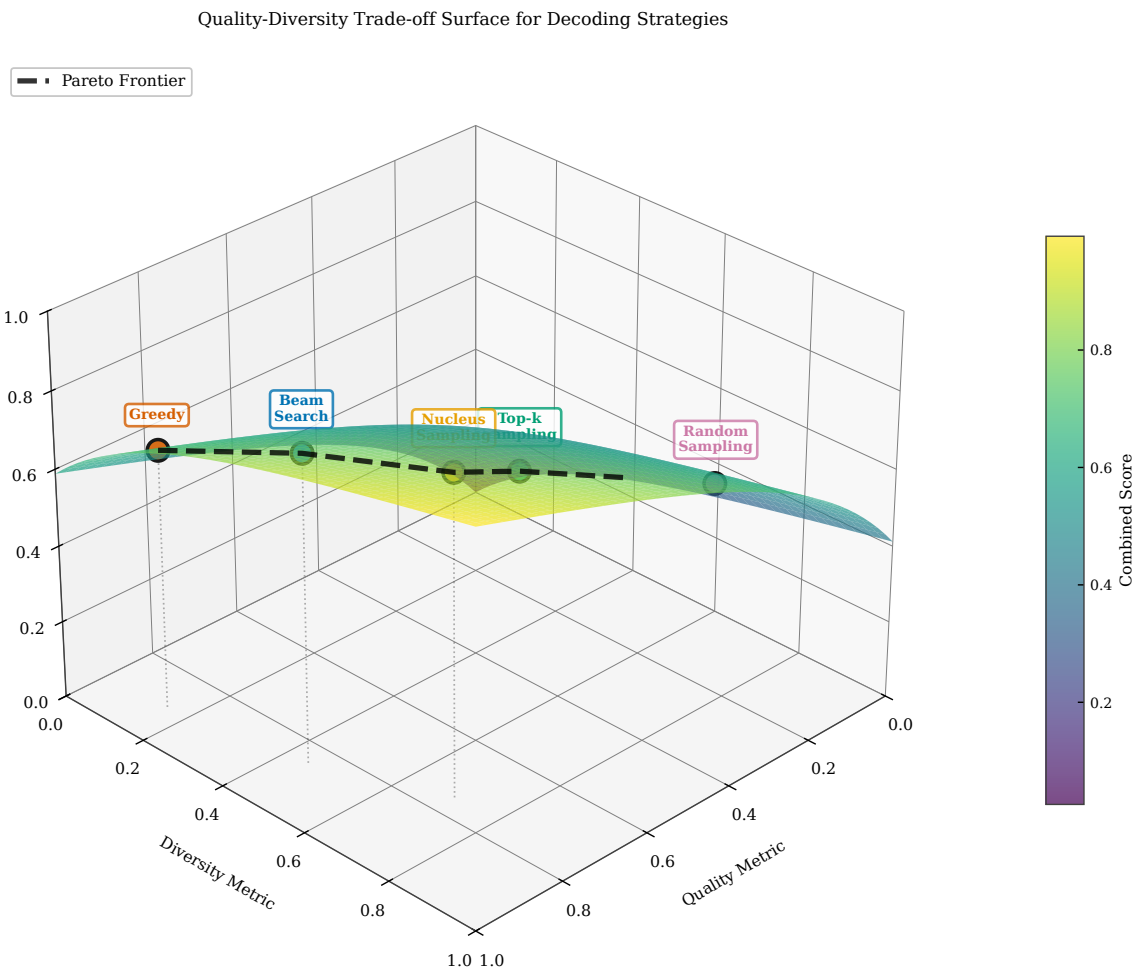


Figure 7.27: Quality-diversity trade-off surface for decoding strategies. Different strategies occupy different positions on this surface: greedy decoding maximizes quality but sacrifices diversity; pure sampling provides diversity but risks quality; nucleus and beam search find different trade-off points.

We can now predict better because:

- Temperature, top-k, and nucleus sampling convert probability distributions to diverse, high-quality text by controlling the quality-diversity trade-off
- Beam search finds globally coherent sequences by maintaining multiple hypotheses, particularly valuable for translation and summarization
- Contrastive decoding improves output quality by contrasting expert and amateur models, reducing degeneration and blandness
- Speculative decoding accelerates generation without changing output distribution, achieving 2-3x speedup through parallel verification

Next: Chapter **??** explores how we train models to produce probability distributions that decode well, examining loss functions, optimization algorithms, and the challenges of training at scale.

Exercises

1. Calculate the greedy decoding output for the first 5 tokens given a toy probability distribution where $P(\text{the}) = 0.3$, $P(\text{cat}) = 0.2$, and other tokens share the remaining probability uniformly. What potential issues might arise with this simple distribution?

2. Derive the relationship between temperature τ and the entropy of the softmax distribution for a vocabulary of size 2 with logits $z_1 = 1$ and $z_2 = 0$. Plot entropy as a function of temperature.

3. Compare top-k sampling with $k = 10$ to nucleus sampling with $p = 0.9$ on a peaked distribution where the top token has probability 0.8 and on a flat distribution where each of the top 100 tokens has probability 0.008. Which method adapts better to the distribution shape?

4. Implement the nucleus (top-p) sampling algorithm in pseudocode. Your algorithm should take a probability distribution and threshold p as input and return a sampled token.

5. Explain why length normalization is necessary in beam search. Calculate the raw and normalized scores (with $\alpha = 0.6$) for sequences of length 5 and length 10 that have the same per-token log-probability of -2.0.

6. For a beam search with beam width $B = 4$ and vocabulary size $|\mathcal{V}| = 10000$, how many candidate sequences are considered at each expansion step? How many survive after pruning?

7. The coverage penalty in beam search tracks attention over source positions. Describe a scenario where under-translation would occur without coverage penalty and explain how the penalty prevents it.

8. In contrastive decoding, what happens if the amateur model assigns probability 0 to a token that the expert model considers likely? How does the plausibility constraint address this issue?

9. Compare the computational costs of greedy decoding, beam search with $B = 5$, and speculative decoding with a draft model generating 4 candidate tokens. Express costs in terms of forward passes of the target model.

10. Design a constrained decoding task for generating product descriptions that must include specific feature keywords. What constraints would you impose, and how would grid beam search handle them?

11. * Derive the acceptance probability formula for speculative decoding that ensures the output distribution exactly matches standard sampling from the target model. Why is this distributional equivalence important?

12. * Analyze the failure mode of typical sampling where excluding high-probability tokens hurts coherence. Under what distribution conditions would typical sampling underperform nucleus sampling, and vice versa?

Bibliography

Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. Accelerating large language model decoding with speculative sampling. *arXiv preprint arXiv:2302.01318*, 2023.

Angela Fan, Mike Lewis, and Yann Dauphin. Hierarchical neural story generation. pages 889–898, 2018.

Chris Hokamp and Qun Liu. Lexically constrained decoding for sequence generation using grid beam search. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 1535–1546, 2017.

Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration. *arXiv preprint arXiv:1904.09751*, 2020.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. pages 19274–19286, 2023.

Xiang Lisa Li, Ari Holtzman, Daniel Fried, Percy Liang, Jason Eisner, Tatsunori Hashimoto, Luke Zettlemoyer, and Mike Lewis. Contrastive decoding: Open-ended text generation as optimization. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 12286–12312, 2023.

Ximing Lu, Sean Welleck, Peter West, Liwei Jiang, Jungo Kasai, Daniel Khashabi, Ronan Le Bras, Lianhui Qin, Youngjae Yu, Rowan Zellers, and Yejin Choi. Neurologic a*esque decoding: Constrained text generation with lookahead heuristics. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*, pages 780–799, 2022.

Clara Meister, Tiago Pimentel, Gian Wiher, and Ryan Cotterell. Locally typical sampling. *Transactions of the Association for Computational Linguistics*, 11:102–121, 2023.

Zhaopeng Tu, Zhengdong Lu, Yang Liu, Xiaohua Liu, and Hang Li. Modeling coverage for neural machine translation. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 76–85, 2016.

Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

Index

Contents

Chapter 1

Training Language Models

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

- Defining loss functions that measure prediction quality mathematically
- Applying optimization algorithms that improve model parameters systematically
- Designing learning rate schedules that balance exploration and convergence
- Scaling training across multiple devices and machines efficiently

The previous chapters have established the architecture of modern language models: transformers that attend to context (Chapter) and decoding strategies that convert probability distributions into text (Chapter). However, a randomly initialized transformer produces meaningless probability distributions, assigning roughly equal likelihood to every word in the vocabulary regardless of context. The critical missing piece is *training*: the process of adjusting billions of parameters so that the model learns to predict contextually appropriate words. This chapter examines the mathematical foundations and practical techniques that transform an untrained transformer into a capable language model, connecting the loss functions that measure prediction quality to the optimization algorithms that systematically improve model parameters, and scaling these techniques to the massive compute clusters required for modern large language models.

1.1 The Training Objective

The fundamental question of training is deceptively simple: given a corpus of text, how do we measure whether a model's predictions are good? The

answer lies in cross-entropy loss, a mathematical quantity that directly measures the model's ability to predict the next word. Consider our running example: "The software engineer who had been working remotely from the mountain cabin for several months finally submitted the code that would..." At each position in this sequence, the model must assign a probability distribution over the entire vocabulary, and cross-entropy loss penalizes the model for assigning low probability to the word that actually appears next. The loss is computed as the negative log-probability of the correct word, $\mathcal{L} = -\log P(w_t \mid w_{<t}; \boldsymbol{\theta})$, where $\boldsymbol{\theta}$ represents the model's parameters. When the model assigns high probability to the correct word, the loss is low; when it assigns low probability, the loss is high, potentially reaching infinity as the probability approaches zero.

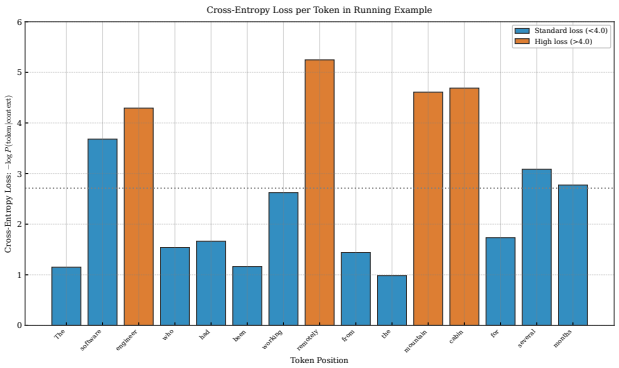


Figure 1.1: Cross-entropy loss per token for our running example. Content words like "engineer" and "remotely" incur higher loss than function words like "the" and "who" because they are harder to predict from context. The model must learn to assign high probability to exactly these challenging tokens.

The total loss for a sequence aggregates the per-token losses, typically as an average: $\mathcal{L} = -\frac{1}{T}\sum_{t=1}^{T} \log P(w_t \mid w_{<t}; \boldsymbol{\theta})$, where T is the sequence length. This formulation connects directly to the probability distributions over vocabulary $\mathcal{V}$ that we introduced in earlier chapters, where the model must assign $P(w)$ to every word in the vocabulary at each position. The information-theoretic interpretation is profound: the loss equals the cross-entropy $H(P_{\text{data}}, P_{\boldsymbol{\theta}})$ between the true data distribution and the model's distribution, which decomposes into the entropy of the data plus the KL divergence between distributions. Minimizing cross-entropy loss is therefore equivalent to minimizing the divergence between what the model predicts and what actually occurs in natural text, pushing $P(w)$ for correct words toward 1.0. The connection to Shannon's work (Chapter) becomes clear: a model that achieves low cross-entropy has learned to predict text with efficiency approaching the intrinsic entropy of language, extracting and ex-

ploiting the statistical regularities that make language predictable. Every gradient update nudges the probability mass in $\mathcal{V}$ toward the empirically correct distribution.

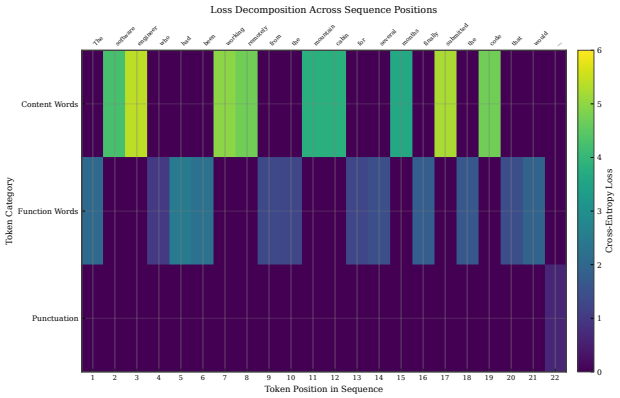


Figure 1.2: Loss decomposition across sequence positions and word categories. Function words (articles, prepositions, pronouns) typically have lower loss because they are more predictable from syntactic context. Content words (nouns, verbs, adjectives) carry more semantic information and thus higher prediction uncertainty.

The loss landscape visualizes how the loss changes as we vary the model parameters, providing crucial intuition for understanding optimization dynamics. For a model with billions of parameters, this landscape exists in a space of unimaginable dimensionality, yet its structure determines whether training succeeds or fails, whether the model converges to useful representations or stagnates in poor configurations. Research has revealed that these landscapes are surprisingly well-behaved for transformer language models: they tend to have many local minima, but these minima achieve similar loss values, making the specific minimum found less important than the training dynamics that navigate toward any good solution. This phenomenon, sometimes called the "lottery ticket" hypothesis, suggests that overparameterized models have sufficient capacity that many different parameter configurations can represent effective next-word predictors. The landscape contains saddle points where gradients vanish in some directions but not others, potentially slowing optimization by providing misleading gradient signals, and flat regions where many parameter configurations achieve similar loss, making progress slow but stable. Understanding this geometry motivates the optimization algorithms we develop in the following sections and explains why certain hyperparameter choices matter.

Training language models uses a technique called teacher forcing, where the model receives the ground-truth previous tokens as input during training, rather than its own predictions. This approach enables efficient parallel com-

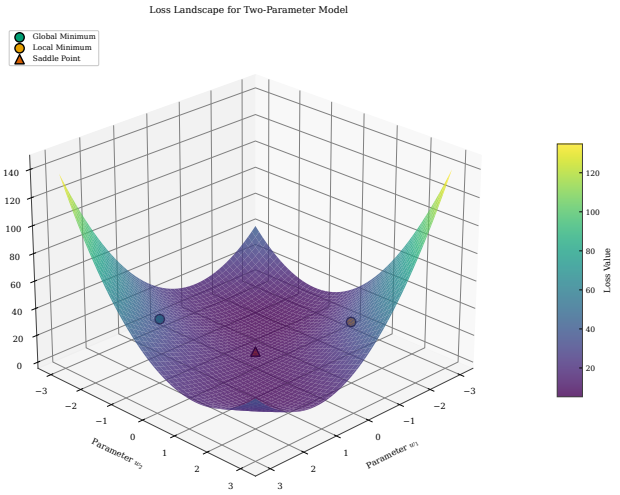


Figure 1.3: A 3D visualization of a loss surface for a simplified two-parameter model. The surface exhibits local minima, saddle points, and flat regions that optimization algorithms must navigate. Real language models have billions of parameters, but the geometric intuitions from low-dimensional visualizations guide algorithm design.

putation because all positions can be processed simultaneously—the model does not need to wait for its own predictions at earlier positions. Teacher forcing also provides a stable learning signal by ensuring the model always conditions on correct context, preventing error accumulation where early mistakes cascade into increasingly incorrect predictions. The alternative, autoregressive training, would require sequential generation and could produce highly variable gradients as small errors compound. However, teacher forcing creates a train-test mismatch called exposure bias: during inference, the model must condition on its own potentially incorrect predictions, a situation never encountered during training. Modern large language models typically accept this mismatch, as the benefits of efficient training outweigh the modest degradation from exposure bias when models are sufficiently capable.

Perplexity provides an interpretable transformation of cross-entropy loss that connects directly to prediction quality and human intuition about model capability: $\text{PPL} = \exp(\mathcal{L})$. A perplexity of 100 means the model is, on average, as uncertain as if it were choosing uniformly among 100 equally likely words at each position, effectively spreading its probability mass across 100 candidates rather than confidently predicting a small set. This interpretation as an "effective vocabulary size" makes perplexity intuitive for practitioners: lower perplexity indicates the model has narrowed down possibilities more effectively by learning which words in $\mathcal{V}$ are contextually appropriate and

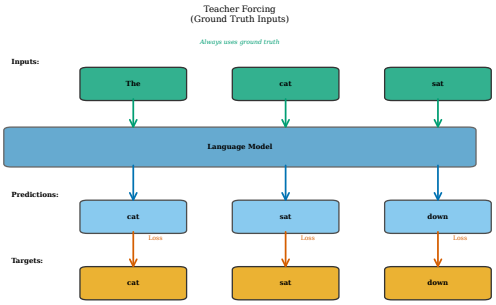


Figure 1.4: Teacher forcing versus autoregressive training. In teacher forcing (left), the model receives ground-truth tokens as input at each position, enabling parallel computation. In autoregressive training (right), the model conditions on its own predictions, requiring sequential generation but matching the inference setting.

which are unlikely. State-of-the-art language models achieve perplexities between 10 and 30 on standard benchmarks like Penn Treebank or WikiText, indicating they reduce uncertainty to a small fraction of the full vocabulary of typically 50,000 or more tokens, demonstrating learned understanding of context. The exponential relationship between loss and perplexity means that small improvements in cross-entropy translate to significant perplexity reductions—a 0.1 decrease in loss corresponds to roughly a 10% relative reduction in perplexity, which practitioners find meaningful for tracking progress. Researchers track both metrics, using loss for optimization because it provides well-behaved gradients suitable for backpropagation, and perplexity for intuitive comparison across models and datasets because it has a direct interpretation in terms of prediction difficulty.

1.2 Gradient-Based Optimization

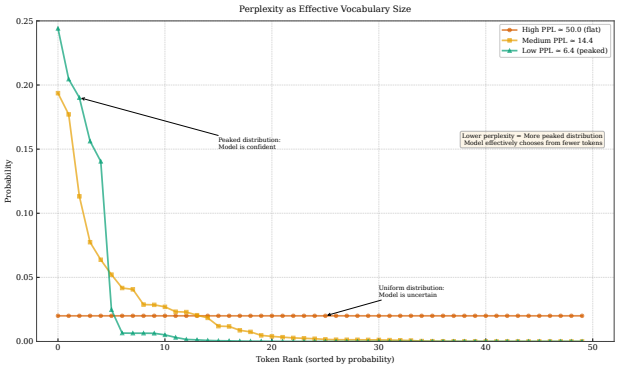


Figure 1.5: Perplexity as effective vocabulary size. A distribution with high perplexity (left) spreads probability across many words, indicating high uncertainty. A distribution with low perplexity (right) concentrates probability on few words, indicating confident prediction. Training reduces perplexity by learning which words are likely in each context.

With a loss function defined, we need a method to systematically improve the model's parameters so that predictions become more accurate over time. Gradient-based optimization provides the solution: compute the gradient of the loss with respect to each parameter, then update parameters in the direction that decreases loss. The gradient $\nabla_{\boldsymbol{\theta}}\mathcal{L} = \nabla_{\boldsymbol{\theta}\mathcal{L}}$ is a vector pointing in the direction of steepest loss increase, so we update parameters by moving in the opposite direction: $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta\nabla_{\boldsymbol{\theta}}\mathcal{L}$, where η is the learning rate controlling step size. This simple rule, applied billions of times, transforms random parameter values into representations that capture the statistical structure of language and enable accurate next-word prediction. For a transformer with billions of parameters, this gradient computation requires backpropagation through every layer, attention mechanism, and feed-forward network, a process that has complexity $O(T^2 \cdot d_{\text{model}})$ per layer due to the attention operation over sequence length T with hidden dimension d_{model}. Despite this computational cost, modern hardware including GPUs and TPUs and software frameworks like PyTorch and JAX make gradient computation practical through automatic differentiation that efficiently computes gradients for arbitrary computational graphs without manual derivation.

Stochastic gradient descent (SGD) computes gradients on small batches of data rather than the entire corpus, introducing noise into the gradient estimates but enabling practical computation on corpora containing billions of tokens. Each batch provides an unbiased estimate of the true gradient, and the stochasticity can actually help optimization by allowing escape from sharp local minima and saddle points that would trap a deterministic optimizer. The noise acts as implicit regularization, pushing the optimizer toward broader minima that often generalize better to unseen text. How-

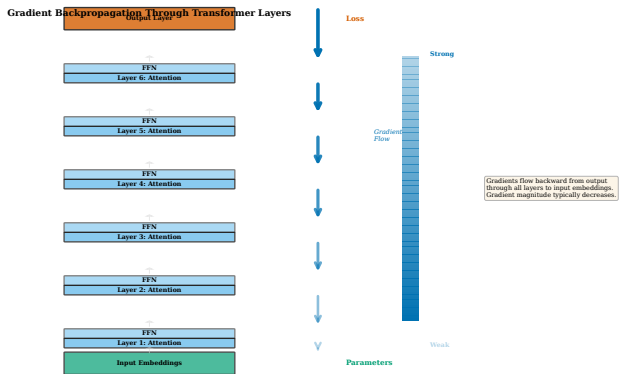


Figure 1.6: Gradient flow through transformer layers during backpropagation. Gradients flow backward from the loss through each layer's attention and feed-forward components. The residual connections (Chapter) provide direct gradient paths that prevent vanishing gradients in deep networks.

ever, vanilla SGD has significant limitations for training language models at scale: gradients can point in rapidly changing directions across batches, causing oscillation that wastes computation; the optimal learning rate differs dramatically across parameters, with some benefiting from aggressive updates and others requiring delicate adjustments; and convergence can be painfully slow in flat regions of the loss landscape where gradients are small but the model has not yet found good representations. These limitations motivated the development of more sophisticated optimizers that adapt to the local geometry of the loss landscape and maintain useful information across training steps.

Momentum addresses SGD's oscillation problem by maintaining an exponential moving average of past gradients, effectively remembering gradient history: $v_t = \beta v_{t-1} + \nabla_{\boldsymbol{\theta}} \mathcal{L}$ and $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta v_t$, where β (typically 0.9) controls how much history to retain, with higher values preserving more gradient information from earlier steps. This formulation causes the optimizer to build up velocity in consistent gradient directions while dampening oscillations in directions where gradients alternate signs, automatically distinguishing between persistent descent directions and noisy fluctuations. The physical analogy is a ball rolling down the loss landscape, accumulating momentum in the downhill direction—the ball smoothly traverses small bumps while accelerating down consistent slopes. Momentum significantly accelerates convergence in practice, particularly for loss landscapes with narrow valleys where SGD would oscillate back and forth across the valley while making slow progress along it, wasting many gradient updates on corrections rather than progress. The accumulated velocity allows momentum to effectively take larger steps in the consistent direction while remaining stable in oscillating directions, achieving faster convergence without increasing the

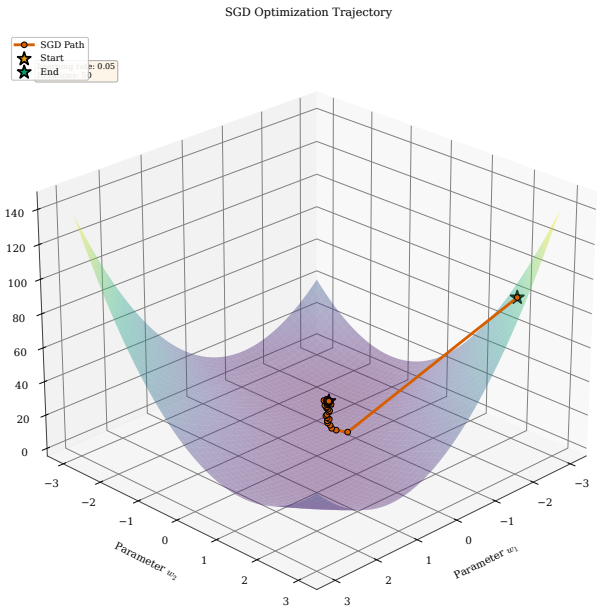


Figure 1.7: SGD optimization trajectory on a loss surface. The path oscillates significantly, especially when the surface has different curvatures in different directions. SGD makes slow progress along narrow valleys and can overshoot in steep directions, motivating momentum-based methods.

nominal learning rate.

Adam (Adaptive Moment Estimation) combines momentum with adaptive learning rates per parameter, becoming the de facto optimizer for training language models since its introduction [2]. Adam maintains two exponential moving averages: the first moment $m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla_{\boldsymbol{\theta}}\mathcal{L}$ (similar to momentum, tracking gradient direction) and the second moment $v_t = \beta_2 v_{t-1} + (1-\beta_2)\nabla_{\boldsymbol{\theta}}\mathcal{L}^2$ (tracking gradient magnitudes, providing scale information). The update rule $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \cdot \hat{m}_t/(\sqrt{\hat{v}_t}+\epsilon)$ uses bias-corrected moments $\hat{m}_t = m_t/(1-\beta_1^t)$ and $\hat{v}_t = v_t/(1-\beta_2^t)$ to counteract initialization bias in the early steps when moving averages have not yet accumulated sufficient history. The division by $\sqrt{\hat{v}_t}$ adapts the effective learning rate per parameter: parameters receiving large gradients get smaller updates to prevent overshooting, while those receiving small gradients get relatively larger updates to accelerate progress in flat directions. This adaptation is crucial for transformers, where different parameter groups including embeddings, attention weights, and feed-forward weights naturally benefit from different effective learning rates due to their distinct gradient distributions and roles in the model.

AdamW modifies Adam to implement weight decay correctly, address-

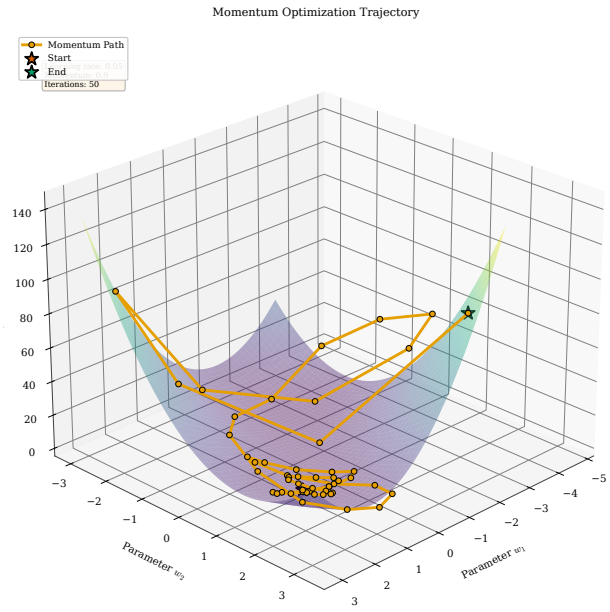


Figure 1.8: Momentum optimization trajectory on the same loss surface. The path is smoother than SGD, with the optimizer building velocity in the consistent downhill direction. Momentum accelerates progress along narrow valleys while damping oscillations across them.

ing a subtle but important flaw in the original formulation [3]. Standard L2 regularization adds a term $\lambda\|\boldsymbol{\theta}\|^2$ to the loss, but when combined with Adam's adaptive learning rates, this does not achieve the intended regularization effect—parameters receiving small gradients are barely regularized because the adaptive scaling reduces their updates. AdamW instead decouples weight decay from the gradient-based update, directly subtracting $\lambda \cdot \boldsymbol{\theta}$ from parameters at each step independently of the gradient magnitude. This decoupled formulation ensures all parameters experience proportional regularization regardless of their gradient magnitudes, improving generalization in large language models by preventing parameters from growing unboundedly large, which can lead to memorization rather than generalization. The typical configuration uses $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$, and weight decay between 0.01 and 0.1, though optimal hyperparameters depend on model size, training duration, and the specific dataset characteristics. AdamW has become the standard choice for transformer training, used in essentially all recent large language model development.

1.3 Learning Rate Schedules

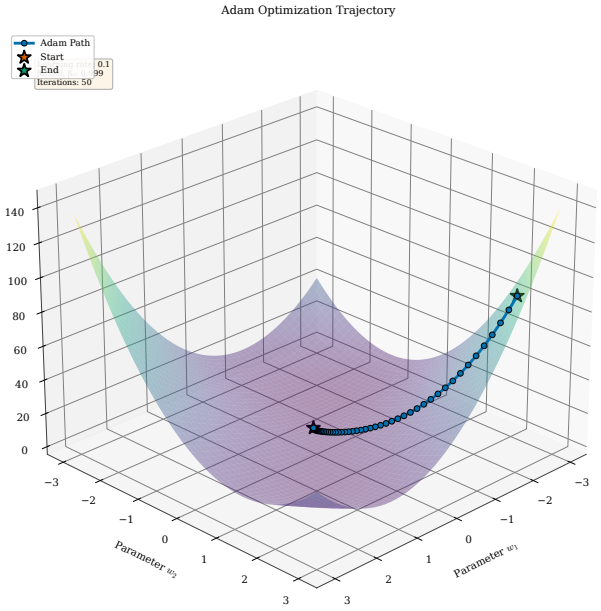


Figure 1.9: Adam optimization trajectory on the same loss surface. The adaptive learning rates allow Adam to take appropriately sized steps in each direction, avoiding both the oscillation of SGD and the potential overshooting of momentum. Adam typically converges faster than either alternative.

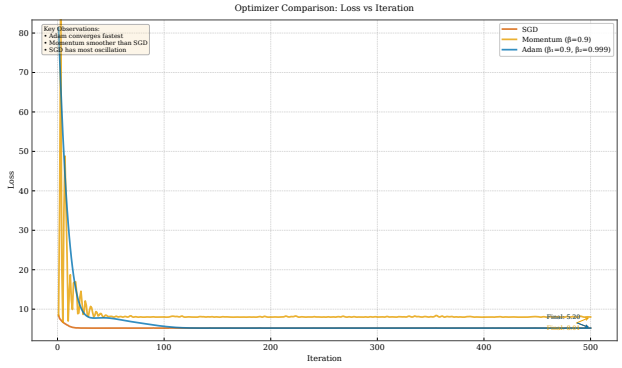


Figure 1.10: Loss curves comparing SGD, Momentum, and Adam on identical training data. Adam converges significantly faster and to a lower final loss. The adaptive learning rates and momentum combination enable rapid initial progress and stable fine-grained optimization near convergence.

The learning rate η is arguably the most important hyperparameter in neural network training, yet the optimal value changes throughout training as the model progresses through different phases of learning. Early in training, large learning rates enable rapid exploration and escape from poor initializations, allowing the model to quickly find promising regions of parameter space where representations begin to capture linguistic structure. Later in training, smaller learning rates enable fine-grained convergence to good solutions without overshooting optimal parameter values, refining representations to capture subtle patterns. Learning rate schedules formalize this intuition by defining how the learning rate changes over training steps, replacing a single fixed value with a dynamic trajectory through learning rate space that adapts to the model's needs at each training phase. The right schedule can mean the difference between training success and failure, between a model that achieves state-of-the-art performance and one that stagnates at suboptimal quality. Practitioners invest significant effort in schedule design and tuning, often running multiple experiments to find configurations that balance rapid early progress with stable late-stage convergence.

Warmup addresses training instability that occurs when using large learning rates from initialization, providing a critical safety mechanism for transformer training that prevents catastrophic early divergence and wasted compute resources during the expensive early stages of optimization. With randomly initialized parameters that have no meaningful structure and produce essentially random outputs, the first few gradient updates can be extremely noisy because the model has not yet learned meaningful representations of language, potentially pushing the model into poor regions of parameter space from which recovery is difficult or impossible even with extended subsequent training. Warmup starts with a very small learning rate near zero, allowing the model to establish reasonable representations and stable internal statistics across its many layers before gradually increasing to the target learning rate over thousands of steps in a controlled linear progression. Linear warmup is most common in practice because of its simplicity and empirical effectiveness: the learning rate at step t equals the maximum rate times t divided by the warmup steps for all steps less than the warmup duration, where the warmup period is typically between one and five percent of total training steps for large language models, though some configurations use longer warmup depending on model size and dataset characteristics. Research suggests that warmup is particularly important for transformers because the LayerNorm statistics and attention patterns need initial stabilization before the model can productively use large learning rates—without warmup, the softmax in attention can produce extremely peaked distributions that create vanishing gradients for non-attended tokens and unstable layer normalization statistics that oscillate rather than converging. The warmup duration depends on model size, with larger models generally requiring longer warmup

periods to allow their more numerous parameters to reach coherent configurations before aggressive optimization begins, since the coordination required across billions of parameters takes more updates to establish.

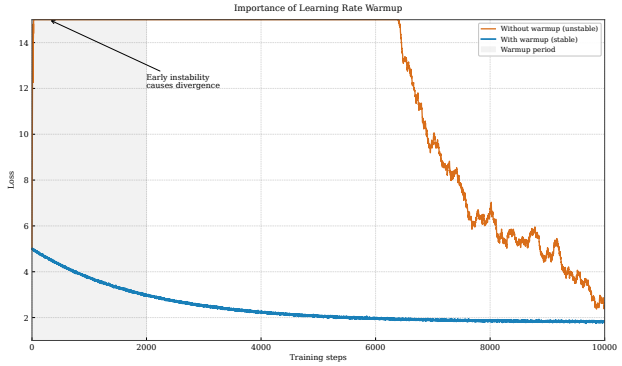


Figure 1.11: The importance of learning rate warmup. Without warmup (orange), training immediately uses a large learning rate, causing unstable loss and potential divergence. With warmup (blue), the learning rate gradually increases, allowing stable initialization before full-speed training.

After the warmup phase completes and the learning rate has reached its maximum value, the learning rate typically decays toward zero following a prescribed schedule, with the specific decay pattern significantly impacting final model quality and the efficiency of compute utilization throughout the remainder of the long training process. Cosine decay has become the dominant choice for language model training due to its smooth decay profile and strong empirical performance across many experiments conducted by major research laboratories and industry practitioners. The cosine schedule decreases the learning rate following a half-cosine curve, starting at the maximum value and smoothly approaching a small terminal value that is often zero or ten percent of the maximum. The cosine shape provides gradual decay early in training when the model is still learning coarse patterns and benefits from larger steps to make rapid progress, accelerating decay in the middle phase as representations stabilize and require more careful adjustment to refine learned patterns, then slowing again near the end to allow fine-tuning of learned representations without excessive oscillation that could undo progress. This pattern allows extended exploration in the productive middle phase while ensuring smooth convergence in the final phase, and empirical studies have shown cosine decay consistently outperforms alternatives across diverse model sizes and datasets, becoming the de facto standard for large language model training at leading research organizations. The original transformer paper [9] used inverse square root decay, where the learning rate decreases proportionally to the inverse square root of the training step, which decays more aggressively but continuously throughout training

without the distinct phases of cosine decay. Linear decay provides simpler implementation and reasonable results, though typically inferior to cosine decay on large-scale language modeling tasks where the smooth acceleration and deceleration of cosine decay better matches the optimization dynamics.

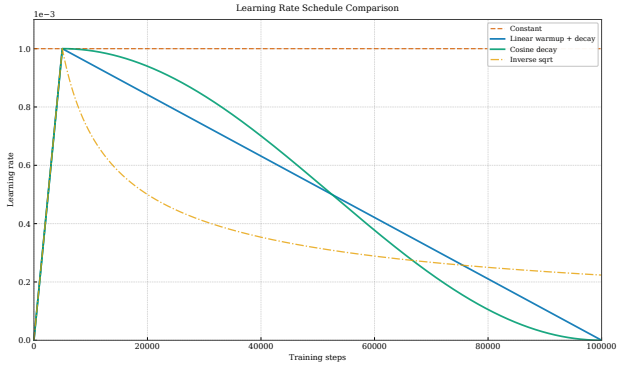


Figure 1.12: Comparison of learning rate schedules. Constant learning rate (gray) maintains the same value throughout. Cosine decay (blue) provides smooth reduction with accelerated decay mid-training. Inverse square root (green) decays continuously but less aggressively than cosine. All schedules include initial warmup.

The interaction between learning rate and training dynamics creates a complex optimization landscape where the schedule must carefully navigate between competing concerns at each training step. Too high a learning rate causes the loss to oscillate or diverge as the optimizer overshoots optimal parameter values, potentially destroying useful representations that took many steps to learn. Too low a learning rate wastes compute on tiny updates that barely improve the model, leaving performance gains on the table and extending training time unnecessarily. The optimal learning rate depends on batch size (larger batches allow larger rates), model size (larger models often need lower rates), and training phase (early exploration versus late convergence), making schedule design both art and science that requires empirical validation. Research has revealed important scaling relationships that provide guidance: larger batch sizes can use larger learning rates following the linear scaling rule [1], and larger models often benefit from lower learning rates due to their more complex loss landscapes. These relationships guide practitioners in adapting schedules when changing training configurations, though each new setting typically requires some experimental tuning to achieve optimal results.

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...”, training at different learning rates produces dramatically different outcomes visible in the loss curves for each token position.

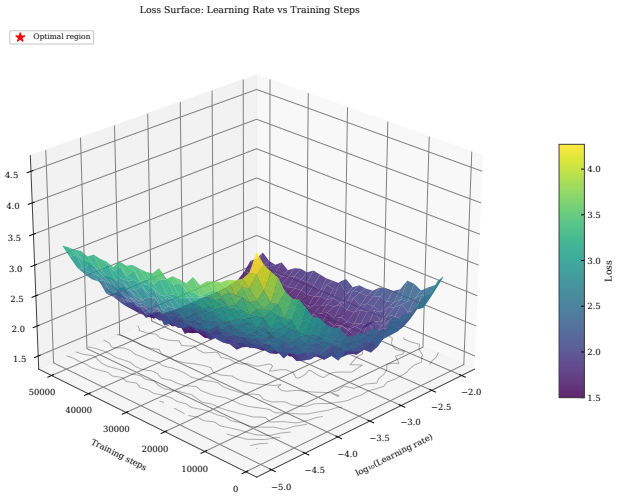


Figure 1.13: Learning rate effect on final loss, shown as a function of learning rate and training duration. There exists an optimal learning rate (ridge in the surface) that achieves lowest loss. Too small wastes steps without making progress; too large causes instability and worse final loss.

With an appropriately tuned learning rate, the model learns to predict technical vocabulary ("engineer," "remotely," "submitted," "code") within context, reducing loss on exactly the challenging tokens that carry semantic information about the software development domain. With too low a learning rate, the model improves slowly, achieving passable predictions for function words like "the" and "from" but struggling with content words even after extended training because parameter updates are too small to capture complex patterns. With too high a learning rate, the loss may decrease initially but then plateau or oscillate as the optimizer overshoots good configurations, never achieving the fine-grained predictions that distinguish good language models from mediocre ones. These dynamics illustrate why learning rate tuning demands careful attention despite advances in adaptive optimizers like Adam, which adapt per-parameter rates but still require appropriate global learning rate settings.

1.4 Batch Size and Gradient Accumulation

Figure 1.14: Loss curves on our running example sentence at three learning rates. The optimal learning rate (blue) achieves lowest final loss. Too low (green) converges slowly and plateaus at higher loss. Too high (orange) shows instability and worse final performance despite faster initial progress.

The batch size—the number of examples processed before a gradient update—profoundly influences training dynamics and final model quality in ways that interact with learning rate and model architecture. Larger batch sizes produce more accurate gradient estimates by averaging over more examples, reducing the variance that causes optimization noise and allows more consistent progress toward lower loss configurations. The gradient variance scales inversely with batch size following $\text{Var}[[]\nabla_{\boldsymbol{\theta}}\mathcal{L}] \propto 1/B$, meaning that doubling the batch size halves the gradient variance, providing more reliable descent directions at each step. However, larger batch sizes require proportionally more compute per step and empirical research suggests they can converge to sharper minima that potentially generalize worse to unseen text than the broader minima found with smaller batches. Modern language model training uses extremely large batch sizes, often millions of tokens processed simultaneously, to maximize hardware utilization on expensive GPU clusters and provide stable gradients at the scale of billions of parameters where optimization noise could otherwise cause significant training instability.

GPU memory constraints limit the batch size that can fit on a single device, particularly for large language models where parameters, gradients, optimizer states, and activations already consume most available memory on even the largest GPUs. Gradient accumulation provides an elegant solution: instead of updating parameters after each forward-backward pass, accumulate gradients across multiple microbatches before updating, effectively simulating larger batches within memory constraints. If we process k microbatches of size b before updating, the effective batch size is $B_{\text{eff}} = k \cdot b$, achieving the gradient averaging benefits of large batches without exceeding memory limits on any single forward-backward pass. The accumulated gradi-

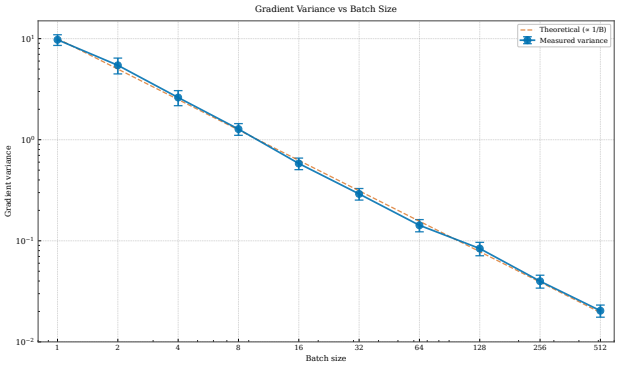


Figure 1.15: Gradient variance decreases with batch size following a $1/B$ relationship. Small batches produce noisy gradient estimates that cause optimization instability. Large batches provide accurate gradients but require more computation per update.

ent is mathematically equivalent to computing the gradient on a single large batch because gradients are linear—the sum of gradients equals the gradient of the sum. This equivalence enables practitioners to decouple batch size choices from hardware constraints, choosing batch sizes for optimal training dynamics while adjusting accumulation steps to fit available memory.

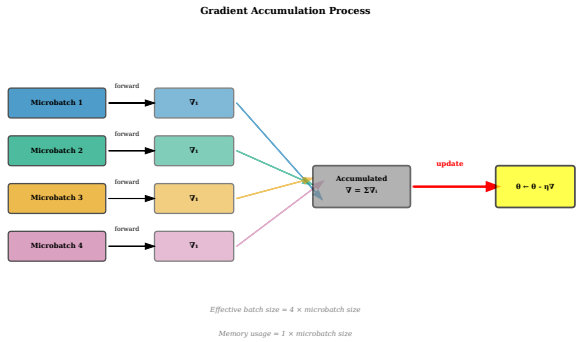


Figure 1.16: Gradient accumulation across four microbatches. Each microbatch computes gradients independently, which are then summed (or averaged) before the parameter update. This achieves large effective batch sizes while fitting within GPU memory constraints.

Memory consumption in language model training divides among several components that must all fit within GPU memory simultaneously: model parameters, gradients for backpropagation, optimizer states for algorithms like Adam, and activations stored from the forward pass. For a 7B parameter model using AdamW with mixed precision training [4], parameters require

14GB (7B parameters times 2 bytes in fp16), gradients require another 14GB in fp16, and Adam's moment estimates require 28GB (two full-precision fp32 copies of all parameters for the first and second moment running averages), totaling 56GB before considering activations. Activation memory scales with batch size and sequence length as the model must store intermediate values for every layer to enable gradient computation during backpropagation, potentially consuming tens of gigabytes for long sequences with large batches. Techniques like activation checkpointing trade compute for memory by recomputing activations during the backward pass rather than storing them, reducing memory requirements significantly at the cost of roughly 30% additional forward computation per step. Mixed precision training stores activations and some parameters in 16-bit format while maintaining critical computations like gradient accumulation in 32-bit, further reducing memory footprint while maintaining numerical stability through careful handling of precision-sensitive operations.

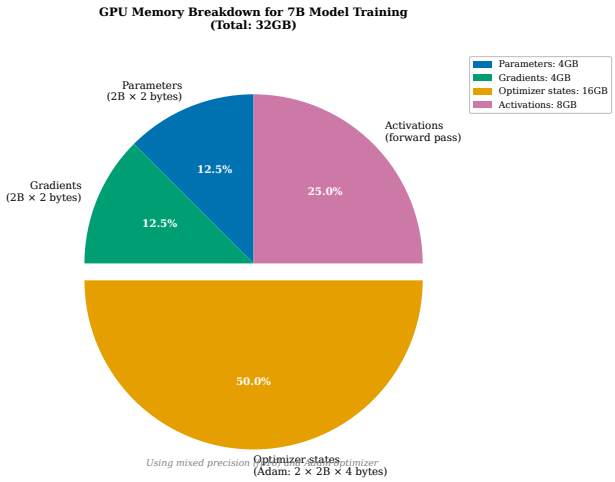


Figure 1.17: GPU memory breakdown for a 7B parameter language model using AdamW. Optimizer states (Adam's moment estimates) consume the largest portion at 50%, followed by gradients and parameters at 25% each. Activations add additional overhead scaling with batch size.

The critical batch size represents the point beyond which increasing batch size no longer improves training efficiency [7], marking a fundamental transition in optimization dynamics. Below this threshold, doubling the batch size roughly halves the number of steps needed to reach a target loss because gradient noise is the limiting factor, making larger batches efficiently convert additional compute into faster progress. Above this threshold, the noise reduction from larger batches provides diminishing returns—the optimization is no longer gradient-noise-limited but rather progresses at a pace determined by the loss landscape geometry and the inherent difficulty of finding

better solutions. The critical batch size depends on the dataset characteristics, model architecture, and training phase, typically increasing as training progresses and the model approaches convergence where gradients become smaller and more correlated across samples. Understanding this threshold helps practitioners choose batch sizes that maximize training efficiency given their compute budget, avoiding both the inefficiency of too-small batches that waste time on noisy updates and too-large batches that waste compute on redundant gradient information.

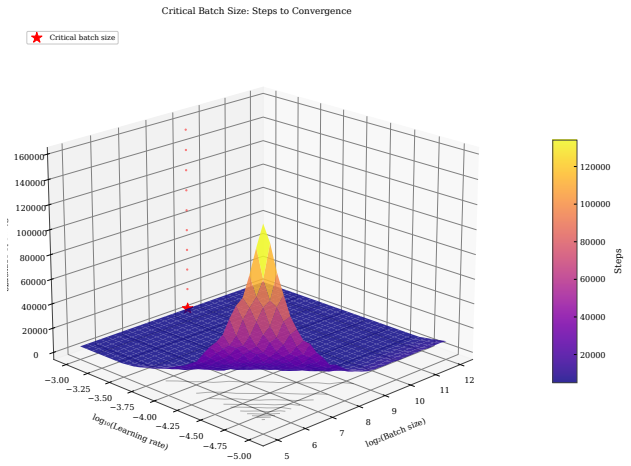


Figure 1.18: Steps to convergence as a function of batch size and learning rate. Below the critical batch size (left region), increasing batch size linearly reduces required steps. Above it (right region), further batch size increases provide diminishing returns. The optimal learning rate scales with batch size up to the critical point.

1.5 Distributed Training

Training large language models requires compute resources far exceeding what a single GPU can provide, necessitating distributed training across hundreds or thousands of accelerator devices connected by high-speed networks. The challenge is to efficiently parallelize training while maintaining mathematical equivalence to single-GPU training (or accepting well-understood approximations that do not compromise model quality or convergence). Modern large language model training runs on clusters containing thousands of GPUs, consuming megawatts of power and costing millions of dollars in compute time over training runs that span weeks. Three complementary parallelism strategies—data parallelism, model parallelism (also called tensor parallelism), and pipeline parallelism—can be combined to scale training

to arbitrary model sizes and cluster configurations, enabling the training of models with hundreds of billions of parameters that would be impossible on any single device. The choice of parallelism configuration depends on model size and architecture, hardware topology including interconnect bandwidth, and the communication costs of each strategy, requiring careful engineering to achieve high utilization of expensive hardware resources.

Data parallelism replicates the entire model on each device, with different devices processing different data batches simultaneously to multiply training throughput. After each device computes gradients on its local batch, an AllReduce operation synchronizes gradients across all devices, ensuring each device has the identical average gradient before updating parameters identically, maintaining model consistency across the cluster. The effective batch size scales with the number of devices: $B_{\text{eff}} = B_{\text{per\_device}} \times N_{\text{devices}}$, enabling the large batch sizes that stabilize training at scale. Data parallelism is simple to implement and scales efficiently when the model fits on a single device, requiring only gradient synchronization rather than more complex activation communication. The communication overhead is the AllReduce operation, which for ring-based implementations requires transferring $2(N-1)/N \cdot |\boldsymbol{\theta}|$ data per device, approaching $2|\boldsymbol{\theta}|$ for large device counts where $|\boldsymbol{\theta}|$ is the parameter count. For 7B parameter models using 16-bit precision, this means roughly 28GB of communication per step, which modern high-bandwidth interconnects like NVLink (600 GB/s) or InfiniBand (400 Gb/s) handle in tens of milliseconds.

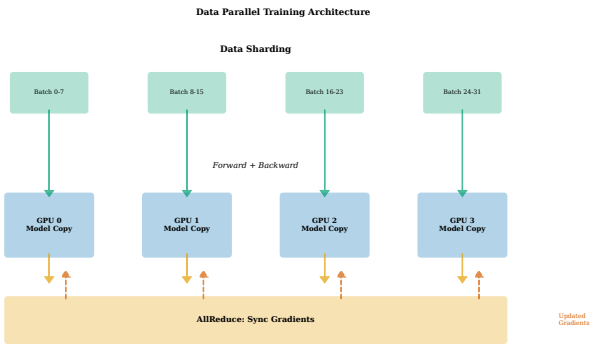


Figure 1.19: Data parallel training across four GPUs. Each GPU holds a complete model replica and processes a different data batch. After computing local gradients, AllReduce synchronizes gradients across all devices, after which each device performs identical parameter updates.

Model parallelism (also called tensor parallelism) splits individual layers across devices when a single layer is too large for one GPU's memory, enabling training of models larger than any single accelerator can hold. The most common approach partitions the large weight matrices in transformer

layers: the attention weight matrices ($\mathbf{W}_Q$, $\mathbf{W}_K$, $\mathbf{W}_V$, $\mathbf{W}_O$) and feed-forward weight matrices ($\mathbf{W}_1$, $\mathbf{W}_2$), distributing both memory requirements and computation. For row-wise partitioning, each device computes a portion of the matrix multiplication independently, then an AllGather operation collects results to reconstruct the full output tensor. For column-wise partitioning, inputs are scattered to devices, each computes partial outputs on its portion of the weight matrix, then results are combined with AllReduce to sum partial products. Megatron-LM [8] demonstrated efficient tensor parallelism for transformers, achieving near-linear scaling to 8-way tensor parallelism within a single node where NVLink provides high-bandwidth interconnection, making the communication overhead negligible compared to computation time.

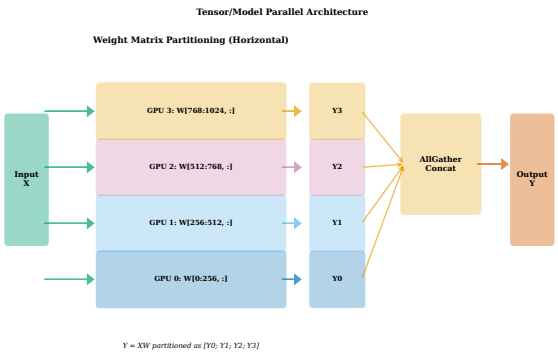


Figure 1.20: Tensor parallel partitioning of a single transformer layer across four GPUs. Weight matrices are split column-wise or row-wise, with each GPU computing a portion of the matrix multiplication. AllReduce or All-Gather operations combine partial results.

Pipeline parallelism assigns different layers to different devices, with activations flowing between devices as data moves through the model from first layer to last [5]. The challenge is the "pipeline bubble": with simple sequential execution, most devices sit idle while waiting for activations from earlier stages or gradients from later stages, wasting expensive compute resources. Microbatching amortizes this idle time by interleaving multiple microbatches through the pipeline, keeping all stages busy processing different portions of the batch. With m microbatches and p pipeline stages, the bubble ratio is $(p-1)/(m+p-1)$, approaching zero as m grows large, though memory constraints limit how many microbatches can be in flight simultaneously. The 1F1B (one forward, one backward) schedule minimizes memory requirements by limiting how many microbatches are simultaneously in flight, performing one backward pass for each forward pass completed. Pipeline parallelism has lower communication volume than tensor parallelism since only activations (not weights) cross device boundaries, but introduces complexity in gradi-

ent synchronization across stages and requires careful batch scheduling to maintain correctness.

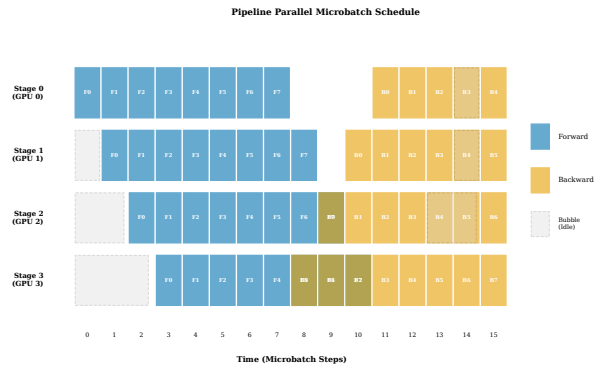


Figure 1.21: Pipeline parallel execution with four stages and multiple microbatches. Forward passes (F) and backward passes (B) are interleaved across microbatches to minimize pipeline bubble (gray regions). The bubble represents unavoidable idle time at the start and end of each training step.

Modern large-scale training combines all three parallelism strategies in a 3D parallelism configuration, carefully allocating each parallelism dimension to match hardware topology and communication characteristics. A typical setup might use 8-way tensor parallelism within each node (utilizing high-bandwidth NVLink connections between GPUs), 8-way pipeline parallelism across nodes (utilizing lower-bandwidth but sufficient network interconnects), and data parallelism across the remaining dimension to scale total throughput. For a 1024-GPU cluster, this could be organized as: tensor parallel groups of 8 GPUs on each node, pipeline parallel groups spanning 8 nodes (64 GPUs), and 16 data parallel replicas processing different batches. The choice of dimensions depends on model architecture (wide layers benefit from tensor parallelism), hardware topology (fast intra-node connections favor tensor parallelism), and the communication costs of each parallelism strategy (data parallelism has highest volume, pipeline parallelism has lowest). Frameworks like Megatron-LM, DeepSpeed, and PyTorch FSDP automate much of this configuration while exposing knobs for expert tuning to achieve optimal hardware utilization.

ZeRO (Zero Redundancy Optimizer) [6] provides an orthogonal approach to memory efficiency by partitioning optimizer states, gradients, and optionally parameters across data parallel ranks rather than replicating them on every device. Standard data parallelism replicates all state on every device, wasting memory for identical copies, but ZeRO recognizes that each device only needs its portion of parameters during the forward pass and its portion of gradients during the update. ZeRO Stage 1 partitions optimizer states (Adam's moment estimates), reducing memory by roughly 4× for Adam

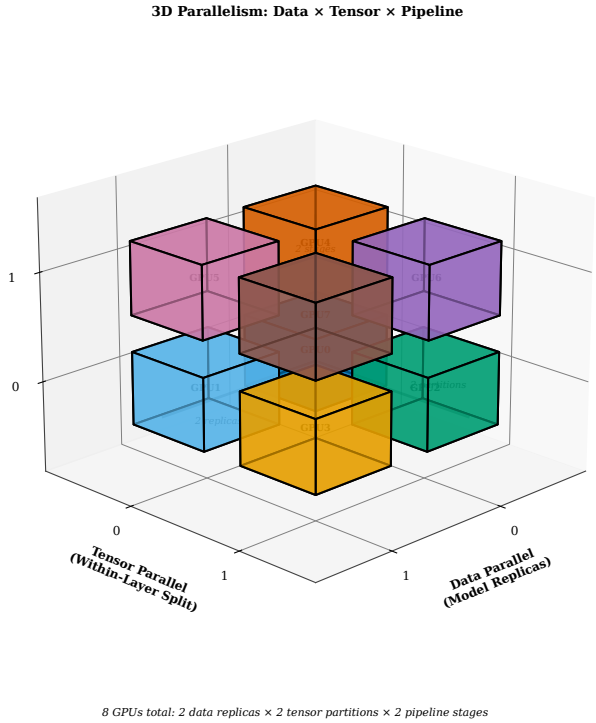


Figure 1.22: 3D parallelism combining data, tensor, and pipeline dimensions. Each axis represents a parallelism strategy, with the full cube covering all GPUs. A 2×2×2 configuration requires 8 GPUs, scaling to thousands of GPUs by extending each dimension.

since optimizer states dominate memory for large models. Stage 2 adds gradient partitioning, eliminating gradient replication, and Stage 3 partitions parameters as well, enabling training of arbitrarily large models on limited hardware by distributing all model state across the cluster. The tradeoff is increased communication volume as devices must gather parameters before forward passes and scatter gradients before updates, though careful implementation overlaps communication with computation to hide latency and maintain high throughput.

1.6 Checkpointing and Training Stability

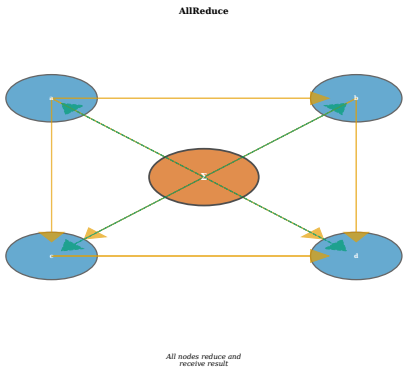


Figure 1.23: Collective communication patterns used in distributed training. AllReduce sums data across all devices and distributes the result. AllGather collects data from all devices onto each device. ReduceScatter combines reduction and scattering for memory-efficient gradient distribution.

Training large language models requires days or weeks of continuous computation on expensive hardware, making checkpoint management essential for recovering from failures and enabling interrupted training to resume without losing significant progress. Hardware failures, power outages, preemption in cloud environments, and software bugs can interrupt training at any point during these extended runs, making robust checkpointing infrastructure critical for successful large-scale training operations. A checkpoint must capture all state necessary to exactly reproduce training continuation: model parameters, optimizer state (Adam's first and second moment estimates), learning rate scheduler state including current step count, data loader position including which samples have been seen, and random number generator states for reproducibility across all devices. For a 7B parameter model with AdamW, checkpoints can exceed 100GB including optimizer states, requiring efficient storage and loading strategies that can write to fast parallel file systems without blocking training for too long. Periodic checkpointing every few hundred or thousand steps provides recovery points while limiting storage costs and checkpoint overhead.

Gradient clipping provides a crucial stability mechanism by limiting the magnitude of gradient updates, preventing the occasional anomalously large gradient from destabilizing training. When gradients exceed a threshold τ, they are scaled down while preserving their direction: $\nabla_{\boldsymbol{\theta}}\mathcal{L} \leftarrow \nabla_{\boldsymbol{\theta}}\mathcal{L} \cdot \min(1, \tau/\|\nabla_{\boldsymbol{\theta}}\mathcal{L}\|)$, where $\|\nabla_{\boldsymbol{\theta}}\mathcal{L}\|$ is the global norm computed across all parameters in the model. This clipping prevents catastrophically large updates that could push the model into poor parameter regions from which recovery is difficult, potentially undoing thousands of steps of progress and wasting significant compute time. The clipping threshold is typically set between 0.5 and 1.0 for transformer language models, chosen to rarely activate

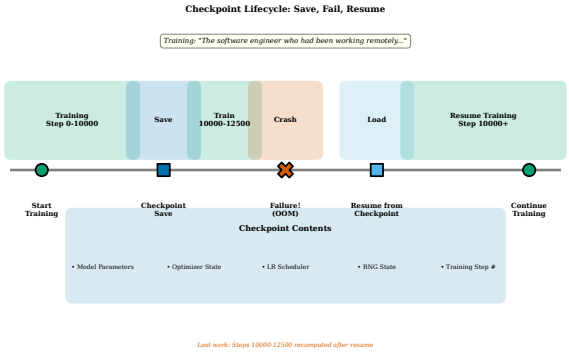


Figure 1.24: The checkpoint lifecycle in training our running example. Training proceeds normally until a failure occurs (hardware error, preemption, or loss spike). The system recovers by loading the most recent checkpoint and resuming training from that state, losing only the steps since the last checkpoint.

during normal training while providing a safety net for occasional outlier batches that produce anomalously large gradients due to unusual data or numerical issues. Gradient clipping by global norm (clipping based on the norm across all parameters jointly) is more common than per-parameter clipping, as it preserves the relative magnitudes of gradients across different parameter groups, maintaining the intended update direction while reducing overall magnitude when necessary.

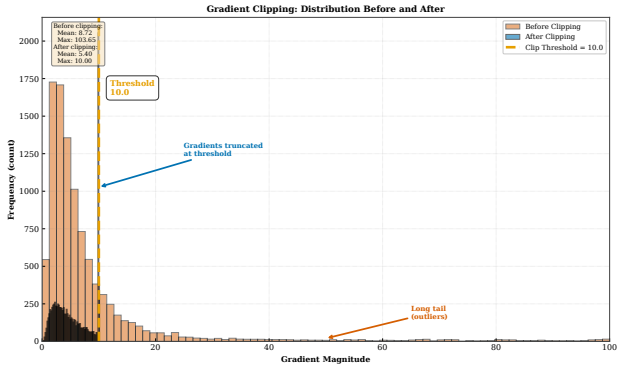


Figure 1.25: Gradient magnitude distribution before and after clipping. Before clipping (blue), occasional large gradients create a heavy tail in the distribution. After clipping (orange), gradients exceeding the threshold are scaled down, truncating the tail and preventing extreme updates.

Loss spikes—sudden large increases in loss during otherwise stable training—are a notorious phenomenon in large-scale training that can range

from minor disturbances to catastrophic failures. Spikes can arise from bad batches containing unusual data distributions, numerical instability in attention computation when some attention weights become extremely large, or unlucky optimizer state configurations where momentum accumulates in problematic directions. Minor spikes often resolve spontaneously as training continues and the optimizer moves past the problematic region, but severe spikes can permanently damage the model by pushing parameters into configurations that produce poor gradients, requiring rollback to a checkpoint to recover. Practitioners monitor training closely and develop spike detection systems that automatically pause training and potentially roll back when loss exceeds historical bounds by a configurable threshold. Post-spike analysis often reveals specific problematic batches that can be removed from training data, or numerical issues (like attention scores approaching infinity) that motivate architectural changes like attention temperature scaling or precision adjustments in sensitive computations.

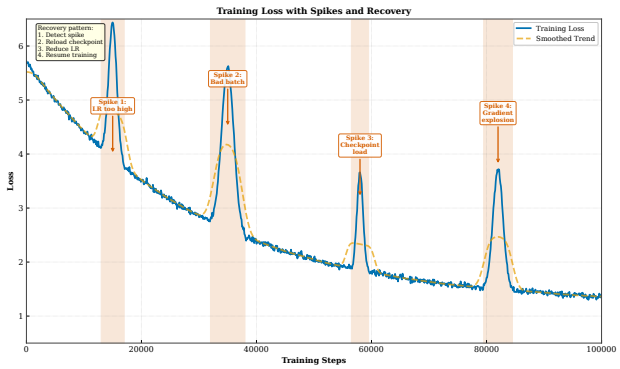


Figure 1.26: Loss spikes during training and recovery. The loss decreases smoothly during normal training but occasionally spikes due to bad batches, numerical issues, or optimizer instability. Most spikes resolve naturally, but severe spikes may require rollback to a previous checkpoint.

Reproducibility at scale is challenging but important for scientific validation, debugging failed runs, and comparing training configurations fairly across experiments in research settings. Exact reproducibility requires fixing all random seeds (for initialization, data shuffling, and dropout), using deterministic algorithms throughout the software stack, and maintaining identical hardware configurations including GPU generation and driver versions. In practice, floating-point non-associativity means that different parallelization strategies or even different execution orders within a single strategy can produce slightly different results because the order of floating-point operations affects rounding errors due to limited precision. These differences compound over billions of operations during training, producing measurably different models from identical configurations even when all seeds are

fixed. Many teams accept "statistical reproducibility"—achieving similar loss curves and comparable downstream performance across runs with different random seeds—rather than bit-exact reproducibility, which would require sacrificing performance for determinism by using slower algorithms and disabling certain hardware optimizations that introduce non-determinism into the computation.

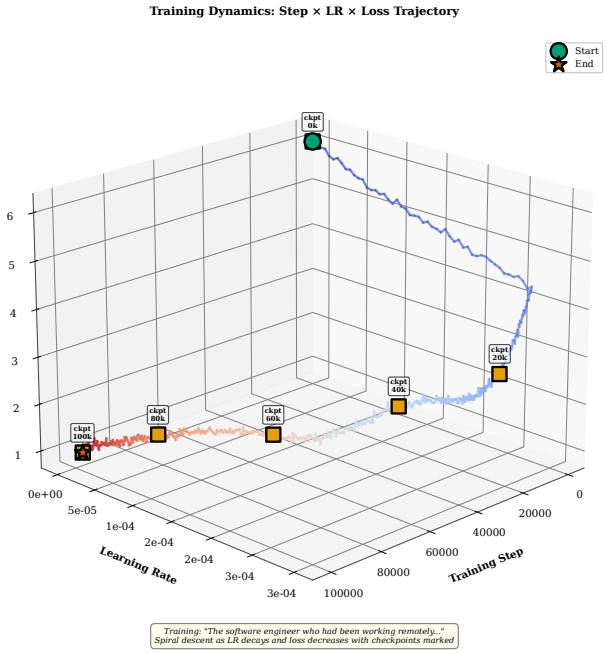


Figure 1.27: Training dynamics visualized in three dimensions: training steps, learning rate (decreasing via schedule), and loss. The trajectory spirals downward as both learning rate and loss decrease, with checkpoints marked along the path. This view captures the complete training trajectory that produces a capable language model.

1.7 Context Representation in Training

Training fundamentally shapes how the model represents context for next-word prediction, transforming random initial weights into sophisticated linguistic processing systems capable of understanding language. Before training, the transformer's attention patterns are random, attending to tokens without semantic purpose, and producing essentially uniform probability distributions over $\mathcal{V}$ regardless of input context. Through millions of gradient updates driven by cross-entropy loss, attention heads learn to implement linguistically meaningful operations: some heads attend to syntactic dependencies like subject-verb agreement across long distances, others to semantic

associations between related concepts, and still others to positional patterns that capture word order and relative positions. The loss signal propagates information about which context representations enable accurate prediction, gradually sculpting the attention patterns and feed-forward transformations that convert raw token sequences into rich, predictive context representations that capture meaning. By the end of training, the model has learned to extract and compose contextual information in ways that support accurate next-word prediction across diverse linguistic contexts and domains.

How This Chapter Represents Context:
Training determines how effectively the model aggregates context for next-word prediction:

- Cross-entropy loss measures whether context representations support accurate prediction
- Gradients flow through attention mechanisms, updating how context tokens contribute to predictions
- Learning rate schedules control how quickly context representations evolve during training
- Large batch sizes provide stable gradients for learning complex context patterns
- Distributed training enables learning from massive corpora that reveal diverse context-prediction relationships

The training process can be understood as learning a mapping from context to probability distribution over the vocabulary $\mathcal{V}$, refining this mapping through billions of gradient updates. Early in training, this mapping is nearly uniform—all words in $\mathcal{V}$ are roughly equally likely regardless of context, with $P(w)$ approximately $1/|\mathcal{V}|$ for every word. As training progresses, the model learns to condition on increasingly subtle contextual cues: not just the previous word, but long-range dependencies spanning dozens or hundreds of tokens, semantic constraints from world knowledge, and syntactic expectations from grammatical structure. For our running example "The software engineer who had been working remotely from the mountain cabin for several months finally submitted the code that would...", an untrained model assigns equal probability to all vocabulary words, including nonsensical completions. After training, the model learns that technical verbs ("revolutionize," "transform," "change") are appropriate completions, that the subject is likely code-related given the software engineering context, and that the sentence structure suggests a consequential outcome worth describing. This learned context sensitivity is exactly what training optimizes,

and the loss function directly rewards distributions where $P(w)$ for correct words approaches 1.0.

We can now predict better because:

- Cross-entropy loss directly optimizes the model's ability to assign high probability to correct next words
- Adam with warmup and cosine decay provides stable, efficient convergence from random initialization to capable language model
- Gradient accumulation and large batch sizes provide stable gradients for learning complex linguistic patterns
- Distributed training parallelism enables learning from trillion-token corpora on thousand-GPU clusters

Next: Chapter examines how these training techniques scale to Large Language Models, exploring the architectures, datasets, and training recipes that produce GPT-4, Claude, and LLaMA—models that demonstrate emergent capabilities far beyond next-word prediction.

Exercises

1. **Cross-Entropy Calculation.** Consider a vocabulary of three words $\{a, b, c\}$. If the model predicts probabilities $(0.7, 0.2, 0.1)$ and the true word is b, calculate the cross-entropy loss for this single prediction. How does the loss change if the prediction were $(0.1, 0.7, 0.2)$?

2. **Perplexity Interpretation.** A language model achieves a cross-entropy loss of 2.5 on a test corpus. Calculate the perplexity. Interpret this value in terms of "effective vocabulary size." If the vocabulary contains 50,000 words, what fraction of the vocabulary has the model effectively eliminated through its predictions?

3. **Gradient Derivation.** Derive the gradient of cross-entropy loss with respect to the logits $\mathbf{z}$ when the output layer uses softmax. Show that the gradient simplifies to $\text{softmax}(\mathbf{z}) - \mathbf{y}$, where $\mathbf{y}$ is the one-hot target vector.

4. **Adam vs SGD Analysis.** Starting from the same initialization, SGD with learning rate 0.01 and Adam with learning rate 0.001 both train on the same data. After 1000 steps, Adam achieves lower loss. Explain

three mechanisms by which Adam might achieve this advantage despite using a smaller nominal learning rate.

5. **Learning Rate Warmup.** Explain why transformers are particularly sensitive to learning rate warmup. Consider the role of LayerNorm statistics and attention patterns at initialization. How might training without warmup cause the model to reach an unrecoverable state?

6. **Batch Size Memory.** Calculate the optimizer memory requirements for a 13B parameter model using AdamW in mixed precision (fp16 parameters and gradients, fp32 optimizer states). How does ZeRO Stage 2 (optimizer state and gradient partitioning) reduce this across 8 GPUs?

7. **Gradient Accumulation Equivalence.** Prove mathematically that gradient accumulation over k microbatches of size b produces the same parameter update as a single batch of size kb (assuming no numerical precision differences). Under what conditions might the results differ in practice?

8. **Pipeline Bubble Ratio.** For a 4-stage pipeline with 16 microbatches, calculate the bubble ratio. How many microbatches would be needed to reduce the bubble to less than 5% of total time? What practical constraints limit microbatch count?

9. **Data vs Model Parallelism.** A 7B parameter model is trained on a cluster of 64 GPUs. Compare two configurations: (a) 64-way data parallelism, (b) 8-way tensor parallelism $\times$ 8-way data parallelism. What are the tradeoffs in communication volume, memory efficiency, and implementation complexity?

10. **(Advanced) Critical Batch Size.** Using the gradient noise scale framework, derive a relationship between critical batch size and model size. Why do larger models tend to have larger critical batch sizes? How does this affect the compute efficiency of training larger models?

11. **(Advanced) AllReduce Complexity.** For ring-based AllReduce with N devices and message size M, derive the total communication time assuming bandwidth B and latency L per message. Under what conditions does the algorithm achieve near-optimal bandwidth utilization?

12. **(Advanced) Checkpoint Frequency Optimization.** Model the tradeoff between checkpoint frequency and expected training time considering: checkpoint overhead c, mean time between failures μ, and recovery time r. Derive the optimal checkpoint interval that minimizes expected training time.

Bibliography

[1] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

[2] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations (ICLR)*, 2015.

[3] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations (ICLR)*, 2019.

[4] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed precision training. 2018.

[5] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, et al. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–15, 2021.

[6] Samyam Rajbhandari, Jeff Rasley, Olatunji Rber, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–16, 2020.

[7] Christopher J Shallue, Jaehoon Lee, Joseph Antognini, Jascha Sohl-Dickstein, Roy Frostig, and George E Dahl. Measuring the effects of data parallelism on neural network training. *Journal of Machine Learning Research*, 20(112):1–49, 2019.

[8] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-

billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[9] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *NeurIPS*, 30, 2017.

Index