Chapter 2: Mathematical Foundations

Learning Objectives

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

1. Apply the chain rule of probability to decompose the joint probability of a word sequence into a product of conditional probabilities.
2. Derive the maximum likelihood estimate for language model parameters and explain its connection to cross-entropy minimization.
3. Compute entropy, cross-entropy, and KL divergence for discrete distributions and interpret their meaning in the context of language modeling.
4. Calculate perplexity from cross-entropy and explain why lower perplexity indicates a better language model.

If language is a sequence of choices, how do we assign a probability to an entire sentence? Chapter 1 established that the central task of language modeling is predicting the next word, and we traced this idea from Shannon's original experiments to modern transformer architectures. But we left the mathematics informal. We spoke of probabilities without defining them precisely, mentioned estimation without deriving the formulas, and invoked perplexity as an evaluation metric without explaining what it measures or why. This chapter corrects those omissions. We develop, from first principles, the mathematical toolkit that every subsequent chapter depends on: probability theory applied to word sequences, maximum likelihood estimation, information theory, and gradient-based optimization. These are not background topics imported from a prerequisite course and applied mechanically. They are the direct machinery of prediction. Probability is the formalism that lets us assign a numerical score to each possible next word. Entropy measures how uncertain those predictions are. Cross-entropy measures how far our model's predictions diverge from reality. Perplexity exponentiates that gap into a number we can interpret and compare. And optimization is the process by which we improve our predictions given data. By the end of this chapter, the reader will possess every mathematical concept needed to build, train, and evaluate the language models that occupy the remainder of this book.

2.1 Probability and Conditional Probability

2.1.1 Probability over Vocabularies

Consider five words: the, cat, sat, on, mat. Call this set V — the vocabulary. A language model operates over such a fixed, finite set of symbols. In practice, V might contain the 30,000 most frequent words in an English corpus, or subword tokens produced by a byte-pair encoding algorithm, but for now our $|V| = 5$ vocabulary is enough to illustrate the machinery. The vocabulary defines the sample space of our probability distribution. A language model is then a function that assigns a probability $P(w \mid \text{context})$ to each word $w \in V$, given some preceding context. These probabilities must satisfy two axioms: non-negativity, $P(w \mid \text{context}) \geq 0$ for every w, and normalization, $\sum_{w \in V} P(w \mid \text{context}) = 1$. Any function satisfying these two conditions is a valid probability distribution over the vocabulary, and therefore a valid (if possibly terrible) language model.

Students often assume that the vocabulary and the set of possible sentences are the same kind of object, but they differ in a fundamental way. The vocabulary V is finite — it has a fixed number of elements, whether that number is five or fifty thousand. The set of possible sentences, however, is countably infinite, because sentences can be of any length. From a vocabulary of five words, we can construct five one-word sentences, twenty-five two-word sentences, one hundred twenty-five three-word sentences, and so on without bound. A language model must assign probabilities to all of these sentences, which is why we need the chain rule decomposition developed in Section 2.1.3 — it reduces the infinite problem of assigning probabilities to sentences of arbitrary length into the finite problem of assigning conditional probabilities over the vocabulary at each position. A useful analogy is to think of the vocabulary as a deck of cards. The deck is finite and fixed. But a sequence of draws from the deck can be arbitrarily long. The analogy breaks down because a deck of cards is drawn without replacement, while words can repeat — indeed, in our toy vocabulary, "the" appears twice in the sentence "the cat sat on the mat." Nevertheless, the core insight holds: a finite vocabulary generates an infinite set of possible sequences, and we need machinery to handle this.

Figure 2.1: Probability Distributions over Vocabulary

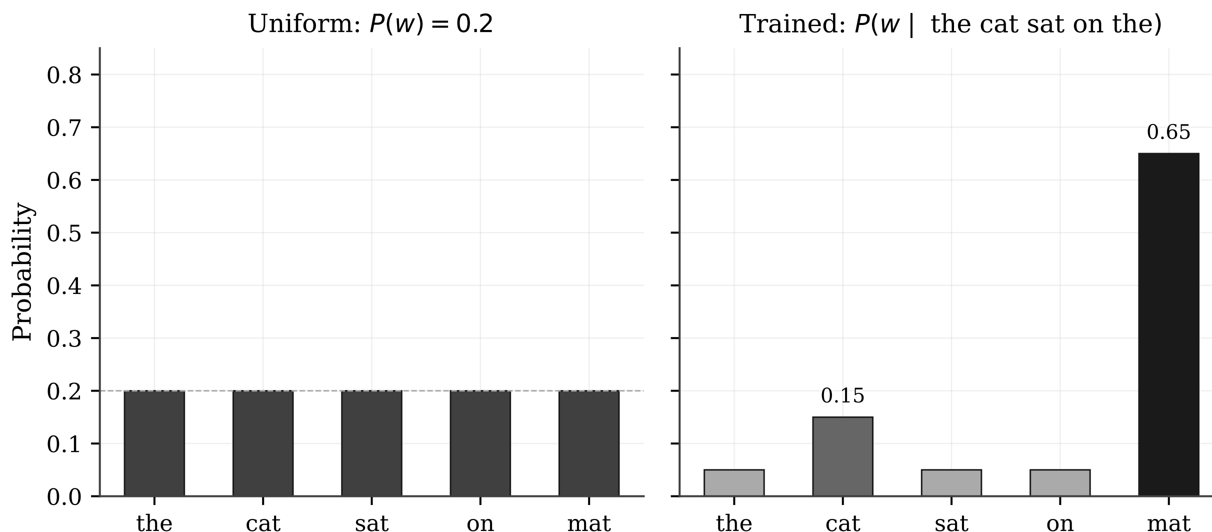

Figure 1: Figure 2.1 – Probability distributions over vocabulary

To ground these ideas in concrete numbers, consider our five-word vocabulary $V = \{\text{the}, \text{cat}, \text{sat}, \text{on}, \text{mat}\}$ and a small corpus consisting of one sentence: "the cat sat on the mat." A uniform model assigns $P(w) = 1/|V| = 1/5 = 0.2$ to every word regardless of context. This is a valid probability distribution — it satisfies non-negativity and normalization — but it is a poor language model because it treats "mat" as equally likely as "cat" after the context "the cat sat on the." A better model would concentrate probability mass on plausible continuations. For instance, after the context "the cat sat on the," a reasonable model might assign $P(\text{mat} \mid \text{the cat sat on the}) = 0.65$, $P(\text{cat} \mid \text{the cat sat on the}) = 0.15$, and distribute the remaining 0.20 among the other three words. Both models are valid distributions over V; the second one is simply better at predicting what comes next. Measuring how much better — quantifying the gap between a good model and a bad one — requires tools we have not yet introduced. But we have been vague about one thing: what exactly do we mean by the probability of two words appearing

together?

2.1.2 Joint and Conditional Distributions

The probability of a single word given a context is the fundamental building block, but the object we ultimately want to compute is the probability of an entire sentence — the joint probability $P(w_1, w_2, \ldots, w_n)$ over a sequence of n words. Joint probability measures the likelihood that word w_1 appears in position one and word w_2 appears in position two and so on through position n. For our toy corpus containing the single sentence "the cat sat on the mat," the joint probability $P(\text{the}, \text{cat}, \text{sat}, \text{on}, \text{the}, \text{mat})$ is the probability that this specific six-word sequence occurs. A language model that assigns high joint probability to grammatical, meaningful sentences and low joint probability to nonsensical word salad is doing its job.

The bridge from joint probability to conditional probability is Bayes' rule in its simplest form. For two words, the joint probability factors as $P(w_1, w_2) = P(w_1) \cdot P(w_2 \mid w_1)$, where $P(w_2 \mid w_1)$ is the conditional probability of the second word given the first. Students frequently confuse $P(w_2 \mid w_1)$ with $P(w_1 \mid w_2)$. The two are not the same. $P(\text{cat} \mid \text{the})$ asks: given that we have seen "the," how likely is "cat" to follow? $P(\text{the} \mid \text{cat})$ asks: given that we have seen "cat," how likely is "the" to follow? In English, "the cat" is far more common than "cat the," so we would expect $P(\text{cat} \mid \text{the})$ to be substantially larger than $P(\text{the} \mid \text{cat})$. Language models predict left to right — they estimate $P(w_t \mid w_1, \ldots, w_{t-1})$, the probability of the next word given all preceding words — so the directionality of conditioning matters enormously.

To make this concrete, suppose our tiny corpus tells us that $P(\text{the}) = 2/6$ (since "the" appears twice in six words) and $P(\text{cat} \mid \text{the}) = 1/2$ (since "the" is followed by "cat" once and by "mat" once, out of its two occurrences). Then the joint probability of the two-word sequence is $P(\text{the}, \text{cat}) = P(\text{the}) \cdot P(\text{cat} \mid \text{the}) = (2/6)(1/2) = 1/6 \approx 0.167$. This factoring of a joint probability into a marginal times a conditional is the seed from which the chain rule grows.

2.1.3 The Chain Rule for Language

If we had to choose the single equation that defines language modeling, it would be this one. The chain rule of probability generalizes the two-word factoring from Section 2.1.2 to sequences of arbitrary length. For any sequence of n words, the joint probability decomposes as:

$$P(w_1, w_2, \ldots, w_n) = \prod_{t=1}^{n} P(w_t \mid w_1, w_2, \ldots, w_{t-1}) \tag{2.1}$$

This equation is not an approximation. It is not a modeling assumption. It is a mathematical identity — it holds for any probability distribution over sequences, regardless of how that distribution was constructed. What makes this equation the heart of language modeling is that each factor $P(w_t \mid w_1, \ldots, w_{t-1})$ is exactly the next-word prediction task: given all the words that have come before, what is the probability of the word at position t? A language model that can estimate each of these conditional probabilities accurately can compute the probability of any sentence of any length, simply by multiplying the factors together. A common misconception is that the chain rule introduces some approximation or simplification. It does not. The approximation comes later, in Chapter 3, when we decide to truncate the conditioning context — for example, approximating $P(w_t \mid w_1, \ldots, w_{t-1})$ by $P(w_t \mid w_{t-1})$, conditioning only on the immediately preceding word. That is the Markov assumption, and it is a modeling choice with real consequences. The chain rule itself

is exact. Understanding the distinction between the exact decomposition (Equation 2.1) and the approximate estimation of its factors is essential for everything that follows in this book.

To see the chain rule in action, we compute the probability of the four-word sentence "the cat sat on" from our running example. Applying Equation (2.1), we obtain $P(\text{the}, \text{cat}, \text{sat}, \text{on}) = P(\text{the}) \cdot P(\text{cat} \mid \text{the}) \cdot P(\text{sat} \mid \text{the}, \text{cat}) \cdot P(\text{on} \mid \text{the}, \text{cat}, \text{sat})$. Suppose our corpus-derived estimates give us $P(\text{the}) = 0.33$, $P(\text{cat} \mid \text{the}) = 0.50$, $P(\text{sat} \mid \text{the}, \text{cat}) = 0.80$, and $P(\text{on} \mid \text{the}, \text{cat}, \text{sat}) = 0.70$. Then the joint probability is $0.33 \times 0.50 \times 0.80 \times 0.70 = 0.0924$. Each factor has a concrete linguistic interpretation: $P(\text{the}) = 0.33$ says that about one-third of sentences in our tiny world begin with "the"; $P(\text{cat} \mid \text{the}) = 0.50$ says that half the time "the" is followed by "cat" rather than by some other word; and so on. The product gives us the probability that all four of these choices happen in sequence. Four numbers, multiplied together, giving us the probability of a sentence. That is the entire mechanism.

2.1.4 Independence Assumptions and the Markov Property

The chain rule in Equation (2.1) is exact, but it creates a practical problem: to estimate $P(w_t \mid w_1, \ldots, w_{t-1})$, we need to condition on the entire history of preceding words. For the tenth word in a sentence, we condition on nine words. For the hundredth word, we condition on ninety-nine. The number of possible contexts grows exponentially with the length of the history, and for long contexts, we will never observe enough training data to estimate these conditional probabilities reliably. This is where independence assumptions enter. The most important independence assumption in language modeling is the Markov property, which states that the probability of the current word depends only on a fixed-length window of preceding words, rather than on the entire history. A first-order (bigram) Markov assumption gives $P(w_t \mid w_1, \ldots, w_{t-1}) \approx P(w_t \mid w_{t-1})$, conditioning only on the immediately preceding word. A second-order (trigram) assumption gives $P(w_t \mid w_1, \ldots, w_{t-1}) \approx P(w_t \mid w_{t-2}, w_{t-1})$, conditioning on the two preceding words. More generally, an $(n-1)$-th order Markov assumption conditions on the previous $n-1$ words.

It is tempting to think that the Markov assumption is always a bad idea — that throwing away history must always hurt prediction. In practice, this is not the case. For many word pairs, the immediate context carries the vast majority of the predictive information. After "the," the word "cat" is much more likely than "philosophy" regardless of what came five words earlier. The Markov assumption trades statistical precision for statistical reliability: we estimate fewer parameters, each from more data, at the cost of ignoring potentially relevant long-range context. The question that drives the entire arc of this textbook — from n-gram models in Chapter 3, through recurrent neural networks in Chapter 5, to the transformer architecture in Chapter 8 — is how much context is enough. n-gram models use a fixed window of two to five words. Recurrent networks use a theoretically unbounded but practically limited hidden state. Transformers process the entire available context up to a maximum sequence length. Each approach represents a different answer to the fundamental tradeoff between context length and estimation reliability, and the Markov property is where that tradeoff first appears. The analogy of weather prediction captures this tradeoff: predicting tomorrow's weather based on the last two days (a second-order Markov model) works surprisingly well, because recent conditions carry most of the relevant information. Predicting based on weather records from ten years ago adds little value for most practical forecasting purposes, even though in principle the distant past is not truly irrelevant.

2.2 Maximum Likelihood Estimation

2.2.1 The Likelihood Function

How do we learn a language model's parameters from data?

We now have the chain rule decomposition (Equation 2.1) telling us how to factor the probability of a sentence into a product of conditional probabilities. The next question is: where do the numerical values of these conditional probabilities come from? The answer, for classical language models, is maximum likelihood estimation (MLE). The idea is simple and powerful: find the parameter values that make the observed training data as probable as possible. Suppose we have a training corpus $\mathcal{D}$ consisting of N words: $w_1, w_2, \ldots, w_N$. The likelihood of the corpus under a language model with parameters θ is the probability that the model assigns to the observed sequence: $L(\theta) = P(\mathcal{D} \mid \theta) = \prod_{t=1}^{N} P(w_t \mid w_1, \ldots, w_{t-1}; \theta)$. The likelihood is a function of the parameters θ, not a probability distribution over θ. Students confuse likelihood with probability, and the distinction matters. A probability distribution assigns probabilities to different possible observations given fixed parameters. The likelihood function assigns scores to different parameter values given fixed (observed) data. The data are constant; the parameters vary. MLE says: among all possible parameter settings, choose the one that maximizes $L(\theta)$ — the one that makes our training corpus most probable. If our model assigns $P(\text{the}) = 0.1$ and $P(\text{cat} \mid \text{the}) = 0.3$, the likelihood of observing "the cat" is $0.1 \times 0.3 = 0.03$. MLE searches for parameter values that make this product as large as possible across the entire training corpus. This principle — find the parameters that best explain the observed data — is one of the oldest and most widely used ideas in statistics, and it is the foundation of both classical n-gram models and the training of modern neural language models.

2.2.2 Log-Likelihood and Its Properties

The likelihood function $L(\theta)$ is a product of potentially millions of probabilities, each less than one. Multiplying a million numbers that are each less than one produces an astronomically small result. In our experience, the numerical underflow example is the moment students truly appreciate why we work with log-probabilities. Consider a modest 100-word sentence where each word has probability 0.1 under the model. The likelihood is $0.1^{100} = 10^{-100}$. A 32-bit floating-point number can represent values as small as approximately 10^{-38}; a 64-bit number can go down to roughly 10^{-308}. Our 100-word likelihood of 10^{-100} already pushes against the limit of 32-bit arithmetic, and for a realistic corpus of millions of words, the product underflows to exactly zero in any standard floating-point representation. The model would appear to assign zero probability to the entire corpus — not because the corpus is impossible, but because the arithmetic cannot represent the answer.

The solution is to work with the logarithm. Because the logarithm is a monotonically increasing function, maximizing $L(\theta)$ is equivalent to maximizing $\log L(\theta)$, and the logarithm converts the product into a sum: $\log L(\theta) = \sum_{t=1}^{N} \log P(w_t \mid w_{<t}; \theta)$. Our problematic 0.1^{100} becomes $100 \times \log(0.1) = 100 \times (-2.30) = -230.0$ in natural logarithm, or -100.0 in base-10 logarithm. The value -230 is a perfectly ordinary floating-point number. No underflow, no special handling, no loss of precision. This is why every practical implementation of language model training works with log-probabilities rather than raw probabilities, and why every practical training loop prints a loss value, not a likelihood value. A natural question arises at this point: does working with log-probabilities change the optimization in any way beyond numerical stability? The answer is no — the parameters that maximize the log-likelihood are exactly the parameters that maximize the likelihood, because the logarithm is strictly monotone. The log transformation changes the

numerical representation but not the mathematical optimum. It is a computational convenience that happens to be computationally essential.

2.2.3 MLE as Counting

In 1948, the same year Shannon published his information theory, the basic idea of estimating probabilities by counting frequencies was already well established in statistics. One might not realize that the simple act of counting word occurrences and dividing by totals is itself the solution to a constrained optimization problem. For n-gram language models, MLE takes an especially elegant form: the maximum likelihood estimate of a conditional probability is simply the ratio of counts. For a bigram model, the MLE of the conditional probability $P(w_t \mid w_{t-1})$ is:

$$\hat{P}(w_t \mid w_{t-1}) = \frac{C(w_{t-1}, w_t)}{C(w_{t-1})} \tag{2.2}$$

where $C(w_{t-1}, w_t)$ is the number of times the bigram (w_{t-1}, w_t) appears in the training corpus, and $C(w_{t-1})$ is the number of times the word w_{t-1} appears. This formula generalizes to any n-gram order: the MLE of $P(w_t \mid w_{t-n+1}, \ldots, w_{t-1})$ is the count of the full n-gram divided by the count of the $(n-1)$-gram prefix. The formula looks obvious — perhaps even trivial — but it is the result of a rigorous optimization. We can derive it by setting up the constrained optimization problem: maximize the log-likelihood $\sum_t \log P(w_t \mid w_{t-1})$ subject to the constraint that $\sum_w P(w \mid w_{t-1}) = 1$ for each context w_{t-1}. Introducing a Lagrange multiplier λ for each normalization constraint, we form the Lagrangian, take the derivative with respect to $P(w_t \mid w_{t-1})$, set it to zero, and solve. The result is precisely Equation (2.2): the count ratio.

To apply this formula to our running example, recall the corpus "the cat sat on the mat." We count bigrams: $C(\text{the}, \text{cat}) = 1$, $C(\text{the}, \text{mat}) = 1$, $C(\text{cat}, \text{sat}) = 1$, $C(\text{sat}, \text{on}) = 1$, $C(\text{on}, \text{the}) = 1$. The unigram counts are $C(\text{the}) = 2$, $C(\text{cat}) = 1$, $C(\text{sat}) = 1$, $C(\text{on}) = 1$, $C(\text{mat}) = 1$. The MLE bigram probabilities are then $\hat{P}(\text{cat} \mid \text{the}) = C(\text{the}, \text{cat})/C(\text{the}) = 1/2 = 0.50$ and $\hat{P}(\text{mat} \mid \text{the}) = 1/2 = 0.50$. Each of the remaining bigrams has an MLE probability of $1/1 = 1.0$: $\hat{P}(\text{sat} \mid \text{cat}) = 1.0$, $\hat{P}(\text{on} \mid \text{sat}) = 1.0$, and $\hat{P}(\text{the} \mid \text{on}) = 1.0$. These deterministic estimates — where a word is predicted with probability one given its predecessor — look suspiciously perfect. They are. They are also the direct consequence of having only one training sentence, and they illustrate a serious problem that we address next.

2.2.4 Overfitting and the Need for Smoothing

Here is a test: compute $P(\text{the cat on the mat})$ using our bigram model. We need $P(\text{the}) \cdot P(\text{cat} \mid \text{the}) \cdot P(\text{on} \mid \text{cat}) \cdot P(\text{the} \mid \text{on}) \cdot P(\text{mat} \mid \text{the})$. The first factor is 0.33. The second is 0.50. The third — $P(\text{on} \mid \text{cat})$ — requires the bigram "cat on," which does not appear in our training corpus "the cat sat on the mat." So $C(\text{cat}, \text{on}) = 0$, and $\hat{P}(\text{on} \mid \text{cat}) = 0/1 = 0$. The product is zero. One missing bigram killed it. The model declares a perfectly reasonable sentence impossible. Perplexity fares even worse: $\log(0) = -\infty$, so the perplexity of the test set becomes infinite.

This is the fatal flaw of MLE on finite data: it assigns zero probability to anything it has not seen, and a single zero is enough to destroy an entire evaluation. The problem does not go away with more data. Zipf's law tells us that a large fraction of possible word sequences are rare, appearing only once or not at all in any finite corpus. English has at least a hundred thousand common words, producing at least ten billion possible bigrams. Most of them will never appear in any training set

of reasonable size, and the MLE assigns zero probability to all of them. The solution is smoothing — redistributing some probability mass from observed n-grams to unobserved ones. What would happen if we gave every unseen bigram a small probability — say, 10^{-6}? Would that be enough, or would it distort the probabilities we already estimated? Every language model in active use, from the simplest bigram model to GPT-4, employs some mechanism to prevent zero probabilities, whether through explicit smoothing, subword tokenization that avoids out-of-vocabulary words, or the implicit smoothing provided by neural network generalization.

2.3 Information Theory Essentials

2.3.1 Entropy of a Language

In 1948, Claude Shannon, a mathematician at Bell Labs, published "A Mathematical Theory of Communication," a paper that single-handedly created the field of information theory. We confess that Shannon's original paper remains one of the most readable foundational documents in all of science — a 55-page tour de force that introduced entropy, channel capacity, and the source coding theorem, all in prose that a determined undergraduate can follow. Shannon needed a measure of how much "information" a random variable carries — a number that captures the uncertainty or surprise associated with its outcomes. He showed that the unique function satisfying a small set of natural axioms (continuity, monotonicity in the number of equally likely outcomes, and additivity for independent events) is what we now call Shannon entropy:

$$H(P) = -\sum_{x \in V} P(x) \log_2 P(x) \tag{2.3}$$

where the sum ranges over all outcomes x in the sample space (for us, all words in the vocabulary V), and $\log_2$ gives the entropy in bits. The convention is that $0 \log_2 0 = 0$, which is justified by the limit $\lim_{p \to 0^+} p \log p = 0$. Entropy has a concrete interpretation: it is the expected number of bits needed to encode a sample drawn from P using an optimal code. If P is uniform over $|V|$ outcomes, each outcome is maximally surprising, and the entropy reaches its maximum value of $H = \log_2 |V|$. If P concentrates all mass on a single outcome, there is no uncertainty and $H = 0$. Languages sit between these extremes. A fair coin, with $P(\text{heads}) = P(\text{tails}) = 0.5$, has entropy $H = 1$ bit. A biased coin with $P(\text{heads}) = 0.99$ has entropy $H = -0.99 \log_2 0.99 - 0.01 \log_2 0.01 \approx 0.081$ bits — almost no uncertainty, because we can almost always predict the outcome correctly.

Students frequently confuse entropy (a property of a single distribution P) with cross-entropy (a comparison between two distributions P and Q). The distinction is critical and cannot be overemphasized. Entropy $H(P)$ measures the irreducible uncertainty in the language itself — how much inherent randomness exists in word sequences, given perfect knowledge of the true distribution. Cross-entropy $H(P, Q)$, which we define in Section 2.3.2, measures the cost of using an imperfect model Q to predict data generated by P. The gap between them — the KL divergence — is precisely the penalty we pay for using the wrong model. A language model cannot beat the entropy of the true distribution; the best it can do is match it. This fact, which follows from Gibbs' inequality, provides an absolute lower bound on the performance of any language model.

To compute entropy on our running example, consider the unigram distribution over $V = \{\text{the}, \text{cat}, \text{sat}, \text{on}, \text{mat}\}$ derived from the corpus "the cat sat on the mat." The empirical probabilities are $P(\text{the}) = 2/6$, $P(\text{cat}) = 1/6$, $P(\text{sat}) = 1/6$, $P(\text{on}) = 1/6$, $P(\text{mat}) = 1/6$. The

entropy is $H(P) = -(2/6)\log_2(2/6) - 4\times(1/6)\log_2(1/6) = -0.333\times(-1.585) - 4\times 0.167\times(-2.585) = 0.528 + 1.724 = 2.252$ bits. The maximum entropy for a five-word vocabulary is $\log_2 5 = 2.322$ bits (the uniform distribution). Our corpus distribution, which slightly favors "the," has entropy $2.252/2.322 = 97\%$ of the maximum — not far from uniform, because our vocabulary is tiny and our corpus is short. Real English has much lower relative entropy, as Shannon's experiments demonstrated.

Figure 2.2: Entropy Comparison

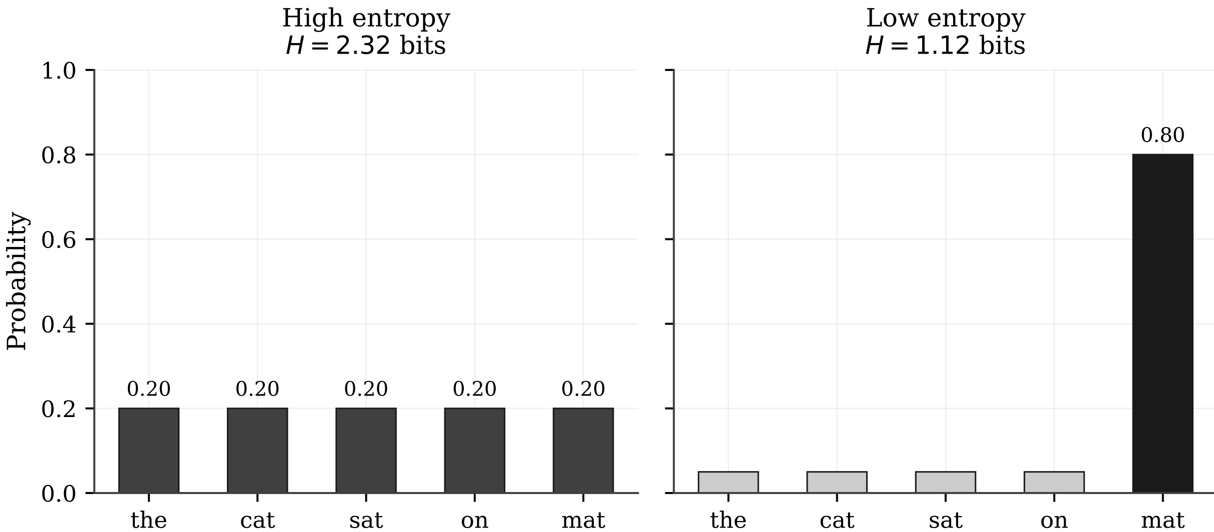

Figure 2: Figure 2.2 – Entropy visualization comparing high-entropy and low-entropy distributions

```
import numpy as np

# English letter frequencies (26 letters + space, from published tables)
freqs = np.array([
    0.0651, 0.0124, 0.0217, 0.0349, 0.1041,  # a-e
    0.0197, 0.0158, 0.0492, 0.0558, 0.0009,  # f-j
    0.0050, 0.0331, 0.0202, 0.0564, 0.0596,  # k-o
    0.0137, 0.0008, 0.0497, 0.0515, 0.0729,  # p-t
    0.0225, 0.0082, 0.0171, 0.0014, 0.0145,  # u-y
    0.0007, 0.1820                             # z, space
])
freqs = freqs / freqs.sum()  # ensure normalization

# Shannon entropy in bits
H = -np.sum(freqs * np.log2(freqs))
H_max = np.log2(len(freqs))  # maximum entropy (uniform)
ratio = H / H_max

print(f"Entropy of English letters: {H:.3f} bits")
print(f"Maximum entropy (uniform):  {H_max:.3f} bits")
```

```python
print(f"Efficiency ratio:          {ratio:.3f}")
# Output: H ~ 4.08 bits, H_max = 4.75 bits, ratio ~ 0.86
```

Shannon himself estimated the entropy rate of English — the entropy per character when taking context into account — through a series of clever experiments where he asked human subjects to guess the next letter in a text. His 1951 follow-up paper estimated the entropy rate of printed English at between 0.6 and 1.5 bits per character, with a best guess around 1.0 to 1.3 bits per character [Shannon, 1951]. The maximum entropy rate, assuming 27 equiprobable symbols (26 letters plus space), is $\log_2 27 \approx 4.75$ bits per character. Shannon's estimate implies that roughly 75 percent of written English is redundant — predictable from context. This estimate has proved remarkably durable: Brown et al. [1992] refined it to approximately 1.75 bits per character using n-gram models, and Radford et al. [2019] reported that GPT-2 achieves approximately 0.93 bits per character on a standard benchmark, suggesting that large neural models approach human-level prediction. The gap between maximum entropy and true entropy is precisely what language models exploit.

```python
import numpy as np
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt

# Entropy vs. temperature for a 5-word vocabulary
logits = np.array([2.0, 1.0, 0.5, 0.1, 0.01])
temperatures = np.linspace(0.1, 5.0, 100)
entropies = []

for T in temperatures:
    scaled = logits / T
    scaled -= scaled.max()  # numerical stability
    probs = np.exp(scaled) / np.exp(scaled).sum()
    H = -np.sum(probs * np.log2(probs + 1e-12))
    entropies.append(H)

H_max = np.log2(len(logits))
plt.figure(figsize=(8, 5))
plt.plot(temperatures, entropies, 'b-', linewidth=2)
plt.axhline(y=H_max, color='r', linestyle='--', label=f'H_max = {H_max:.2f} bits')
plt.xlabel('Temperature T')
plt.ylabel('Entropy (bits)')
plt.title('Entropy vs. Temperature')
plt.legend()
plt.savefig('entropy_vs_temperature.png', dpi=150)
print(f"H_max = {H_max:.2f} bits. Plot saved.")
```

Sidebar: Shannon's Entropy and the English Language

Claude Shannon's estimate of English entropy is one of the most cited results in all of computational linguistics, and its history reveals how deeply information theory and language modeling are intertwined. In his landmark 1948 paper, Shannon calculated the maximum entropy rate of English at approximately 4.75 bits per character, assuming 27

equiprobable symbols (the 26 letters of the alphabet plus the space character). This maximum entropy corresponds to a language where every character is equally likely and independent of all others — a language with no structure whatsoever. Shannon recognized that real English is vastly more structured. To estimate the true entropy rate, he devised what is now called the "Shannon guessing game": human subjects were shown a sequence of characters from a text and asked to guess the next character. From the number of guesses required, Shannon derived upper and lower bounds on the entropy rate. His 1951 paper reported an upper bound of approximately 1.5 bits per character and estimated the true rate at around 1.0 to 1.3 bits per character. This means that roughly three-quarters of written English is redundant — predictable from the surrounding context. The estimate was refined over subsequent decades. Brown, Della Pietra, Mercer, Della Pietra, and Lai [1992] used n-gram language models trained on large corpora to estimate an upper bound of about 1.75 bits per character, a number that served as a benchmark for language modeling performance throughout the 1990s and 2000s. What makes Shannon's result so remarkable is its durability: modern neural language models have driven the measured upper bound below 1.0 bits per character, approaching but never surpassing Shannon's theoretical lower bound. GPT-2, when evaluated on standard English text, achieves approximately 0.93 bits per character — a value that suggests these models capture the vast majority of the statistical structure of English. The fundamental insight, which has held for over seven decades, is that natural language is highly redundant, and this redundancy is precisely what makes prediction possible.

2.3.2 Cross-Entropy

How do we measure the cost of using the wrong model? Entropy tells us how uncertain a language is under the true distribution. But we never know the true distribution — we only have a model Q that approximates it. Cross-entropy measures the cost of using our model Q to predict data generated by the true distribution P:

$$H(P, Q) = -\sum_{x \in V} P(x) \log_2 Q(x) \tag{2.4}$$

The cross-entropy $H(P, Q)$ is the expected number of bits needed to encode a sample from P using a code optimized for Q. If our model Q matches the true distribution P perfectly, then $H(P, Q) = H(P)$ — the cross-entropy equals the entropy, and no bits are wasted. If our model Q differs from P, the cross-entropy exceeds the entropy: $H(P, Q) > H(P)$. This excess is the penalty we pay for using the wrong model, and it equals the KL divergence, as we show in Section 2.3.3. The cross-entropy can never be less than the entropy; this follows from Gibbs' inequality, which we state without proof: for any two distributions P and Q over the same sample space, $H(P, Q) \geq H(P)$, with equality if and only if $P = Q$. We state Gibbs' inequality without proof because its derivation requires Jensen's inequality applied to the concave logarithm function, which is an exercise at the end of this chapter (Exercise 4).

One might reasonably assume that minimizing cross-entropy is just one possible training objective for language models, and that alternatives like accuracy might work equally well. This assumption is incorrect, and the reasons are worth examining carefully. Cross-entropy is the standard loss function for training language models — from n-grams to transformers — and its connection to MLE is

not accidental. Maximizing the log-likelihood $\sum_t \log Q(w_t \mid w_{<t})$ is mathematically equivalent to minimizing the cross-entropy $H(P_{\text{data}}, Q)$, where P_{data} is the empirical distribution of the training corpus. This equivalence means that every language model trained by MLE is implicitly minimizing cross-entropy, whether the practitioners think of it that way or not.

Figure 2.3: Cross-Entropy Comparison

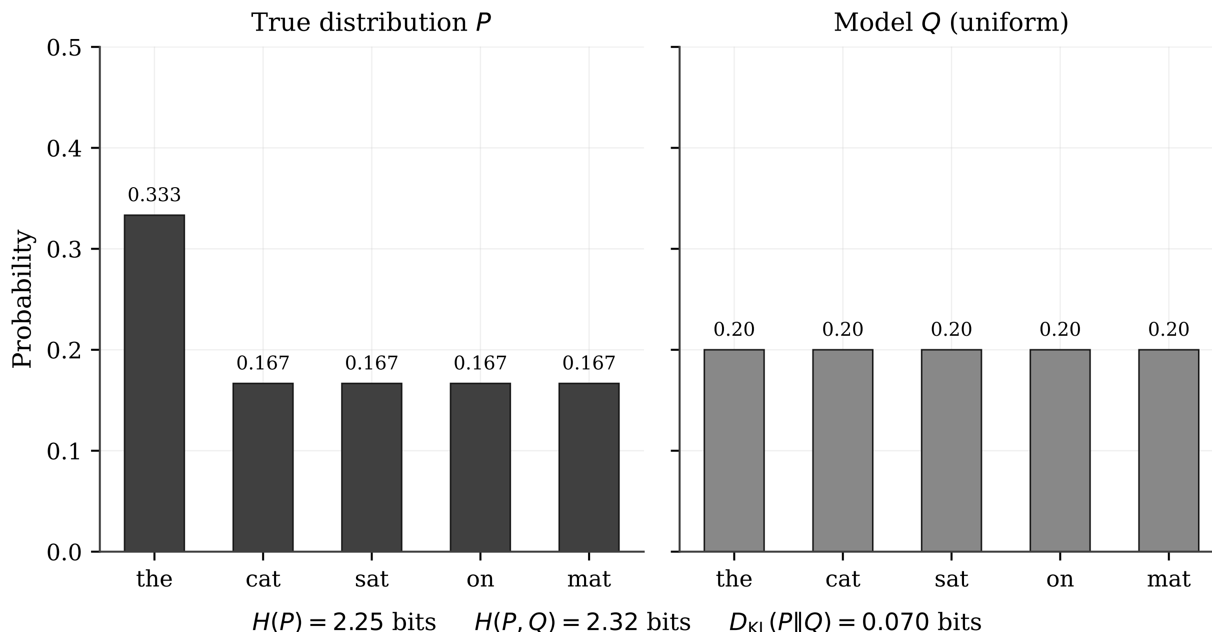

Figure 3: Figure 2.3 – Cross-entropy comparison

To make cross-entropy concrete, we return to our running example. The true unigram distribution from our corpus is $P(\text{the}) = 2/6$, $P(\text{cat}) = P(\text{sat}) = P(\text{on}) = P(\text{mat}) = 1/6$. Suppose our model is the uniform distribution $Q(w) = 1/5$ for all w. The cross-entropy is $H(P, Q) = -(2/6)\log_2(1/5) - 4 \times (1/6)\log_2(1/5) = -(2/6)(-2.322) - (4/6)(-2.322) = 2.322$ bits. This equals $\log_2 5$, which makes sense: the uniform model uses the maximum number of bits regardless of the true distribution. Compare this with the entropy $H(P) = 2.252$ bits computed in Section 2.3.1. The difference, $2.322 - 2.252 = 0.070$ bits, is the KL divergence between P and Q — the bits wasted by using the uniform model instead of the true distribution.

Sidebar: Why Cross-Entropy, Not Accuracy?

A natural question that students raise repeatedly is why language models are trained with cross-entropy loss rather than accuracy — the simple fraction of correctly predicted next words. The answer involves three distinct arguments, each sufficient on its own. The first argument is that accuracy is coarse. If the true next word is "cat" and the model assigns $P(\text{cat}) = 0.49$ and $P(\text{dog}) = 0.51$, accuracy counts this as wrong. If the model assigns $P(\text{cat}) = 0.001$ and $P(\text{armadillo}) = 0.999$, accuracy also counts this as wrong. Both predictions are equally penalized, even though the first is far closer to correct than the second. Cross-entropy, by contrast, penalizes the second prediction far more severely: $-\log_2(0.001) = 9.97$ bits versus $-\log_2(0.49) = 1.03$ bits. Cross-entropy

cares about the magnitude of the mistake, not just whether the top prediction was right. The second argument is computational: accuracy involves an argmax operation, which is not differentiable. Gradient-based optimization requires a differentiable loss function, and cross-entropy, which involves only logarithms and sums, is smooth everywhere the model assigns non-zero probability. The third argument is information-theoretic: cross-entropy is a *proper scoring rule*, meaning that the expected score is uniquely minimized when the model distribution Q equals the true distribution P [Gneiting and Raftery, 2007]. Accuracy is not proper — a model can maximize accuracy without recovering the true distribution. These three properties — sensitivity to prediction confidence, differentiability, and properness — make cross-entropy the unique appropriate training objective for language models, which is why every major language model from n-grams to GPT-4 uses it.

2.3.3 KL Divergence and Its Asymmetry

KL divergence, named after Solomon Kullback and Richard Leibler who introduced it in 1951 [Kullback and Leibler, 1951], provides a precise measure of the gap between two probability distributions. For a true distribution P and a model distribution Q, the KL divergence is:

$$D_{\mathrm{KL}}(P\|Q) = \sum_{x \in V} P(x) \log_2 \frac{P(x)}{Q(x)} \tag{2.5}$$

The KL divergence has a clean decomposition: $D_{\mathrm{KL}}(P\|Q) = H(P,Q) - H(P)$. It is the cross-entropy minus the entropy — exactly the extra bits wasted by using model Q instead of the true distribution P. KL divergence is always non-negative ($D_{\mathrm{KL}}(P\|Q) \geq 0$, a consequence of Gibbs' inequality), and it equals zero if and only if $P = Q$. This makes it a measure of how much Q differs from P, where zero means perfect agreement and larger values mean greater divergence. A natural but incorrect intuition is that KL divergence is a distance metric, like Euclidean distance. It is not. KL divergence fails two of the three requirements for a distance metric: it is asymmetric ($D_{\mathrm{KL}}(P\|Q) \neq D_{\mathrm{KL}}(Q\|P)$ in general) and it does not satisfy the triangle inequality. The asymmetry has practical significance that directly affects language model training.

The asymmetry of KL divergence matters because the two directions penalize different kinds of errors. In the "forward" direction, $D_{\mathrm{KL}}(P\|Q)$, the sum is weighted by $P(x)$, so the divergence is large when P assigns substantial probability to some outcome x but Q assigns it very little probability. This direction is "mode-covering": it forces Q to place mass everywhere P does, preventing the model from assigning zero probability to events that actually occur. In the "reverse" direction, $D_{\mathrm{KL}}(Q\|P)$, the sum is weighted by $Q(x)$, penalizing the model for placing mass where the true distribution does not. This direction is "mode-seeking": it encourages Q to concentrate on the modes of P even if it misses some of them. Language model training minimizes the forward KL divergence $D_{\mathrm{KL}}(P_{\mathrm{data}}\|Q_{\mathrm{model}})$, which is equivalent to minimizing cross-entropy. This choice ensures that the model does not assign zero probability to events that appear in the data — precisely the behavior we want, given the catastrophic consequences of zero probabilities discussed in Section 2.2.4.

A numerical example clarifies the asymmetry. Let $P = (0.9, 0.1)$ and $Q = (0.5, 0.5)$ be distributions over two outcomes. The forward KL divergence is $D_{\mathrm{KL}}(P\|Q) = 0.9 \log_2(0.9/0.5) + 0.1 \log_2(0.1/0.5) = 0.9 \times 0.848 + 0.1 \times (-2.322) = 0.763 - 0.232 = 0.531$ bits. The reverse KL divergence is $D_{\mathrm{KL}}(Q\|P) = 0.5 \log_2(0.5/0.9) + 0.5 \log_2(0.5/0.1) = 0.5 \times (-0.848) + 0.5 \times 2.322 = -0.424 + 1.161 = 0.737$ bits. The two values differ — 0.531 versus 0.737 — confirming that KL divergence is not symmetric. The

reverse direction is larger in this case because the uniform model Q places substantial mass (0.5) on the outcome where P has only 0.1, and the reverse KL penalizes this mismatch more heavily. Understanding which direction we minimize, and why, is essential for understanding the behavior of trained language models and the advanced training techniques such as reinforcement learning from human feedback (RLHF) discussed in Chapter 12.

2.3.4 Perplexity

If cross-entropy measures the cost in bits, can we express that cost as something more intuitive? Perplexity is the standard evaluation metric for language models, and it transforms cross-entropy from an abstract bit count into a concrete, interpretable number. The perplexity of a model Q on a test sequence $w_1, w_2, \ldots, w_N$ is:

$$\mathrm{PP}(Q) = 2^{H(P,Q)} = 2^{-\frac{1}{N}\sum_{i=1}^{N}\log_2 Q(w_i|w_{<i})} \tag{2.6}$$

The perplexity can be interpreted as the effective vocabulary size: 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. A perplexity of 20 means the model has narrowed its predictions down to 20 effective choices. A perplexity of 1 would mean the model predicts every word with certainty — a perfect language model. A perplexity equal to $|V|$ (the vocabulary size) corresponds to a uniform model that gains no information from context. Lower perplexity is better, because it means the model is more certain about its predictions and assigns higher probability to the words that actually appear.

It is tempting to think that perplexity directly measures the quality of generated text or the usefulness of a model for downstream applications. It does not. Perplexity measures how well the model predicts held-out text — it is an intrinsic metric that evaluates the model's probabilistic calibration. A model with perplexity 20 on Wikipedia text may have perplexity 200 on legal contracts, because the word distribution differs between domains. Furthermore, a model with lower perplexity does not necessarily produce better translations, summaries, or question answers. These limitations, which we discuss in Section 2.4.3, are important to keep in mind even as we rely on perplexity as our primary evaluation metric throughout this book.

To compute perplexity on our running example, suppose our unigram model assigns the probabilities $Q(\text{the}) = 2/6$, $Q(\text{cat}) = Q(\text{sat}) = Q(\text{on}) = Q(\text{mat}) = 1/6$ to the held-out test sentence "the cat sat on the mat" (six words). The per-word log-probabilities are $\log_2 Q(\text{the}) = \log_2(2/6) = -1.585$, and $\log_2 Q(\text{cat}) = \log_2(1/6) = -2.585$. The average negative log-likelihood (the cross-entropy) is $H = -(1/6)[(-1.585)+(-2.585)+(-2.585)+(-2.585)+(-1.585)+(-2.585)] = (1/6)(13.510) = 2.252$ bits per word. The perplexity is $\mathrm{PP} = 2^{2.252} = 4.76$. Since our vocabulary has $|V| = 5$ words and a uniform model would give $\mathrm{PP} = 5.0$, our unigram model achieves a perplexity only slightly below the uniform baseline — which makes sense, because the unigram distribution over our tiny corpus is nearly uniform.

```python
import numpy as np

# Unigram model from corpus "the cat sat on the mat"
vocab = ['the', 'cat', 'sat', 'on', 'mat']
train_counts = np.array([2, 1, 1, 1, 1])  # raw counts
V = len(vocab)
```

```python
# Add-1 (Laplace) smoothing
smoothed = (train_counts + 1) / (train_counts.sum() + V)

# Held-out test sentence
test_sentence = ['the', 'cat', 'sat', 'on', 'the', 'mat']
log_probs = []
for word in test_sentence:
    idx = vocab.index(word)
    lp = np.log2(smoothed[idx])
    log_probs.append(lp)
    print(f"  P({word}) = {smoothed[idx]:.4f}, log2 = {lp:.3f}")

cross_entropy = -np.mean(log_probs)
perplexity = 2 ** cross_entropy

print(f"\nCross-entropy: {cross_entropy:.3f} bits/word")
print(f"Perplexity:    {perplexity:.2f}")
print(f"Vocab size:    {V} (uniform baseline PP = {V})")
```

2.3.5 Bits-per-Character and Bits-per-Token

Perplexity and cross-entropy can be measured at different levels of granularity, and the choice of granularity affects comparability across models. Bits-per-character (BPC) measures the cross-entropy at the character level: the average number of bits needed to predict each character. Bits-per-token (BPT) measures the cross-entropy at the token level, where a token might be a word, a subword unit, or a character depending on the tokenizer. The relationship between them depends on the average number of characters per token: if a tokenizer produces tokens averaging c characters each, then BPT $\approx c \times$ BPC. Shannon's estimate of 1.0 to 1.5 bits per character is expressed in BPC units, making it directly comparable across models regardless of tokenization.

Students compare perplexity values across models with different tokenizers, and this comparison is almost always invalid. A model operating at the character level (vocabulary size roughly 100) will report perplexity in the range of 2 to 5 for good English modeling. A model operating at the subword level (vocabulary size 30,000 to 50,000) will report perplexity in the range of 15 to 100 for comparable modeling quality. These numbers look dramatically different, but they reflect the same underlying predictive capability expressed at different granularities. Comparing them is like comparing fuel efficiency measured in miles per gallon versus miles per tank — the latter depends on tank size (vocabulary size) in the same way that token-level perplexity depends on the tokenizer. BPC provides a tokenizer-independent metric that allows fair comparison: if two models achieve the same BPC on the same test text, they are equally good at predicting that text, regardless of how they tokenize it. The standard practice for cross-model comparisons on character-level tasks is to report BPC rather than token-level perplexity, and we follow this convention in Chapter 4 when comparing word-level and subword-level models. For most practical purposes within a single model family using a consistent tokenizer, token-level perplexity remains the standard metric, and the perplexity values we report throughout Chapters 3 through 8 are all token-level unless explicitly noted otherwise.

2.4 Evaluation Metrics

2.4.1 Held-Out Perplexity

Perplexity must be computed on held-out data to be meaningful. The standard practice is to divide a corpus into three non-overlapping sets: a training set used to estimate model parameters, a validation (or development) set used to tune hyperparameters such as the n-gram order or the smoothing method, and a test set used for final evaluation. The splits are typically made at the document level to avoid data leakage: all sentences from a given document go into the same split, preventing the model from seeing paraphrases or related sentences across splits. A common split ratio is 80/10/10 (training/validation/test), though the exact proportions vary by dataset. Perplexity computed on the training set reflects how well the model has memorized the training data, not how well it generalizes to new text. Students report training perplexity as a measure of model quality, and this practice is misleading. An overfit model can achieve very low training perplexity — even approaching 1.0 — by memorizing the training data verbatim, while performing terribly on new text. Only held-out perplexity, computed on data the model has never seen during training, measures the generalization that we actually care about.

The analogy of academic assessment captures this distinction precisely. A student who memorizes the answer key for a specific exam (the training set) will achieve a perfect score on that exam (low training perplexity). But this performance reveals nothing about the student's understanding. The real test of knowledge is performance on a new exam with unseen questions (the test set). Held-out perplexity is the unseen exam. An important practical detail is that we tune hyperparameters on the validation set and report final results on the test set, never touching the test set during model development. This discipline prevents the test set from leaking into the model selection process and ensures that reported perplexity values are honest estimates of generalization performance. In Chapter 3, we will use held-out perplexity as the primary metric for comparing n-gram models of different orders and with different smoothing techniques, and in Chapters 5 through 8, we will track how held-out perplexity improves as we move from recurrent networks to attention mechanisms to transformer architectures.

2.4.2 Interpreting Perplexity Values

What does a perplexity of 50 actually mean in practice?

Raw perplexity numbers are meaningless without context. We have found that the perplexity number line is the single most useful reference tool students take away from this chapter. A perplexity of 50,000 on a 50,000-word vocabulary means the model is no better than random guessing — every word is equally likely regardless of context. This is the worst possible perplexity for a model that assigns non-zero probability to every word. At the other extreme, a perplexity of 1 means the model predicts every word with certainty — it is a perfect oracle that knows exactly what comes next. All real language models fall between these extremes, and the typical range depends on the model family, the corpus domain, and the vocabulary size. Unigram models, which ignore context entirely, typically achieve perplexities between 500 and 1,000 on standard English corpora. Bigram models reduce this to roughly 200 to 400. Trigram models with good smoothing achieve approximately 70 to 150. Neural language models based on LSTMs brought perplexity down to the 50 to 80 range on benchmark corpora like Penn Treebank. Transformer-based models have pushed further, achieving perplexities below 20 on the same benchmarks, with the largest models reaching single-digit perplexities on some evaluation sets.

These numbers have concrete meaning. A perplexity of 50 means the model is, on average, as uncertain as if choosing uniformly among 50 words at each position — a dramatic reduction from a vocabulary of 50,000. A perplexity of 20 means the model has narrowed each prediction to 20 effective choices. The progression from 1,000 to 50 to 20 represents genuinely different levels of predictive power: the trigram model eliminates 98 percent of the vocabulary at each step, while the transformer eliminates 99.96 percent. But perplexity comparisons are only valid under strict conditions: the test set, the vocabulary, and the tokenization must all be identical. Comparing the perplexity of a character-level model (vocabulary approximately 100) with a word-level model (vocabulary approximately 50,000) is not meaningful, as we discussed in Section 2.3.5. Similarly, perplexity on Wikipedia text is not comparable to perplexity on medical records, because the word distributions differ substantially. When we report perplexity improvements in later chapters, we always specify the benchmark corpus and vocabulary to ensure fair comparison.

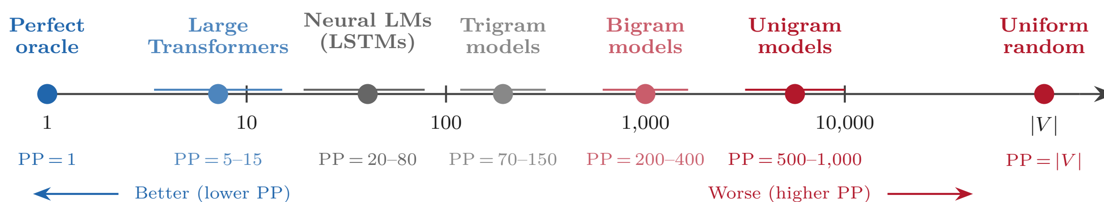

Figure 4: Figure 2.4 – Perplexity interpretation number line

2.4.3 When Perplexity Fails

Perplexity is our best intrinsic metric for language models, but it does not always predict success on downstream tasks. The disconnect between perplexity and task performance is subtle, and understanding it requires distinguishing between what perplexity measures and what we ultimately want. Perplexity measures how well a model predicts held-out text — it evaluates the model's estimate of the probability distribution over next words. A good language model, by this criterion, is one that assigns high probability to the words that humans actually write. But many applications of language models — machine translation, text summarization, question answering, dialogue generation — have their own success criteria that do not reduce to next-word prediction accuracy. A translation system is judged by BLEU score or human evaluation, not by the perplexity of its language model component. A summarization system is judged by ROUGE score or factual accuracy, not by how well it predicts the next word. Perplexity may correlate with these downstream metrics, but the correlation is imperfect and sometimes misleading.

No single number tells the whole story.

The analogy of GPA versus job performance captures the relationship. A student's GPA (the intrinsic metric) correlates with job performance (the downstream metric), but the correlation is far from perfect. A student with a 4.0 GPA may lack practical skills; a student with a 3.0 GPA may excel in the workplace. Similarly, a model with lower perplexity generally performs better on downstream tasks, but exceptions abound. Research has documented cases where improvements in perplexity did not translate to improvements in translation quality, and cases where models with higher perplexity produced better summaries because they better captured document-level coherence rather than word-level prediction. This tension between intrinsic and extrinsic evaluation is a recurring theme in

NLP. Chapters 7 and 10 will introduce the downstream evaluation benchmarks — BLEU, ROUGE, SuperGLUE, and human evaluation protocols — that complement perplexity as measures of model quality. For the remainder of this book, we use perplexity as our primary metric for comparing language models, while keeping in mind that it is a necessary but not sufficient condition for good performance on the tasks that ultimately matter.

2.5 Optimization Basics

2.5.1 Gradient Descent and SGD

The mathematical framework we have developed so far — the chain rule for decomposing sentence probabilities, MLE for learning parameters, and cross-entropy for measuring performance — is complete for n-gram models, where the MLE has a closed-form solution (count and divide). But for neural language models, which have millions or billions of parameters connected through nonlinear transformations, there is no closed-form solution. We must find the parameters that minimize the cross-entropy loss through iterative optimization, and the workhorse algorithm is gradient descent. The basic idea is simple: compute the gradient of the loss function $\mathcal{L}(\theta)$ with respect to the parameters θ, and update the parameters in the direction that reduces the loss: $\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L}(\theta)$, where η is the learning rate, a positive scalar that controls the step size. The gradient $\nabla_\theta \mathcal{L}$ points in the direction of steepest increase of the loss; by moving in the opposite direction, we decrease the loss. Repeating this update many times drives the parameters toward a (local) minimum of the loss function. Think about tuning an old analog radio — the kind with a physical dial. You hear static, so you nudge the dial slightly and listen: did the signal get clearer or worse? You keep nudging in whichever direction reduces the static. You never see the full spectrum of frequencies; you only hear the immediate result of each small adjustment. Gradient descent works the same way, except the "dial" has millions of knobs and the "signal" is a loss function. The procedure will eventually find a setting that sounds clear, though not necessarily the clearest setting on the entire spectrum.

In practice, computing the gradient over the entire training corpus (full-batch gradient descent) is computationally prohibitive for the corpora used in language modeling, which contain millions or billions of tokens. Stochastic gradient descent (SGD) approximates the full gradient by computing it on a small random subset of the data — a mini-batch — at each update step. A mini-batch might contain 32, 64, or 256 sentences. The gradient computed on a mini-batch is a noisy estimate of the true gradient, but it is an unbiased estimate in expectation: the expected value of the mini-batch gradient equals the true gradient. This noise is not purely a nuisance. Empirical evidence suggests that the noise in SGD provides a regularization effect, helping the optimizer escape sharp local minima that generalize poorly and settle into flatter minima that generalize better [Goodfellow et al., 2016]. The tradeoff between batch size and gradient noise is a practical consideration: larger batches produce more accurate gradient estimates but require more computation per update, while smaller batches are noisier but allow more frequent parameter updates. The optimal batch size depends on the model, the dataset, and the available hardware, and is typically determined through experimentation.

2.5.2 Adam and Adaptive Methods

In January 2015, Diederik Kingma and Jimmy Ba published a short paper at ICLR titled "Adam: A Method for Stochastic Optimization" that has since accumulated over 200,000 citations, making it one of the most cited papers in all of machine learning [Kingma and Ba, 2015]. Adam combines

two ideas — momentum and adaptive learning rates — into a single optimizer that has become the default choice for training neural language models. To understand Adam, we first need to understand its components. Momentum maintains a running average of past gradients, which smooths out oscillations and accelerates convergence along consistent gradient directions. Formally, the first-moment estimate is $m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$, where $g_t = \nabla_\theta \mathcal{L}$ is the current gradient and β_1 (typically 0.9) controls the decay rate. RMSProp, introduced by Geoffrey Hinton in an unpublished lecture note, maintains a running average of squared gradients: $v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$, where β_2 (typically 0.999) controls the second-moment decay rate. The division by $\sqrt{v_t}$ gives each parameter its own effective learning rate, scaling down updates for parameters with large gradients and scaling up updates for parameters with small gradients.

Adam combines both: the update rule is $\theta \leftarrow \theta - \eta \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$, where $\hat{m}_t = m_t / (1 - \beta_1^t)$ and $\hat{v}_t = v_t / (1 - \beta_2^t)$ are bias-corrected estimates that account for the initialization of $m_0 = v_0 = 0$, and $\epsilon = 10^{-8}$ prevents division by zero. The bias correction is important during the initial training steps: without it, the estimates m_t and v_t are biased toward zero because they are initialized at zero, which would cause the optimizer to take inappropriately large or small steps. Adam's appeal is practical — it converges faster than plain SGD for most neural network architectures, it is relatively insensitive to the initial learning rate (a well-chosen default of $\eta = 10^{-3}$ or 3×10^{-4} works for most problems), and it requires minimal hyperparameter tuning beyond the learning rate itself. The AdamW variant [Loshchilov and Hutter, 2019], which decouples weight decay from the gradient update, has become the standard for training transformer-based language models and is the optimizer used in essentially all large language model training runs as of 2024. One important caveat: Adam does not always outperform SGD with momentum. For some fine-tuning tasks and some model architectures, SGD generalizes better despite converging more slowly. The choice of optimizer remains an empirical decision, though Adam is the safe default for training language models from scratch.

2.5.3 Learning Rate Schedules

The learning rate is arguably the single most important hyperparameter in neural language model training. Whether cosine decay is truly optimal or merely the first schedule that worked well enough to become convention is a question the field has not conclusively settled. A constant learning rate throughout training is rarely optimal. If the learning rate is too high, the optimizer oscillates wildly and may diverge — the loss increases instead of decreasing. If the learning rate is too low, the optimizer converges painfully slowly and may get trapped in a poor local minimum. The solution is a learning rate schedule that varies η over the course of training, typically starting small, increasing to a peak, and then decaying. The most common schedule for transformer language models consists of two phases: a warmup phase, where the learning rate increases linearly from zero (or near zero) to a peak value over the first several thousand training steps, followed by a decay phase, where the learning rate decreases gradually according to a cosine or linear schedule.

Warmup serves a specific purpose: at the beginning of training, the parameters are randomly initialized and the gradients are unreliable. Large learning rates at this stage cause the optimizer to take large steps based on noisy, uninformative gradients, which can destabilize training irreversibly. By starting with a small learning rate and increasing it gradually, we allow the model to develop meaningful gradient signals before the optimizer begins taking large steps. The cosine decay schedule, which decreases the learning rate according to $\eta_t = \eta_{\min} + \frac{1}{2}(\eta_{\max} - \eta_{\min})(1 + \cos(\pi t / T))$, where T is the total number of training steps, has become the standard in transformer training. It provides a smooth transition from the peak learning rate to a near-zero rate at the end of training, allowing the optimizer to make fine-grained adjustments in the final stages. The analogy of driving a car

captures the logic: you accelerate gently from a stop (warmup), cruise at highway speed (peak rate), and decelerate smoothly as you approach your destination (cosine decay). Trying to drive at highway speed from the first moment would cause a crash (training divergence). The learning rate schedule is not a minor detail — it is one of the most important hyperparameters in transformer training, and getting it wrong can mean the difference between a well-trained model and a complete training failure. Chapter 8 will return to this topic when we discuss the specific schedules used in training the original transformer and its successors.

We now have the complete mathematical toolkit. The chain rule (Equation 2.1) tells us how to decompose sentence probabilities into next-word predictions. Maximum likelihood estimation (Equation 2.2) tells us how to learn those predictions from data by counting. Entropy (Equation 2.3) measures the inherent uncertainty in language. Cross-entropy (Equation 2.4) measures the cost of imperfect prediction. KL divergence (Equation 2.5) measures the gap between our model and the truth. Perplexity (Equation 2.6) converts that gap into an interpretable number. And gradient-based optimization gives us the machinery to improve our predictions iteratively when closed-form solutions do not exist. But we have not yet built an actual language model. In Chapter 3, we apply these tools to construct the simplest and historically most important language model: the n-gram model. We will see that count-and-divide — the MLE formula we just derived — is both the foundation and the fundamental limitation of classical language modeling.

With these mathematical tools in hand, we make a natural transition to Chapter 3, where we apply probability theory and information-theoretic measures to build classical n-gram language models.

Exercises

Exercise 1 (Theory, Basic). Apply the chain rule of probability to decompose $P(\text{"I"}, \text{"love"}, \text{"NLP"})$ into a product of three conditional probabilities. Write out each factor explicitly and explain what each factor represents in terms of the language modeling task.

Hint: $P(w_1, w_2, w_3) = P(w_1) \cdot P(w_2 \mid w_1) \cdot P(w_3 \mid w_1, w_2)$. The first factor $P(\text{"I"})$ is the probability of starting a sentence with the word "I."

Exercise 2 (Theory, Basic). Compute the entropy of a fair six-sided die (six equally likely outcomes) and of a loaded die where face 6 has probability 0.5 and the remaining five faces each have probability 0.1. Which die has higher entropy, and why?

Hint: $H = -\sum P(x) \log_2 P(x)$. The fair die has maximum entropy for six outcomes: $H_{\text{fair}} = \log_2 6 \approx 2.585$ bits.

Exercise 3 (Theory, Intermediate). Derive the perplexity of a language model that assigns a uniform distribution over a vocabulary of size $|V|$ at every position. Show that $\text{PP} = |V|$. Explain why this is the worst possible perplexity for a model that assigns non-zero probability to every word.

Hint: If $Q(w) = 1/|V|$ for all w, then $H(P, Q) = -\sum_x P(x) \log_2(1/|V|) = \log_2 |V|$, regardless of P.

Exercise 4 (Theory, Advanced). Prove that KL divergence is non-negative: $D_{\text{KL}}(P \| Q) \geq 0$ for all distributions P and Q, with equality if and only if $P = Q$. Use Jensen's inequality and the concavity of the logarithm.

Hint: Write $D_{\mathrm{KL}}(P\|Q) = -\sum_x P(x) \log \frac{Q(x)}{P(x)}$ and apply Jensen's inequality to the concave function log, noting that $\mathbb{E}_P\left[\log \frac{Q(X)}{P(X)}\right] \leq \log \mathbb{E}_P\left[\frac{Q(X)}{P(X)}\right] = \log 1 = 0$.

Exercise 5 (Programming, Basic). Write a Python function `entropy(probs)` that takes a probability distribution (as a NumPy array) and returns its Shannon entropy in bits. Test it on: (a) a uniform distribution over 10 outcomes, (b) a distribution where one outcome has probability 0.9 and the remaining nine outcomes share 0.1 equally.

Hint: Use `np.log2` and handle zeros with `np.where(probs > 0, probs * np.log2(probs), 0)`. The uniform case should give $H = \log_2 10 \approx 3.322$ bits.

Exercise 6 (Programming, Basic). Compute the cross-entropy between the true English letter frequency distribution (from published tables, as used in Code Example 1) and three models: (a) a uniform distribution over 27 symbols, (b) a distribution based on Scrabble tile frequencies, (c) a distribution estimated from a paragraph of your own writing. Report which model achieves the lowest cross-entropy.

Hint: $H(P, Q) = -\sum_x P(x) \log_2 Q(x)$. Ensure that Q assigns non-zero probability to every letter by using add-ϵ smoothing if necessary.

Exercise 7 (Programming, Intermediate). Build a unigram language model from a text file (for example, a chapter of a public-domain novel from Project Gutenberg). Compute its perplexity on a held-out chapter from the same book. Then compute perplexity on a chapter from a different book in a different domain. Report both perplexity values and explain the difference.

Hint: Use add-1 smoothing to avoid zero probabilities. Domain mismatch will increase perplexity because the word distribution shifts — a model trained on Dickens will be surprised by the vocabulary of a medical textbook.

Exercise 8 (Programming, Intermediate). Plot the entropy of a categorical distribution over 5 outcomes as a function of a temperature parameter T applied to logits $[3.0, 2.0, 1.0, 0.5, 0.1]$. Vary T from 0.1 to 10.0 and identify the temperature at which the entropy reaches 90 percent of its maximum value ($H_{\max} = \log_2 5 \approx 2.322$ bits).

Hint: Apply softmax(logits/T) and compute entropy for each T. The entropy should increase monotonically with temperature, approaching $H_{\max}$ as $T \to \infty$.

Exercise 9 (Programming, Intermediate). Implement SGD and Adam from scratch (without using any external optimizer library) to minimize the simple quadratic loss function $\mathcal{L}(\theta) = (\theta - 3)^2$. Plot the convergence curves (θ versus iteration number) for both optimizers starting from $\theta_0 = 0$ with learning rate $\eta = 0.1$. Which optimizer converges faster?

Hint: For Adam, implement the first-moment estimate m, second-moment estimate v, and bias correction terms $\hat{m} = m/(1 - \beta_1^t)$ and $\hat{v} = v/(1 - \beta_2^t)$ exactly as described in Kingma and Ba [2015].

Exercise 10 (Programming, Advanced). Download the first 1,000 sentences of WikiText-2. Build a bigram model with add-1 smoothing. Compute the perplexity on a held-out set of 100 sentences. Then compute the KL divergence between the true bigram distribution (from raw counts) and the smoothed bigram distribution for a specific high-frequency context word such as "the." Discuss how smoothing affects the KL divergence.

Hint: The KL divergence is from the MLE distribution to the smoothed distribution. Smoothing redistributes probability mass from frequent bigrams to rare ones, so $D_{\mathrm{KL}}(\hat{P}_{\mathrm{MLE}} \| \hat{P}_{\mathrm{smooth}})$ will be

non-zero. The magnitude depends on how much mass is redistributed.