LLMs Derived

A Mathematical and Engineering Guide to Large Language Models



Choral Software

LLMs Derived: A Mathematical and Engineering Guide to Large Language Models

Copyright © 2026 Choral Software

All rights reserved. No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the copyright holder, except in the case of brief quotations embodied in critical reviews and certain other noncommercial uses permitted by copyright law. For permission requests, contact the publisher.

First Edition, 2026

Published independently through Amazon Kindle Direct Publishing.

The Python code examples in this book are provided for educational purposes. While every effort has been made to ensure the accuracy of the mathematics, code, and technical claims in this book, the publisher makes no warranties, express or implied, regarding the completeness or accuracy of its contents, and assumes no liability for damages resulting from the use of the information or code contained herein. Library names, product names, and trademarks mentioned in this book (including PyTorch, NumPy, and others) are the property of their respective owners; their use here is for identification and educational purposes only and does not imply endorsement.

Cover design by Choral Software.

Preface

This is a book about what a large language model computes, not what it might one day be able to do. Every claim in it is either a derivation you can check with a pencil, a piece of code you can run, or a citation to a published result — nothing here rests on intuition alone, and nothing is included because it sounds impressive. If a topic is genuinely unsettled (why certain abilities appear to emerge with scale, what sparse autoencoders do and don't reveal about a model's internals, how well RLHF actually generalizes), the text says so explicitly and cites the debate rather than picking a side for you.

Who this book is for. You should be comfortable with linear algebra (matrix products, eigenvalues, norms), multivariable calculus (the chain rule, gradients, Jacobians), and probability (expectations, conditional distributions, maximum likelihood). You should be able to read Python fluently and have at least seen PyTorch before, though Chapter 3 gives a compressed refresher of the specific autodiff and optimization machinery the rest of the book leans on. This is not an introduction to machine learning — it assumes you already know what a neural network and a loss function are, and instead spends its pages on what is specific to language models: attention, tokenization, the transformer block, pretraining at scale, alignment, and inference. If you are a graduate student in ML, a software engineer moving into an ML role, or a researcher in an adjacent field who wants the actual mathematics rather than a product-launch-level summary, this book was written for you.

How to read it. The eighteen chapters are organized into five parts that build on each other in order: Foundations (Chapters 1–4) establish notation and the probability, optimization, and neural-network machinery every later chapter uses; Representing Text (Chapters 5–6) covers tokenization and embeddings; The Transformer Architecture (Chapters 7–10) derives attention and assembles the full modern decoder-only model, including the architectural variants (grouped-query attention, mixture-of-experts) that production systems actually use; Training at Scale (Chapters 11–14) covers pretraining, optimization, scaling laws, and distributed systems; and Adaptation, Alignment, and Deployment (Chapters 15–18) covers fine-tuning, RLHF and DPO, inference-time decoding, and evaluation. Read the parts in order the first time through — later chapters assume earlier derivations rather than repeating them, and cross-references point back to the specific chapter and result being reused rather than re-deriving it. After a first pass, each chapter is self-contained enough to use as a reference: if you already know how attention works and only want the derivation of the DPO loss, Chapter 16 does not require you to reread Chapter 7 first, only to accept its results.

Every chapter follows the same shape: a short abstract stating what you'll be able to do afterward, several sections of derivation and explanation, at least one runnable Python listing per chapter, a Summary recapping the key results, and Exercises mixing hand derivations with coding tasks. The exercises don't come with solutions — working through a derivation yourself is where the understanding actually happens, and the book gives you everything needed to check your own answer (the same equations, in the same notation, are right there in the chapter).

On the code. Every code listing in this book is real, complete Python, not pseudocode — it uses PyTorch as the primary framework, with NumPy for a handful of from-scratch pedagogical listings (a tiny autodiff engine in Chapter 3, byte-pair encoding in Chapter 5) where the point is transparency rather than production performance. Listings assume batch-first tensors, `(batch, seq, d_model)`, unless a chapter explicitly discusses a layout that requires otherwise, such as a KV cache. Appendix B gives the package versions and setup used to write and check this book's code, along with notes

on what to do if a listing behaves differently under a newer library version. None of the code here is optimized for production deployment; it is optimized for you being able to read every line and know exactly what it does and why.

A note on scope. This book is about how language models work — the mathematics and engineering underneath them — not about prompting them, not about product applications built on top of them, and not about the broader societal or policy questions they raise, all of which are real and important but are different books. Where the text touches on limitations, safety-relevant behavior, or open research questions (hallucination, interpretability, evaluation contamination), it does so with the same standard as everywhere else: grounded in published, citable work, stated with appropriate uncertainty, and never overclaimed.

Chapter 1 starts, deliberately, with the simplest possible question: what does a language model actually compute? Everything else in the book is an answer to some part of that question, at increasing levels of depth.

Contents

Preface i

I Foundations 1

Introduction: What a Language Model Actually Computes 3
- The autoregressive factorization of language . . . 3
- From N-Grams to GPT-Scale Language Models: A Brief History . . . 5
- What "Large" Means: Parameters, Tokens, and Compute . . . 6
- The Three Phases: Pretraining, Adaptation, and Inference . . . 9
- Notation and Conventions Used Throughout This Book . . . 11
- How This Book Is Organized . . . 13
- Summary . . . 14
- Exercises . . . 15

Probability and Information Theory for Language Modeling 17
- Probability chain rule and autoregressive modeling . . . 17
- Entropy, cross-entropy, KL divergence . . . 19
- Perplexity as an evaluation metric . . . 21
- Maximum likelihood estimation as cross-entropy minimization . . . 22
- N-gram models and their limits . . . 24
- Python: entropy, cross-entropy, perplexity, and a bigram model from scratch . . . 26
- Summary . . . 30
- Exercises . . . 30

Linear Algebra, Calculus, and Optimization Refresher (LLM-relevant slice) 33
- Matrix and Tensor Operations in Deep Networks . . . 33
- Gradients, Jacobians, and the Chain Rule . . . 36
- Backpropagation as Reverse-Mode Automatic Differentiation . . . 38
- Gradient Descent, Stochastic Gradient Descent, and Momentum . . . 39
- Second-Order Methods and Why They Don't Scale . . . 41
- Python: A Minimal Reverse-Mode Autodiff Engine . . . 43
- Summary . . . 46
- Exercises . . . 47

Neural Network Building Blocks 49
- The multilayer perceptron and universal approximation . . . 49
- Activation functions . . . 51
- Normalization: BatchNorm vs. LayerNorm vs. RMSNorm . . . 53
- Residual connections and gradient flow . . . 56
- Weight initialization . . . 58

Python: LayerNorm and RMSNorm from scratch, verified against PyTorch 59
Summary 65
Exercises 65

II Representing Text 67

Tokenization: From Text to Tensors 69
Why not just use characters or words 69
Byte-Pair Encoding 71
WordPiece and Unigram language model tokenization 73
Byte-level BPE 75
Vocabulary size tradeoffs 76
Python: byte-pair encoding training and encoding from scratch 78
Summary 83
Exercises 84

Embeddings and Positional Encoding 85
Token Embeddings as a Learned Lookup Table 85
Why Order Must Be Injected Explicitly 86
Sinusoidal Absolute Positional Encoding 88
Learned Absolute Positions 89
Relative Position and Rotary Position Embeddings (RoPE) 90
ALiBi as an Alternative 92
Python: Sinusoidal PE and RoPE from Scratch 94
Summary 96
Exercises 97

III The Transformer Architecture 99

Attention: Mathematical Foundations 101
The Information Bottleneck of RNN Encoders 101
The Query, Key, and Value Formalism 102
Scaled Dot-Product Attention 104
Computational Complexity 107
Causal Masking for Autoregressive Models 108
Python: Scaled Dot-Product Attention From Scratch 110
Summary 113
Exercises 113

Multi-Head Self-Attention, Implemented 115
Why multiple heads: the subspace argument 115
Full multi-head formalism and parameter count 117
Self-attention vs. cross-attention 118
Implementing a complete `MultiHeadSelfAttention` module in PyTorch 120
Attention pattern visualization 123

FlashAttention . . . 125
Summary . . . 128
Exercises . . . 128

The Transformer Block and Full Architecture **131**
Position-wise feedforward network . . . 131
Pre-norm vs. post-norm . . . 133
The residual stream view . . . 135
Assembling encoder, decoder, and decoder-only blocks . . . 137
Parameter counting formula . . . 139
Python: a complete GPT-style decoder-only Transformer . . . 141
Summary . . . 147
Exercises . . . 148

Modern Architectural Variants **149**
KV caching for autoregressive inference . . . 149
Multi-query and grouped-query attention . . . 151
Sliding-window and sparse attention patterns . . . 153
Mixture-of-experts . . . 155
Long-context tricks: RoPE scaling and YaRN . . . 158
Python: grouped-query attention and a top-1 Mixture-of-Experts router . . . 159
Summary . . . 165
Exercises . . . 165

IV Training at Scale 167

Pretraining: Objectives, Data, and Practice **169**
The causal language modeling objective . . . 169
Data pipeline: crawling, filtering, and deduplication . . . 171
Data mixing and curriculum . . . 175
Packing sequences and attention masking for packed sequences . . . 176
Tokenizer training at corpus scale . . . 178
Python: a complete pretraining loop for a toy GPT . . . 179
Summary . . . 186
Exercises . . . 187

Optimization at Scale **189**
SGD with Momentum Recap . . . 189
Adam and AdamW Derived . . . 189
Learning-Rate Schedules: Warmup and Cosine Decay . . . 193
Gradient Clipping . . . 196
Mixed-Precision Training . . . 197
Python: AdamW Implemented from Scratch . . . 199
Summary . . . 203
Exercises . . . 204

Scaling Laws and Compute-Optimal Training **205**
Empirical scaling laws . . . 205
The Chinchilla compute-optimal analysis . . . 207
Fitting a scaling law to synthetic data in Python . . . 211
Emergent abilities . . . 215
Using scaling laws for training-run planning . . . 216
Summary . . . 218
Exercises . . . 219

Distributed Training Systems **221**
Memory accounting for training . . . 221
Data parallelism . . . 225
Tensor (model) parallelism . . . 227
Pipeline parallelism . . . 228
ZeRO stages 1–3 . . . 230
Putting it together: 3D parallelism for a frontier-scale run . . . 232
Summary . . . 233
Exercises . . . 234

V Adaptation, Alignment, and Deployment **237**

Fine-Tuning and Parameter-Efficient Adaptation **239**
Supervised Fine-Tuning: Objective and Data Format . . . 239
The Cost of Full Fine-Tuning and the Case for Parameter-Efficient Methods . . . 242
LoRA: Low-Rank Adaptation . . . 244
QLoRA and the Arithmetic of Quantization . . . 246
Adapters, Prefix Tuning, and Prompt Tuning . . . 248
Python: A LoRA Linear Layer from Scratch . . . 249
Summary . . . 252
Exercises . . . 253

Alignment: RLHF, Reward Models, and DPO **255**
Why Pretraining and SFT Alone Under-Specify Desired Behavior . . . 255
The Bradley-Terry Preference Model and Reward Model Training . . . 256
The RLHF Objective and PPO . . . 258
Direct Preference Optimization . . . 260
Constitutional AI and RLAIF . . . 264
Python: The DPO Loss and a Toy Training Step . . . 264
Summary . . . 267
Exercises . . . 268

Inference and Decoding **269**
Autoregressive decoding and the KV cache, revisited quantitatively . . . 269
Greedy decoding and beam search . . . 272
Sampling: temperature, top-k, top-p, and typical sampling . . . 274
Speculative decoding . . . 276

Quantization for inference . . . 278
Python: temperature, top-k, and top-p sampling from scratch . . . 279
Summary . . . 283
Exercises . . . 283

Evaluation, Interpretability, and Limitations **285**
Intrinsic vs. extrinsic evaluation . . . 285
Benchmark suites and contamination . . . 288
Hallucination: a probabilistic framing . . . 290
Mechanistic interpretability primer . . . 292
Known limitations . . . 295
Where the field is heading . . . 296
Summary . . . 297
Exercises . . . 298

Notation Reference **301**

Python Environment and Reproducing the Code **305**

Glossary **307**

References **311**

About the Publisher **317**

Part I

Foundations

Introduction: What a Language Model Actually Computes

A language model is, at its mathematical core, a single object: a conditional probability distribution over the next token given everything that came before it. This chapter states that object precisely, traces the four architectural eras that got us from counting word frequencies to training networks with hundreds of billions of parameters, gives you real numbers for what "large" means today, lays out the three-phase lifecycle (pretraining, adaptation, inference) that organizes the rest of the book, and fixes the notation you will see on every subsequent page.

The autoregressive factorization of language

A sequence of text is a sequence of discrete tokens $x_1, x_2, \ldots, x_n$, each drawn from a finite vocabulary $\mathcal{V}$. A language model assigns a probability $p(x_1, \ldots, x_n)$ to this sequence. There is nothing specific to language in that statement — it is just a joint distribution over n discrete random variables, and in principle a table of $|\mathcal{V}|^n$ numbers would specify it completely. For any vocabulary and sequence length used in practice, that table is astronomically larger than the number of atoms in the observable universe, so it is never constructed. Everything interesting about language modeling is about how to represent and estimate this joint distribution *implicitly* and *tractably*.

The chain rule of probability provides the factorization every modern LLM is built on. For any joint distribution over n random variables, and for any fixed ordering of them,

$$p(x_1, x_2, \ldots, x_n) = \prod_{t=1}^{n} p(x_t \mid x_1, \ldots, x_{t-1}) = \prod_{t=1}^{n} p(x_t \mid x_{<t}), \tag{1.1}$$

where $x_{<t}$ denotes the prefix $(x_1, \ldots, x_{t-1})$, and $x_{<1}$ is the empty sequence. This identity is exact — it follows directly from the definition of conditional probability applied repeatedly ($p(a, b) = p(a)p(b \mid a)$, then treat b as itself a joint variable and recurse), and it makes no assumption whatsoever about the structure of language. It holds for any joint distribution over any ordered collection of discrete variables. What makes it useful for language is a modeling choice, not a mathematical one: we fix the ordering to be the natural left-to-right reading order of text, and we replace the intractable true conditionals $p(x_t \mid x_{<t})$ with conditionals produced by a parametric function $p_\theta(x_t \mid x_{<t})$, where θ denotes the collection of all trainable weights of a neural network.

This single substitution — true joint replaced by a product of learned one-step-ahead conditionals — is what "autoregressive language modeling" means, and it is the mathematical foundation shared by GPT, LLaMA, PaLM, and essentially every consumer-facing generative language model as of this writing. It has three properties that make it tractable where the full joint is not:

First, each factor $p_\theta(x_t \mid x_{<t})$ is a distribution over only $|\mathcal{V}|$ outcomes — a single softmax over the vocabulary — rather than over $|\mathcal{V}|^n$ joint outcomes. A neural network that consumes a variable-length prefix and outputs one categorical distribution over $\mathcal{V}$ is an entirely standard supervised-learning ob-

ject: it is a classifier, and classifiers with tens of thousands of output classes are routine. The autoregressive factorization turns an intractable density-estimation problem into n tractable classification problems that share parameters.

Second, training such a model requires no special generative-modeling machinery. Given a corpus of real sequences, maximizing $\prod_t p_\theta(x_t \mid x_{<t})$ is ordinary maximum-likelihood estimation, which reduces to minimizing cross-entropy loss token by token — the same loss used for any classification problem. Chapter 2 (Probability and Information Theory for Language Modeling) derives this equivalence between maximum likelihood and cross-entropy minimization in full and connects it to the perplexity metric used to report model quality.

Third, and most consequentially for how these models are used, the factorization is directly generative. If you have learned (an approximation to) each conditional $p_\theta(x_t \mid x_{<t})$, you can produce a sample from the full joint $p_\theta(x_1, \ldots, x_n)$ by sampling $x_1 \sim p_\theta(x_1)$, then $x_2 \sim p_\theta(x_2 \mid x_1)$, and so on — sampling one token at a time, each conditioned on everything sampled so far, until you choose to stop. This is exactly what happens when you interact with a chat model: every token you see was drawn from a categorical distribution conditioned on the entire conversation so far, including the tokens the model itself already emitted. Chapter 17 (Inference and Decoding) covers the mechanics of that sampling process — greedy decoding, beam search, temperature and nucleus sampling — in detail; for now, the point is that autoregressive factorization is simultaneously a training objective (cross-entropy on next-token prediction) and a generation procedure (iterative sampling), and those two uses of the same factorization are what make the architecture practical end to end.

It is worth being explicit about what the factorization does *not* require. It does not require the ordering to be left-to-right — the chain rule holds for any permutation of the variables. Left-to-right is chosen because it matches how humans produce and consume text, which makes training data (ordinary documents) directly usable without reordering, and because it yields a generation procedure that produces text incrementally, the way a person would type it. It also does not require that $p_\theta(x_t \mid x_{<t})$ actually be computed from *all* of $x_{<t}$ by every architecture that has been used for language modeling historically — n-gram models, discussed below, truncate the context to a fixed window. But every architecture discussed in this book after Chapter 6 computes $p_\theta(x_t \mid x_{<t})$ using the full prefix, which is exactly the capability that self-attention (Chapter 7) provides and that recurrent architectures approximate imperfectly, as the history below explains.

Non-autoregressive alternatives. The autoregressive factorization is not the only way to build a language model, and understanding the alternative clarifies what autoregression buys you. BERT (Devlin et al., 2019) trains an encoder that consumes an entire sequence at once, with roughly 15% of tokens replaced by a special mask token, and predicts the masked tokens from the *bidirectional* context — the model sees tokens both before and after each masked position. Formally, this objective targets $p(x_i \mid x_{\mathcal{M}\setminus i})$ for masked positions i in a masked set $\mathcal{M}$, conditioning on the unmasked remainder of the sequence in both directions. This is not a valid factorization of a single joint distribution the way the chain rule is — the set of conditionals a masked language model learns does not correspond to any single, consistent ordering, and in general no factorization of $p(x_1, \ldots, x_n)$ into these particular conditionals exists (this is why masked language models are usually described as learning a *pseudo-likelihood* objective rather than a proper generative model). What bidirectional masking buys is a representation-learning objective well suited to producing embeddings for classification, retrieval, and other discriminative downstream tasks, since every position gets to use context from both directions when predicting held-out tokens. What it does not directly support is autoregressive generation: there is no natural "next token given everything so far" sampling procedure for a masked language model, because the model was never trained to condition only on a causal prefix. This is precisely why

encoder-only masked models like BERT excel at classification and retrieval benchmarks but are not used as open-ended text generators, while decoder-only autoregressive models are the architecture of choice whenever the product is free-form generation — which is the case for essentially every system this book calls an "LLM." From here on, "language model" in this book means an autoregressive model in the sense of Equation 1.1 unless stated otherwise.

From N-Grams to GPT-Scale Language Models: A Brief History

The autoregressive factorization has been the mathematical target for decades; what has changed roughly every decade is *how* $p_\theta(x_t \mid x_{<t})$ is represented and estimated. Four eras summarize the progression.

N-gram models (1980s–2000s). The earliest tractable language models make a Markov assumption: instead of conditioning on the full prefix $x_{<t}$, they truncate to the previous $k-1$ tokens, $p(x_t \mid x_{<t}) \approx p(x_t \mid x_{t-k+1}, \ldots, x_{t-1})$, and estimate each such conditional directly from corpus counts — a k-gram model's parameters are, in the simplest case, just normalized co-occurrence counts. This makes estimation trivial (counting) but forces a severe tradeoff: larger k captures more context but the number of distinct k-grams grows exponentially with k, so almost every possible k-gram is never observed in any finite corpus, no matter how large. This is the curse of dimensionality applied directly to sequences, and it is why practical n-gram models rarely use k larger than 4 or 5 without aggressive smoothing techniques (Katz backoff, Kneser–Ney smoothing) to assign non-zero probability to unseen combinations. N-gram models cannot represent long-range dependencies by construction — a fact reflected directly in Equation 1.1, since they replace the true conditional on the full prefix with a conditional on a fixed-width window. Chapter 2 (Probability and Information Theory for Language Modeling) works through the data-sparsity argument and smoothing in more depth; the point to carry forward here is that every subsequent era in this history is, in one way or another, a better answer to the same question n-grams could not answer: how do you condition on a long, variable-length history without needing exponentially much data?

Recurrent networks: RNNs and LSTMs (1990s–2014). Recurrent neural networks replace the discrete counting table with a continuous hidden state $\mathbf{h}_t \in \mathbb{R}^d$ updated at every step by a shared, learned function, $\mathbf{h}_t = f_\theta(\mathbf{h}_{t-1}, \mathbf{x}_t)$, and predict $p_\theta(x_{t+1} \mid x_{\leq t}) = g_\theta(\mathbf{h}_t)$. Because $\mathbf{h}_t$ is in principle a function of the entire prefix $x_{\leq t}$ (it is computed recursively from $\mathbf{h}_{t-1}$, which itself depends on everything before it), RNNs sidestep the fixed-window limitation of n-grams: the conditioning context is, architecturally, unbounded. In practice, plain RNNs trained by backpropagation through time suffer from vanishing and exploding gradients over long sequences, which makes them unable to actually learn to exploit long-range dependencies even though nothing in the architecture rules it out. The Long Short-Term Memory (LSTM) architecture (Hochreiter and Schmidhuber, 1997) addresses this with gating mechanisms — learned input, forget, and output gates that control how information is written to and read from a memory cell — which substantially improves gradient flow over long sequences and made recurrent models the dominant architecture for sequence modeling, including early neural machine translation and language modeling systems, through the early-to-mid 2010s. A further refinement in this era, attention (Bahdanau et al., 2015), was introduced not to replace recurrence but to augment it: rather than forcing an encoder to compress an entire source sentence into one fixed-size vector for a decoder to consume, attention let the decoder look back at a learned weighted combination of *all* encoder hidden states at every output step. This idea — computing a distribution over positions and taking a weighted sum of their representations — is the direct conceptual ancestor of the self-attention

mechanism that defines the next era, and Chapter 7 (Attention: Mathematical Foundations) opens with exactly this motivation.

The Transformer (2017). Vaswani et al. (2017) made two changes that jointly define the modern era. First, they replaced recurrence entirely with self-attention: instead of processing tokens sequentially and passing information forward through a hidden state, every position attends directly to every other position in a single layer, computing $p_\theta(x_t \mid x_{<t})$'s representation from a weighted combination of *all* prefix positions' representations in one step, with the weights themselves learned and input-dependent. Second, because this removes the sequential dependency between time steps that recurrence imposes, all positions in a training sequence can be processed in parallel on a GPU, rather than one step at a time — a change with more practical consequence for what could subsequently be trained than any single accuracy improvement. Self-attention also has a computational cost that recurrence does not: computing all pairwise interactions among n positions costs $O(n^2 d)$, quadratic in sequence length, whereas an RNN's per-step cost is constant in n. That tradeoff — quadratic compute in exchange for full parallelism and unimpeded long-range information flow (no vanishing gradients through n sequential recurrent steps) — is one of the central engineering themes of this book and is analyzed formally in Chapter 7. The original Transformer was an encoder-decoder architecture built for machine translation; the decoder-only variant that discards the encoder and processes a single stream of tokens autoregressively — the architecture used by every model in the next era — is introduced in Chapter 9 (The Transformer Block and Full Architecture).

GPT-scale large language models (2018–present). Radford et al. (2018) demonstrated that a decoder-only Transformer, pretrained with the plain autoregressive objective of Equation 1.1 on a large unlabeled text corpus and then fine-tuned on a downstream task, outperformed task-specific architectures — establishing the pretrain-then-adapt paradigm around which this entire book, and this chapter's third section below, is organized. Radford et al. (2019) then showed that scaling the same recipe up (1.5 billion parameters, a larger and more diverse web-text corpus) produced a model capable of performing many tasks *without* any task-specific fine-tuning at all, given only a natural-language description of the task — an early demonstration of what would later be called zero-shot and few-shot capability. Brown et al. (2020) pushed the same architecture and objective to 175 billion parameters and showed that few-shot performance — providing a handful of input-output examples in the prompt itself, with no gradient updates — continued to improve smoothly with scale across dozens of tasks, a result that reframed scale itself (parameters, data, compute) as a primary lever for capability rather than a secondary engineering concern. Nothing about the mathematical objective changed across these three papers — each is, at its core, still Equation 1.1 optimized by cross-entropy loss on a decoder-only Transformer. What changed was scale, data curation, and the resulting behavior, which is precisely what the next section quantifies.

What "Large" Means: Parameters, Tokens, and Compute

"Large" in "large language model" is a moving target — BERT-large's 340 million parameters (Devlin et al., 2019) was considered large in 2019 and is three orders of magnitude smaller than several models trained since — but it is possible to be precise about the three quantities that define a training run's scale and how they trade off against each other: the number of trainable parameters N, the number of tokens D the model is trained on, and the total training compute C, measured in floating-point operations (FLOPs).

Table 1.1 gives parameter and token counts for several models referenced throughout this book, each figure as reported in the cited source.

Model	Parameters (N)	Training tokens (D)	Approx. training compute ($C \approx 6ND$, FLOPs)
GPT-1 (Radford et al., 2018)	117M	not precisely reported (BooksCorpus, ~7,000 books)	—
GPT-2 (Radford et al., 2019)	1.5B	not precisely reported (WebText, ~40GB of text)	—
GPT-3 (Brown et al., 2020)	175B	300B	3.15×10^{23}
PaLM (Chowdhery et al., 2022)	540B	780B	2.53×10^{24}
Chinchilla (Hoffmann et al., 2022)	70B	1.4T	5.88×10^{23}
LLaMA-65B (Touvron et al., 2023)	65B	1.4T	5.46×10^{23}

Two rows in the table are marked “not precisely reported”: the original GPT-1 and GPT-2 papers describe their training corpora (BooksCorpus; a scrape of outbound Reddit links called WebText) in terms of document counts and disk size rather than exact token counts, so no precise D or compute figure is given for them here rather than estimating one — an explicit caveat is preferable to an invented number. The other four rows use token counts stated directly in their respective papers.

The parameter count N. This is simply the number of scalar trainable weights in the network — every entry of every weight matrix and bias vector, summed across all layers. Chapter 9 (The Transformer Block and Full Architecture) derives a closed-form parameter-counting formula in terms of the architectural hyperparameters L (number of layers), d_{model} (hidden dimension), h (number of attention heads), d_{ff} (feedforward dimension), and $|\mathcal{V}|$ (vocabulary size); for now, treat N as a single number describing model size. N also determines memory footprint directly: storing parameters alone in 16-bit precision costs $2N$ bytes, so a 175-billion-parameter model requires 350 GB just for its weights, before accounting for gradients, optimizer state, or activations — several times more memory than fits on a single accelerator, which is why Chapter 14 (Distributed Training Systems) is necessary at all, and why Chapter 17 (Inference and Decoding) discusses quantizing weights to 8-bit or 4-bit precision to make inference feasible on smaller hardware.

The token count D. This is the number of tokens (roughly, subword pieces; see Chapter 5 for the exact tokenization procedure) the model’s parameters are updated against during pretraining — not the size of the available data pool, but how much of it (potentially with repetition) is actually shown to the optimizer. D matters independently of N: a model can be undertrained (too few tokens for its parameter count, leaving capacity on the table) or, in principle, trained past the point of benefit for its size. The precise relationship between how N and D should be jointly chosen for a fixed compute budget is the subject of the Chinchilla scaling analysis, covered fully in Chapter 13 (Scaling Laws and Compute-Optimal Training); the short version, visible already in Table 1.1, is that later models (Chinchilla, LLaMA) deliberately use far more training tokens per parameter than GPT-3 did, reflecting the finding that GPT-3-era models were, by that later analysis, substantially undertrained relative to their parameter count.

Training compute C. The standard approximation used throughout the scaling-laws literature (Kaplan et al., 2020; Hoffmann et al., 2022) is

$$C \approx 6ND, \tag{1.2}$$

where C is total training compute in FLOPs, N is the number of (non-embedding) parameters, and D is the number of training tokens. The factor of 6 has a concrete derivation, not just an empirical fit. For a Transformer, the dominant cost per token is the matrix multiplications in its linear layers; a matrix-vector product using a weight matrix with p entries costs approximately $2p$ FLOPs (one multiply and one add per entry), so processing a single token through the entire network's forward pass costs approximately $2N$ FLOPs — each parameter is touched by one multiply-accumulate per token. Backpropagation then costs roughly twice the forward pass: computing gradients with respect to the layer inputs costs about as much as the forward pass itself, and computing gradients with respect to the weights costs a comparable amount again, for a further $\approx 4N$ FLOPs per token. Summing forward and backward gives $2N + 4N = 6N$ FLOPs per token, and multiplying by D tokens gives Equation 1.2. This approximation deliberately ignores the $O(n^2 d)$ cost of the attention operation itself (Chapter 7), which is a real cost but is subdominant to the $O(nd^2)$ cost of the linear projections for the sequence lengths and model dimensions typically used in pretraining, and it ignores embedding and unembedding layers if a stricter "non-embedding parameter" count is used — both are the kind of tuned-constant caveats the original scaling-law papers themselves flag.

As a worked example, take GPT-3's reported configuration directly from Table 1.1: $N = 175 \times 10^9$ parameters, $D = 300 \times 10^9$ tokens.

$$C \approx 6 \times (175 \times 10^9) \times (300 \times 10^9) = 3.15 \times 10^{23} \text{ FLOPs.}$$

This matches the order of magnitude Brown et al. (2020) report directly for GPT-3's training run, which is a useful sanity check on the approximation. To make a number this large concrete, it helps to convert it into device-time. An NVIDIA A100 GPU has a peak bf16 tensor-core throughput of approximately 312×10^{12} FLOP/s; real training runs never sustain peak throughput, because of communication overhead, memory-bandwidth limits, and pipeline bubbles (Chapter 14 quantifies these), so achieved throughput is commonly reported as a fraction of peak called model FLOPs utilization (MFU), often in the 30–50% range for well-optimized large-scale training. At 40% MFU, GPT-3's compute budget corresponds to roughly 29,200 single-GPU-days; spread across a cluster of 1,024 A100s running in parallel, that is approximately 28.5 wall-clock days — consistent with the order of magnitude publicly reported for training runs of this scale. The code below implements this calculation directly, so you can recompute it for any (N, D) pair, including ones you plan yourself.

```
import math

def training_flops(n_params: float, n_tokens: float) -> float:
    """Approximate total training compute in FLOPs using the standard
    C ~= 6 * N * D rule (Kaplan et al., 2020; Hoffmann et al., 2022),
    where N is the parameter count and D is the number of training
    tokens. The factor of 6 comes from ~2 FLOPs per parameter for the
    forward pass (one multiply-accumulate per parameter, applied once
    per token) plus ~4 FLOPs per parameter for the backward pass
    (roughly twice the forward-pass cost, for gradients with respect
    to both activations and weights)."""
```

```
    return 6.0 * n_params * n_tokens


def gpu_days(total_flops: float, flops_per_sec: float,
             mfu: float = 0.4, n_gpus: int = 1) -> float:
    """Convert a FLOPs budget into wall-clock days on a cluster of
    n_gpus devices, given each device's peak FLOP/s and an achieved
    model FLOPs utilization (MFU) fraction of that peak."""
    effective_flops_per_sec = flops_per_sec * mfu * n_gpus
    seconds = total_flops / effective_flops_per_sec
    return seconds / 86400.0


models = {
    # name: (parameters, training tokens)
    "GPT-3 (175B params, 300B tokens)": (175e9, 300e9),
    "PaLM (540B params, 780B tokens)": (540e9, 780e9),
    "Chinchilla (70B params, 1.4T tokens)": (70e9, 1.4e12),
    "LLaMA-65B (65B params, 1.4T tokens)": (65e9, 1.4e12),
}

A100_BF16_PEAK_FLOPS = 312e12 # NVIDIA A100 bf16 tensor-core peak, FLOP/s

for name, (n_params, n_tokens) in models.items():
    flops = training_flops(n_params, n_tokens)
    days = gpu_days(flops, A100_BF16_PEAK_FLOPS, mfu=0.4, n_gpus=1024)
    print(f"{name:40s} C = {flops:.3e} FLOPs "
          f"~ {days:6.1f} days on 1024 A100s @ 40% MFU")
```

Running this reproduces the 3.15×10^{23} FLOPs and roughly 28.5-day figure computed above for GPT-3, and gives comparable figures for the other three models — PaLM's larger compute budget corresponds to roughly 229 days on the same cluster, for instance. The relationship $C \approx 6ND$ is the single equation that ties together every axis of "large": it says compute grows linearly in both parameters and tokens, so doubling either one (holding the other fixed) doubles the compute bill, and it is the quantity that Chapter 13's compute-optimal analysis takes as a fixed budget to be allocated between N and D.

The Three Phases: Pretraining, Adaptation, and Inference

Every LLM you can interact with today — a chat assistant, a code-completion tool, a document summarizer — has gone through the same three-phase lifecycle, and the chapters of this book are organized around exactly these three phases. Understanding the boundary between them now will make it clear, as you read later chapters in isolation, which phase's problem each chapter is solving.

Pretraining is where the autoregressive objective of Equation 1.1 is applied directly: a randomly initialized model is trained by gradient descent to minimize cross-entropy loss on next-token prediction across a large, broad, largely unlabeled text corpus — commonly hundreds of billions to trillions of

tokens, as Table 1.1 shows. Nothing about the objective during pretraining refers to any downstream task; the model is simply learning to predict what token comes next, across an enormous variety of contexts, and in doing so develops internal representations of syntax, factual associations, and reasoning patterns that later prove reusable. This is the single most expensive phase in both compute and data terms — it is the phase Equation 1.2's $C \approx 6ND$ estimate describes — and it produces what is usually called a *base model*: a model that is very good at continuing text plausibly but has no particular tendency to be helpful, follow instructions, or converse. Part IV of this book (Chapters 11–14) is entirely devoted to this phase: Chapter 11 (Pretraining: Objectives, Data, and Practice) covers the objective in more depth along with the data pipeline (crawling, deduplication, quality filtering) that produces the training corpus; Chapter 12 (Optimization at Scale) covers the specific optimizer (AdamW) and learning-rate schedules used to actually perform the minimization; Chapter 13 (Scaling Laws and Compute-Optimal Training) covers how to choose N and D for a fixed compute budget; and Chapter 14 (Distributed Training Systems) covers how a training run of this scale is physically distributed across thousands of accelerators.

Adaptation takes a pretrained base model and steers it toward a specific, useful behavior — typically "be a helpful, instruction-following assistant" — without repeating the enormous cost of pretraining. This happens in (potentially several) stages. Supervised fine-tuning (SFT) continues training the same cross-entropy objective, but on a much smaller, curated dataset of instruction-response pairs, which teaches the model the *format* of a helpful response. Because updating all N parameters is often unnecessary and expensive for this smaller-scale adaptation, parameter-efficient fine-tuning methods such as LoRA train a small number of additional parameters while leaving the pretrained weights frozen. Beyond format, alignment techniques such as reinforcement learning from human feedback (RLHF) and Direct Preference Optimization (DPO) further adjust the model using human (or AI-generated) *preferences* between candidate responses, rather than single target responses, to reduce undesirable behaviors and better match nuanced human judgment. Part V (Chapters 15–16) covers this phase: Chapter 15 (Fine-Tuning and Parameter-Efficient Adaptation) covers SFT and LoRA/QLoRA in full mathematical detail, and Chapter 16 (Alignment: RLHF, Reward Models, and DPO) derives the reward-modeling objective and the RLHF and DPO loss functions from first principles.

Inference is what happens every time a trained, adapted model is actually used to produce output — the iterative sampling procedure sketched in the first section of this chapter, applied at deployment time rather than during training. This phase has its own distinct engineering concerns, orthogonal to the ones that dominate training: how to decode a sequence from the model's conditional distributions (greedy, beam search, temperature and nucleus sampling), how to avoid recomputing already-processed context at every new token via key-value caching, how to reduce memory and latency through quantization, and how techniques like speculative decoding can produce multiple tokens per expensive forward pass. Chapter 17 (Inference and Decoding) covers this phase in full. A fourth topic that spans all three phases — how do you know whether any of this actually worked — is covered separately in Chapter 18 (Evaluation, Interpretability, and Limitations), which addresses evaluation methodology, mechanistic interpretability, and the known limitations of models built this way.

It is worth noting explicitly that these three phases differ enormously in compute cost and consequently in who performs them: pretraining a frontier-scale model costs the compute budgets in Table 1.1, measured in thousands of GPU-days, and is performed by a small number of organizations; adaptation costs orders of magnitude less (SFT and LoRA-style fine-tuning are often feasible on a handful of GPUs, sometimes even one); and inference, while cheap per query, is run continuously at massive aggregate scale and is where the majority of a deployed model's lifetime compute is typically spent. Everything specific to the *mathematics* of language modeling covered in Parts I–III of this book (the

factorization, the architecture, attention) is shared by all three phases; what differs across pretraining, adaptation, and inference is the objective being optimized (or, for inference, the decision procedure applied to an already-fixed distribution) and the systems constraints under which it runs.

Notation and Conventions Used Throughout This Book

The notation below is used identically in every chapter of this book, without exception. Each symbol is redefined at first use within each chapter (a reader may reasonably start from any chapter), but this table is the canonical reference and matches Appendix A, which consolidates every symbol used across the whole book.

Category	Convention	Examples
Scalars	italic lowercase	x, n, d
Vectors	bold lowercase	$\mathbf{x} \in \mathbb{R}^d$
Matrices	bold uppercase	$\mathbf{W} \in \mathbb{R}^{d \times k}$
Tensors of rank ≥ 3	bold sans-serif	T
Sets	calligraphic	$\mathcal{D}$ (a dataset), $\mathcal{V}$ (a vocabulary)
Model parameters (collectively)	θ	the full set of trainable weights
Sequence length	n (general) or T (Chapters 7–10 and 17 only, where position is treated as a time index)	—
Hidden / model dimension	d_{model}	—
Number of attention heads	h	—
Per-head dimension	$d_k = d_{\text{model}}/h$	—
Vocabulary size	$\lvert\mathcal{V}\rvert$	—
Number of layers	L	—
True / data distribution	$p(\cdot)$	$p(x_t \mid x_{<t})$
Model distribution	$p_\theta(\cdot)$	$p_\theta(x_t \mid x_{<t})$
Expectation	$\mathbb{E}$	$\mathbb{E}_{x\sim p}[\cdot]$
KL divergence	D_{KL}	$D_{\text{KL}}(p \,\|\, q)$
Entropy	H	$H(p)$
Cross-entropy	$H(p, q)$	$H(p, q) = -\mathbb{E}_{x\sim p}[\log q(x)]$

A few conventions follow from this table that are worth stating once. Subscript t (or, in the chapters that use it, T-indexed positions) always denotes a position within a sequence, so x_t is the t-th token and $\mathbf{h}_t$ is the hidden-state vector at position t, never an epoch or training-step index — training steps or iterations, when they appear, are written with a superscript in parentheses, $\theta^{(i)}$, to avoid collision. The subscript θ on p_θ always denotes "produced by the model with these parameters," which is what distinguishes a quantity that is estimated (subject to training) from a quantity, like $p(\cdot)$ without subscript, that describes the true, unknown data-generating process the model is trying to approximate. Batch dimensions in code are always listed first in tensor shape comments, i.e., `(batch, seq, d_model)`, matching PyTorch's default convention and the Code section of the accompanying style notes; the one

deliberate exception is when a chapter is specifically discussing a memory layout, such as a KV cache or a tensor-parallel sharding scheme, for which a different ordering is more natural — those cases are flagged explicitly when they occur.

The code below implements the chain-rule identity from the first section of this chapter directly, on a small enough example that you can inspect every number involved. It builds a toy joint distribution over sequences of length 3 from a vocabulary of size 3 by normalizing random weights, then verifies numerically that the product of conditionals derived from that joint (Equation 1.1's right-hand side) reproduces the joint itself (its left-hand side) for every possible sequence.

```
import numpy as np

rng = np.random.default_rng(0)

vocab_size = 3
seq_len = 3

# Ground-truth joint distribution p(x_1, x_2, x_3) as a (3, 3, 3)
# tensor, obtained here by normalizing random positive weights. In a
# real language model this joint is never available explicitly -it
# is only constructed here, at toy scale, to check that the chain-rule
# factorization used throughout the chapter is exact.
joint_weights = rng.random((vocab_size,) * seq_len)
joint = joint_weights / joint_weights.sum()


def conditional_from_joint(joint, prefix):
    """Return p(x_t | x_<t) for the given prefix (a tuple of already
    -chosen token indices), computed by slicing out the prefix and
    marginalizing and normalizing over the remaining positions."""
    t = len(prefix)
    if t == 0:
        return joint.sum(axis=tuple(range(1, seq_len)))
    slicer = tuple(prefix) + (slice(None),) * (seq_len - t)
    sub = joint[slicer]              # shape: (vocab_size,) * (seq_len - t)
    remaining_axes = tuple(range(1, sub.ndim))
    marginal = sub.sum(axis=remaining_axes) if remaining_axes else sub
    return marginal / marginal.sum()


def sequence_prob_autoregressive(joint, x):
    """Compute p(x_1, ..., x_n) as the product of conditionals
    (Equation 1.1's right-hand side), each derived from the same
    joint distribution used to compute the direct value below."""
    prob = 1.0
    for t in range(len(x)):
        cond = conditional_from_joint(joint, x[:t])
```

```
        prob *= cond[x[t]]
    return prob


# Check the chain rule against direct lookup for every possible
# sequence of length 3 over the toy vocabulary.
max_abs_error = 0.0
for x1 in range(vocab_size):
    for x2 in range(vocab_size):
        for x3 in range(vocab_size):
            x = (x1, x2, x3)
            direct = joint[x]
            chained = sequence_prob_autoregressive(joint, x)
            max_abs_error = max(max_abs_error, abs(direct - chained))

print(f"max |p(x) - prod_t p(x_t | x_<t)| = {max_abs_error:.2e}")
# max |p(x) - prod_t p(x_t | x_<t)| = 1.39e-17 (zero, up to floating-point
    error)
```

The output confirms, to floating-point precision, that multiplying the conditionals recovers the joint exactly — the chain rule is an identity, not an approximation. What a real language model does differently is that it never has access to `joint`: it only ever learns the conditionals directly, from example sequences, using a neural network in place of the `conditional_from_joint` lookup used here. Every architecture in Parts II and III of this book is, from this point of view, a different proposal for how to compute a good approximation to that conditional from an arbitrarily long prefix.

How This Book Is Organized

The book has five parts, each building on the ones before it, plus three reference appendices.

Part I — Foundations (Chapters 1–4) establishes the mathematical and software tools every later chapter assumes. This chapter has stated the central object (the autoregressive factorization) and the vocabulary and notation used throughout. Chapter 2 (Probability and Information Theory for Language Modeling) develops entropy, cross-entropy, KL divergence, and perplexity, and formally connects maximum-likelihood training to cross-entropy minimization — the derivation that this chapter's first section only sketched. Chapter 3 (Linear Algebra, Calculus, and Optimization Refresher) reviews the specific slice of linear algebra, backpropagation, and gradient-based optimization used by every later chapter, including a from-scratch autodiff engine. Chapter 4 (Neural Network Building Blocks) covers the generic deep-learning components — activation functions, normalization layers, residual connections, initialization — that Transformer-specific chapters later assume as given.

Part II — Representing Text (Chapters 5–6) covers how raw text becomes the tensors a neural network consumes. Chapter 5 (Tokenization: From Text to Tensors) derives byte-pair encoding and related subword schemes. Chapter 6 (Embeddings and Positional Encoding) covers how discrete tokens become vectors and, critically, how position is injected into an architecture — self-attention, introduced next, is permutation-invariant without it.

Part III — The Transformer Architecture (Chapters 7–10) is the mathematical heart of the book. Chapter 7 (Attention: Mathematical Foundations) derives scaled dot-product attention from first

principles, including why the scaling factor is necessary. Chapter 8 (Multi-Head Self-Attention, Implemented) extends this to the multi-head formalism used in practice and implements it completely in PyTorch. Chapter 9 (The Transformer Block and Full Architecture) assembles the full decoder-only block used by GPT-style models and derives the parameter-counting formula this chapter used informally in Table 1.1's discussion. Chapter 10 (Modern Architectural Variants) covers the departures from the original 2017 design that current production models actually use — KV caching, grouped-query attention, mixture-of-experts routing, and long-context extensions.

Part IV — Training at Scale (Chapters 11–14) covers the pretraining phase described above in full: the training objective and data pipeline (Chapter 11), the optimizer (Chapter 12), the scaling laws that determine how to spend a compute budget (Chapter 13, which returns to and formalizes the $C \approx 6ND$ relationship derived in this chapter), and the distributed-systems techniques required to actually run a training job of this size across many accelerators (Chapter 14).

Part V — Adaptation, Alignment, and Deployment (Chapters 15–18) covers everything downstream of a trained base model: parameter-efficient fine-tuning (Chapter 15), preference-based alignment via RLHF and DPO (Chapter 16), the decoding and inference-serving techniques used to actually run a model in production (Chapter 17), and evaluation, interpretability, and the field's known limitations (Chapter 18).

The three appendices — Notation Reference, Python Environment Setup, and Glossary — are meant to be consulted as needed rather than read in order, and the reference list at the end of the book collects every citation used across all eighteen chapters in one place. Chapters within each part depend on the chapters before them within that part, and later parts depend on earlier parts, but each chapter defines its symbols at first use, so a reader with the relevant background from Parts I–III can reasonably start from any chapter in Part IV or V directly.

Summary

- A language model computes $p_\theta(x_1, \ldots, x_n) = \prod_{t=1}^{n} p_\theta(x_t \mid x_{<t})$: the exact chain-rule factorization of the joint distribution over tokens, with the true conditionals replaced by a learned, parametric approximation. This factorization is simultaneously the training objective (next-token cross-entropy) and the generation procedure (iterative sampling).
- Masked language models such as BERT (Devlin et al., 2019) target bidirectional conditionals on a masked subset of positions rather than the chain-rule factorization, which makes them well suited to representation learning for classification and retrieval but not to open-ended autoregressive generation.
- Four architectural eras — n-gram counting, recurrent networks (RNNs and LSTMs, Hochreiter and Schmidhuber, 1997), the Transformer's self-attention (Vaswani et al., 2017), and GPT-scale decoder-only pretraining (Radford et al., 2018; Radford et al., 2019; Brown et al., 2020) — represent successive answers to how to condition on a long history without an exponential data requirement.
- "Large" is quantified by parameters N, training tokens D, and total training compute C; the standard approximation is $C \approx 6ND$ FLOPs, derived from $\approx 2N$ FLOPs per token for the forward pass and $\approx 4N$ for the backward pass. GPT-3's reported configuration ($N = 175$B, $D = 300$B) gives $C \approx 3.15 \times 10^{23}$ FLOPs, matching the order of magnitude reported by Brown et al. (2020).
- Every deployed LLM passes through three phases: pretraining (the expensive, broad-corpus application of the autoregressive objective; Chapters 11–14), adaptation (SFT, PEFT, and preference-

based alignment applied to a pretrained base model; Chapters 15–16), and inference (the decoding procedure applied at deployment time; Chapter 17), with evaluation and interpretability (Chapter 18) cutting across all three.
- The book's notation is fixed and consistent throughout: scalars italic lowercase, vectors bold lowercase, matrices bold uppercase, tensors of rank ≥ 3 bold sans-serif, sets calligraphic, parameters collectively θ, and $p(\cdot)$ versus $p_\theta(\cdot)$ distinguishing the true data distribution from the model's approximation to it.
- The book proceeds in five parts — Foundations, Representing Text, The Transformer Architecture, Training at Scale, and Adaptation/Alignment/Deployment — each building on the mathematical objects defined in the ones before it.

Exercises

1. Prove the chain-rule factorization of Equation 1.1 for a joint distribution over three discrete random variables (x_1, x_2, x_3), starting only from the definition of conditional probability $p(a \mid b) = p(a, b)/p(b)$. Then state, in one or two sentences, why the identity you proved holds for *any* fixed ordering of x_1, x_2, x_3, not only the order $1, 2, 3$.
2. Modify the `conditional_from_joint` / `sequence_prob_autoregressive` code from this chapter to use a vocabulary of size 4 and a sequence length of 4, and confirm the chain-rule identity still holds to floating-point precision. Then deliberately corrupt one entry of one of the conditional distributions (so the conditionals no longer come from a consistent joint) and report what happens to the sum of `sequence_prob_autoregressive(joint, x)` over all sequences `x` — is it still 1?
3. A masked language model in the style of BERT predicts a masked token x_i from the rest of the sequence, $x_{\mathcal{M}\setminus i}$, where $\mathcal{M}$ is the masked set. Explain concretely why you cannot use such a model to sample a full sentence left to right the way you can with an autoregressive model — what input would you have to supply for the very first token, before any other token's value is known?
4. Using the `training_flops` function from this chapter, compute the approximate training compute for a hypothetical model with $N = 7 \times 10^9$ parameters trained on $D = 2 \times 10^{12}$ tokens (roughly LLaMA-2 7B's published configuration). Then compute how many wall-clock days that would take on a cluster of 512 A100 GPUs at 35% MFU, using the `gpu_days` function.
5. Table 1.1 shows that Chinchilla (70B parameters) uses roughly 4.7 times as many training tokens as GPT-3 (175B parameters) while having less than half as many parameters, yet its total training compute is still lower than PaLM's. Using only the $C \approx 6ND$ relationship (not the full scaling-law derivation, which is deferred to Chapter 13), verify Chinchilla's compute figure in Table 1.1 by hand, and explain in one sentence why a smaller model trained on more tokens can still land at higher or lower total compute than a larger model, depending on the token count chosen.
6. For each of the three phases described in this chapter (pretraining, adaptation, inference), name one quantity that is fixed/frozen during that phase and one quantity that changes. (For example: during inference, is θ fixed or changing? Is the KV cache fixed or changing?)

Probability and Information Theory for Language Modeling

This chapter builds the probabilistic and information-theoretic vocabulary that the rest of the book relies on: entropy, cross-entropy, KL divergence, and perplexity, together with the precise sense in which training a language model is a maximum-likelihood problem. It closes with n-gram models, worked by hand and in code, as a concrete (and concretely limited) instance of the sequential density estimation problem that motivates everything from Chapter 4 onward.

Probability chain rule and autoregressive modeling

Chapter 1 introduced the factorization that every model in this book is built around: for a sequence of tokens $x_1, \dots, x_n$ drawn from a vocabulary $\mathcal{V}$,

$$p(x_1, \dots, x_n) = \prod_{t=1}^{n} p(x_t \mid x_{<t}),$$

where $x_{<t} = (x_1, \dots, x_{t-1})$ denotes the prefix preceding position t (with $x_{<1}$ the empty sequence). This is not a modeling assumption; it is the chain rule of probability, an identity that holds for *any* joint distribution over a finite or countable sequence of discrete random variables. It is worth pausing on why this identity, rather than some other decomposition, is the organizing principle of language modeling, because the answer explains both what makes the problem hard and why the neural architectures introduced starting in Chapter 4 take the shape they do.

Conditioning turns one hard problem into many merely-hard problems. The joint distribution $p(x_1, \dots, x_n)$ lives on a space of size $|\mathcal{V}|^n$ — for a sequence of length 50 and a vocabulary of 50,000 subword tokens, that is $|\mathcal{V}|^{50} = 50{,}000^{50}$, a number with roughly 235 digits. No amount of data lets you estimate a discrete distribution over an outcome space that large directly: almost every one of the $|\mathcal{V}|^n$ possible sequences will never be observed, so an unconditional model would place zero probability on every sequence a user actually types. The chain rule sidesteps this not by making the outcome space smaller — the joint distribution is exactly as large as before — but by re-expressing the estimation problem as a *sequence* of much smaller conditional estimation problems. At each position t, the object to be estimated is a single categorical distribution over $\mathcal{V}$, i.e., a vector of $|\mathcal{V}|$ nonnegative numbers summing to one. That is a tractable estimation target regardless of how large n is. The difficulty of the original problem has not disappeared; it has been relocated into the *conditioning*: the categorical distribution at position t must depend on $x_{<t}$, and $x_{<t}$ ranges over $|\mathcal{V}|^{t-1}$ possible histories. Section 5 of this chapter makes that relocated difficulty precise by counting parameters directly; the short version is that naively tabulating $p(x_t \mid x_{<t})$ for every possible history is exactly as infeasible as tabulating the joint distribution was, and the entire point of a parametric conditional model $p_\theta(x_t \mid x_{<t})$ — culminating in the Transformer of Part III — is to make that dependence on an unboundedly long, unboundedly varied history *learnable* from a manageable number of parameters, by sharing statistical strength across similar contexts rather than storing a separate number for each one.

Language modeling is fundamentally sequential density estimation, not classification. It is useful to state precisely what is being estimated at each step. Given a fixed prefix $x_{<t}$, $p(\cdot \mid x_{<t})$ is a genuine probability density (here, a probability mass function, since $\mathscr{V}$ is discrete) over the vocabulary — not a single "correct answer" to be classified, even though computing $\arg\max_{x_t} p_\theta(x_t \mid x_{<t})$ superficially resembles a classification decision at inference time. The training signal, however, is not a classification signal: the target is the full conditional distribution over next tokens, and the loss used to fit it (Section 4.2) is precisely a divergence between distributions, not a 0/1 classification error. This distinction matters because it explains why the objective used to train language models generalizes cleanly to the density-estimation view even though a single human-written training corpus supplies only one sample x_t per context $x_{<t}$ rather than an explicit distribution — a point taken up carefully in Section 4.

Exchangeability versus order-dependence. A sequence of random variables $x_1, \ldots, x_n$ is called *exchangeable* if its joint distribution is invariant under every permutation σ of the indices:

$$p(x_1, \ldots, x_n) = p(x_{\sigma(1)}, \ldots, x_{\sigma(n)}) \quad \text{for all permutations } \sigma.$$

Exchangeability is the formal property underlying "bag-of-words" style models, and much of classical statistics — including the i.i.d. sampling assumption — implicitly relies on it or the closely related property of independence. De Finetti's representation theorem shows that any infinite exchangeable sequence can be represented as a mixture of i.i.d. sequences: conditional on some latent parameter, the observations are independent draws. This is a powerful simplification when it applies, because it means order carries no information and the only estimation problem is estimating the mixing distribution.

Natural language is emphatically not exchangeable. "The dog bit the mailman" and "the mailman bit the dog" contain the same multiset of words and have wildly different — indeed, mutually exclusive in ordinary readings of the world — meanings. Word order encodes syntax (who is the subject versus the object of the verb), scope (negation, quantifiers), discourse structure (what is presupposed versus asserted), and much more. Formally, this means $p(x_1, \ldots, x_n) \neq p(x_{\sigma(1)}, \ldots, x_{\sigma(n)})$ for essentially every non-identity permutation σ once n is more than a couple of tokens, so no de Finetti-style reduction to an i.i.d. mixture is available, and no representation that discards order (a bag-of-words histogram, an unordered set of embeddings) can, even in principle, represent p exactly. This is the probabilistic justification for a fact that will resurface as an engineering problem in Chapter 6: the self-attention mechanism introduced in Chapter 7 is itself permutation-*equivariant* (it treats its input as a set unless told otherwise), so positional information must be injected explicitly — an architectural patch for exactly the property established here, that language is not exchangeable.

The chain-rule factorization is compatible with order-dependence precisely because it treats the sequence asymmetrically: x_t is conditioned on everything before it, not treated interchangeably with it. Note also that the chain rule requires no independence or Markov assumption of any kind — it is exact for arbitrary joint distributions, arbitrarily long-range dependencies included. Any *approximation* enters only when a model restricts how $p_\theta(x_t \mid x_{<t})$ may depend on $x_{<t}$: n-gram models (Section 5) restrict it to a fixed-width window; recurrent networks compress $x_{<t}$ into a fixed-size hidden state; Transformers, as will become clear in Part III, are distinguished by conditioning on the *entire* prefix through attention, subject only to a maximum context length imposed for computational reasons rather than modeling ones. The remainder of this chapter develops the tools needed to say precisely what it means for a conditional model $p_\theta(x_t \mid x_{<t})$ to be a *good* approximation to $p(x_t \mid x_{<t})$, and to show that fitting p_θ by maximum likelihood is exactly the same operation as minimizing a particular information-theoretic divergence between the two.

Entropy, cross-entropy, KL divergence

Everything in this section is stated for a discrete random variable x taking values in a finite set $\mathscr{V}$ under two distributions, p (typically the true or data distribution) and q (typically a model). In language modeling x will usually stand in for the next token x_t, conditioned implicitly on a fixed history $x_{<t}$, but the definitions themselves are general and say nothing about sequences. Logarithms are natural logarithms (ln) unless a base is stated; information measured this way is in *nats*. Measuring in bits (log base 2) rescales every quantity below by the constant factor $1/\ln 2 \approx 1.4427$ and changes no qualitative conclusion; nats are the convention used throughout this book because they compose naturally with the natural-log softmax and cross-entropy operations used in the deep learning framework code in later chapters.

Entropy. The (Shannon) entropy of a distribution p over $\mathscr{V}$ is

$$H(p) = -\sum_{x \in \mathscr{V}} p(x) \log p(x) = \mathbb{E}_{x \sim p}[-\log p(x)], \tag{2.1}$$

with the convention $0 \log 0 = 0$ (justified by the limit $\lim_{u \to 0^+} u \log u = 0$). $H(p)$ is the expected value, under p itself, of the *surprisal* $-\log p(x)$ of an outcome — an outcome that p assigns high probability is unsurprising ($-\log p(x)$ near 0), and one it assigns low probability is surprising ($-\log p(x)$ large). Entropy is therefore the average surprisal a well-informed observer — one who knows p exactly — experiences when sampling from p. Equivalently, by Shannon's source coding theorem, $H(p)$ (in bits) is the minimum expected number of bits per symbol needed to losslessly encode outcomes drawn from p, achieved by an optimal code that assigns each outcome x a codeword of length $-\log_2 p(x)$ bits; the more predictable the source, the shorter the optimal code, and the lower the entropy. $H(p) \geq 0$ always, with $H(p) = 0$ exactly when p is a point mass (the outcome is certain, hence never surprising), and $H(p)$ is maximized, at $\log |\mathscr{V}|$, by the uniform distribution over $\mathscr{V}$ (maximum uncertainty).

Cross-entropy. The cross-entropy of p relative to q is

$$H(p, q) = -\sum_{x \in \mathscr{V}} p(x) \log q(x) = \mathbb{E}_{x \sim p}[-\log q(x)]. \tag{2.2}$$

This is the expected surprisal of outcomes drawn from the *true* distribution p, but measured using the code (or, equivalently, the probabilities) prescribed by a possibly different distribution q. It answers: if the world generates outcomes according to p, but I evaluate them (or code for them) as if they came from q, what is my expected surprisal? $H(p, q) = H(p)$ exactly when $q = p$; whenever $q \neq p$, using the "wrong" distribution costs something on average, formalized below. Note $H(p, q)$ is generally not symmetric in p and q — $H(p, q) \neq H(q, p)$ in general — because the outer expectation is always taken with respect to the first argument.

KL divergence. The Kullback–Leibler divergence from q to p (read "the KL divergence of p from q," or informally "how far q is from p"), is

$$D_{\mathrm{KL}}(p \,\|\, q) = \sum_{x \in \mathscr{V}} p(x) \log \frac{p(x)}{q(x)} = \mathbb{E}_{x \sim p}\left[\log \frac{p(x)}{q(x)}\right]. \tag{2.3}$$

The decomposition $H(p, q) = H(p) + D_{\mathrm{KL}}(p \,\|\, q)$. This identity, used repeatedly for the rest of the book, follows from splitting the logarithm inside the cross-entropy sum:

$$H(p, q) = -\sum_x p(x) \log q(x) = -\sum_x p(x) \log q(x) + \sum_x p(x) \log p(x) - \sum_x p(x) \log p(x)$$

$$= -\sum_x p(x)\log p(x) \;+\; \sum_x p(x)\log p(x) - \sum_x p(x)\log q(x).$$

So

$$H(p,q) = H(p) + D_{\mathrm{KL}}(p\,\|\,q). \tag{2.4}$$

Cross-entropy decomposes into two pieces: $H(p)$, the entropy of the true distribution — an irreducible floor determined entirely by how inherently unpredictable the source is, with no dependence on q whatsoever — plus $D_{\mathrm{KL}}(p\|q)$, the *excess* cost incurred specifically because $q \neq p$. This is precisely the decomposition that makes cross-entropy the right training loss: since $H(p)$ is a constant with respect to any parameters used to define q, minimizing $H(p,q)$ over q and minimizing $D_{\mathrm{KL}}(p\|q)$ over q are the same optimization problem, differing only by an additive constant.

Non-negativity (Gibbs' inequality). $D_{\mathrm{KL}}(p\|q) \geq 0$ for all p, q, with equality if and only if $p = q$ everywhere p has support. This is why $H(p,q) \geq H(p)$ always — cross-entropy can never undercut the true entropy — and it follows from Jensen's inequality applied to the strictly convex function $-\log$. Write

$$D_{\mathrm{KL}}(p\,\|\,q) = \mathbb{E}_{x\sim p}\left[-\log\frac{q(x)}{p(x)}\right] \geq -\log\mathbb{E}_{x\sim p}\left[\frac{q(x)}{p(x)}\right] = -\log\sum_{x:\,p(x)>0} p(x)\cdot\frac{q(x)}{p(x)} = -\log\sum_{x:\,p(x)>0} q(x) \geq -\log 1 =$$

where the first inequality is Jensen's inequality for the convex function $u \mapsto -\log u$ applied to the random variable $q(x)/p(x)$, and the last inequality holds because $\sum_{x:\,p(x)>0} q(x) \leq \sum_{x\in\mathcal{V}} q(x) = 1$. Equality in Jensen's inequality for a strictly convex function requires the random variable $q(x)/p(x)$ to be almost surely constant under p, which combined with both distributions summing to 1 forces $q(x) = p(x)$ wherever $p(x) > 0$.

Why D_{KL} is asymmetric, and why the direction matters. In general $D_{\mathrm{KL}}(p\|q) \neq D_{\mathrm{KL}}(q\|p)$; swapping the two distributions with the toy numbers worked in Section 3 below changes the value (0.0351 versus 0.0371 nats — small here only because p and q happen to be close). The asymmetry is not a technical nuisance; it reflects a real, qualitative difference in what each direction penalizes, and which of the two directions appears in a training objective has direct consequences for the behavior of the resulting model.

$D_{\mathrm{KL}}(p\|q)$ — the "forward" KL divergence, with the expectation taken under the true distribution p — weights every outcome by how likely it is under p. Crucially, if $q(x) \to 0$ at some point x where $p(x) > 0$, the term $p(x)\log\left(p(x)/q(x)\right) \to \infty$: forward KL imposes an essentially unbounded penalty on q for failing to cover any region where p has real probability mass, however small. A q that minimizes forward KL is therefore driven to spread its probability mass over the *entire* support of p — it is often described as "mass-covering" or "mean-seeking" — because leaving any part of p's support uncovered is catastrophically expensive. By contrast, $D_{\mathrm{KL}}(q\|p) = \mathbb{E}_{x\sim q}[\log(q(x)/p(x))]$, the reverse direction, weights outcomes by q instead: it costs nothing for q to assign zero probability wherever q itself has decided to assign zero probability, even at points where $p(x) > 0$, since such points simply never get sampled under the expectation. A q that minimizes reverse KL is free to collapse onto a single mode of p and ignore the rest — reverse KL is "mode-seeking" or "zero-forcing."

This distinction is not academic for language models: as Section 4 shows, standard maximum-likelihood training minimizes (a sample estimate of) $D_{\mathrm{KL}}(p\,\|\,p_\theta)$ — forward KL, with the true data distribution p as the reference. This is one contributing reason why models trained this way tend to

spread probability across many plausible continuations of a prompt rather than collapsing onto a single "most likely" completion — collapsing would leave probability mass at some observed continuations at zero, which forward-KL training penalizes severely. It is also why sampling from p_θ can occasionally produce low-probability, atypical continuations: the training objective never drove those probabilities all the way to zero, because doing so would have required abandoning some (rare) training examples entirely. This asymmetry resurfaces in a different guise in Chapter 16 (Alignment: RLHF, Reward Models, and DPO), where the reinforcement-learning-from-human-feedback objective adds an explicit KL penalty term to keep a fine-tuned policy close to a reference model; which direction of KL is used there again has direct, documented consequences for how conservatively the fine-tuned model behaves, and is worth revisiting once this asymmetry is fresh.

Perplexity as an evaluation metric

Cross-entropy is the quantity language models are trained to minimize, but it is reported in nats (or bits) per token, units that are informative to an information theorist and opaque to almost everyone else. Perplexity is a monotonic reparameterization of cross-entropy chosen specifically to be more interpretable, and it is the metric actually quoted in nearly every language modeling paper and leaderboard.

Definition. Given the true distribution p and a model q, the perplexity of q with respect to p is defined as the exponentiated cross-entropy,

$$\mathrm{PPL}(p, q) = \exp\left(H(p, q)\right) = \exp\left(\mathbb{E}_{x\sim p}[-\log q(x)]\right). \tag{2.5}$$

In practice one never has access to the true distribution p that generates natural language, so $H(p, q)$ cannot be computed exactly. What is computable is an empirical estimate from a held-out corpus of text presumed to be sampled from p. Given a held-out sequence $x_1, \ldots, x_n$ and a trained autoregressive model p_θ, define the empirical average per-token negative log-likelihood,

$$\hat{H} = -\frac{1}{n}\sum_{t=1}^{n} \log p_\theta(x_t \mid x_{<t}),$$

which, by the chain rule, is exactly $-\frac{1}{n}\log p_\theta(x_1, \ldots, x_n)$, the negative log-probability the model assigns to the entire held-out sequence, normalized by length. By the law of large numbers, if the held-out tokens are representative samples from the true conditional distributions, $\hat{H}$ is a consistent (and unbiased in expectation, token by token) Monte Carlo estimate of the population cross-entropy $H(p, p_\theta)$ from the previous section, so the empirically reported perplexity is

$$\mathrm{PPL} = \exp(\hat{H}) = \exp\left(-\frac{1}{n}\sum_{t=1}^{n} \log p_\theta(x_t \mid x_{<t})\right). \tag{2.6}$$

This is the formula implemented in every language modeling codebase: run the model over held-out text, sum (or average) the per-token log-probabilities it assigns to the *actual observed* next tokens, negate, average by length, and exponentiate.

The effective-branching-factor intuition. Perplexity has a clean operational reading. Suppose a model q were exactly uniform over some set of k tokens at every step (assigning probability $1/k$ to each of k options and 0 elsewhere, and suppose for this thought experiment that the true next token always falls among those k). Then $H(p, q) = -\log(1/k) = \log k$, so $\mathrm{PPL} = \exp(\log k) = k$. Perplexity k therefore corresponds exactly to "the model is, on average, as uncertain as if it had to choose

uniformly among k equally likely options at each step" — the *effective branching factor* of the prediction task as seen by the model. A model with perplexity 20 on a token-level benchmark is, in this precise sense, behaving as if guessing uniformly among 20 possibilities at every position, even though its actual predictive distribution is (hopefully) far from uniform and highly peaked when the context is informative. Lower perplexity is better; perplexity is bounded below by 1 (achieved only by a model that assigns probability 1 to the correct token at every step — impossible on genuinely stochastic text) and bounded above, for a uniformly-random model, by $|\mathscr{V}|$ (perplexity equal to vocabulary size is the "no better than chance" baseline). Two further facts fall directly out of the decomposition in Section 2: because $H(p, q) = H(p) + D_{\mathrm{KL}}(p\|q) \geq H(p)$, no model can achieve a perplexity below $\exp(H(p))$, the *irreducible* perplexity of the data source itself — a floor set by the genuine unpredictability of language (synonym choice, factual underdetermination, stylistic freedom), not by any deficiency of the model. And because PPL is a strictly increasing function of $H(p, q)$, minimizing perplexity, minimizing cross-entropy, and minimizing $D_{\mathrm{KL}}(p\|q)$ during training are all exactly the same optimization problem viewed through three different units.

Worked example. Consider a toy vocabulary $\mathscr{V} = \{a, b, c, d\}$ with a hypothetical true next-token distribution $p = (0.5, 0.25, 0.125, 0.125)$ and a model's predicted distribution $q = (0.4, 0.3, 0.2, 0.1)$ at some context. By hand, using natural logs:

$$H(p) = -\sum_i p_i \ln p_i = 0.5 \ln \frac{1}{0.5} + 0.25 \ln \frac{1}{0.25} + 2 \times 0.125 \ln \frac{1}{0.125}$$

$$= 0.5(0.6931) + 0.25(1.3863) + 0.25(2.0794) = 0.3466 + 0.3466 + 0.5199 = 1.2130 \text{ nats},$$

$$H(p, q) = -\sum_i p_i \ln q_i = 0.5(0.9163) + 0.25(1.2040) + 0.125(1.6094) + 0.125(2.3026)$$

$$= 0.4581 + 0.3010 + 0.2012 + 0.2878 = 1.2481 \text{ nats},$$

so $D_{\mathrm{KL}}(p\|q) = H(p, q) - H(p) = 1.2481 - 1.2130 = 0.0351$ nats (positive, as Gibbs' inequality requires), and

$$\mathrm{PPL}(p, q) = e^{1.2481} \approx 3.484, \qquad \mathrm{PPL}(p, p) = e^{1.2130} \approx 3.364.$$

The model's perplexity (3.484) is higher than the irreducible perplexity of the source itself (3.364) by exactly the multiplicative factor $e^{D_{\mathrm{KL}}(p\|q)} = e^{0.0351} \approx 1.0358$ — the model behaves as though choosing among about 3.5 options instead of the source's own effective 3.4, a small but nonzero penalty for q not exactly matching p. Code Listing 1 below reproduces these numbers exactly, confirming the hand computation.

Maximum likelihood estimation as cross-entropy minimization

Chapter 1 stated that language models are trained to maximize the likelihood of a training corpus. This section makes the connection to the previous two sections precise: maximizing likelihood over a dataset is, term for term, identical to minimizing a cross-entropy — not merely analogous to it, but the same arithmetic expression.

Setup. Let $\mathcal{D} = \{x^{(1)}, \ldots, x^{(N)}\}$ be a training corpus of N token sequences (or, viewed at the token level after applying the chain rule, a very long stream of $(\textit{context}, \textit{next-token})$ pairs), each presumed to be an independent sample from the true data distribution p. The negative log-likelihood (NLL) of the corpus under a model p_θ is

$$\mathrm{NLL}(\theta) = -\sum_{i=1}^{N} \log p_\theta(x^{(i)}) = -\sum_{i=1}^{N} \sum_{t=1}^{n_i} \log p_\theta(x_t^{(i)} \mid x_{<t}^{(i)}),$$

where the second equality applies the chain rule from Section 1 to expand each sequence's log-probability into a sum of per-token conditional log-probabilities, and n_i is the length of sequence $x^{(i)}$. Training a language model by "minimizing the loss" means finding θ that minimizes $\mathrm{NLL}(\theta)$, equivalently that maximizes $\prod_i p_\theta(x^{(i)})$ — the standard maximum likelihood estimator (MLE).

The empirical distribution. Define the *empirical distribution* of the training corpus as the discrete distribution that places probability mass $1/N$ on each observed training example and zero elsewhere:

$$\hat{p}_{\text{data}}(x) = \frac{1}{N} \sum_{i=1}^{N} \mathbb{1}[x = x^{(i)}].$$

This is a legitimate probability distribution over the (enormous but finite, for finite sequence lengths and vocabulary) space of possible sequences: it is nonnegative and sums to 1. It is not the true distribution p that generates language — it is a specific, data-dependent approximation to p, supported only on the N examples actually observed (or fewer, if some sequences repeat).

The exact identity. Now compute the cross-entropy between $\hat{p}_{\text{data}}$ and the model p_θ, applying the cross-entropy definition from Section 2 with $\hat{p}_{\text{data}}$ in the role of p:

$$H(\hat{p}_{\text{data}}, p_\theta) = -\sum_x \hat{p}_{\text{data}}(x) \log p_\theta(x) = -\sum_x \left(\frac{1}{N} \sum_{i=1}^{N} \mathbb{1}[x = x^{(i)}] \right) \log p_\theta(x) = -\frac{1}{N} \sum_{i=1}^{N} \log p_\theta(x^{(i)}),$$

where the last step collapses the sum over all possible sequences x down to just the N observed ones, since the indicator is zero everywhere else. The right-hand side is exactly $\frac{1}{N}\mathrm{NLL}(\theta)$. So:

$$\frac{1}{N}\mathrm{NLL}(\theta) = H(\hat{p}_{\text{data}}, p_\theta). \tag{2.7}$$

This is not an approximation or an asymptotic argument — it is an algebraic identity, true for any finite N and any θ. Minimizing the average per-example negative log-likelihood over a training corpus *is*, by definition, minimizing the cross-entropy between the empirical distribution of that corpus and the model. Every training run described in this book that reports a "loss" computed with `nn.CrossEntropyLoss` or `F.cross_entropy` in PyTorch is, whether or not the implementer thinks of it this way, minimizing an empirical cross-entropy of exactly this form, with the token-level empirical distribution over $(x_{<t}, x_t)$ pairs seen during training standing in for $\hat{p}_{\text{data}}$.

Connecting to KL and to the asymmetry result. Applying the decomposition from Section 2 to this identity,

$$H(\hat{p}_{\text{data}}, p_\theta) = H(\hat{p}_{\text{data}}) + D_{\mathrm{KL}}(\hat{p}_{\text{data}} \,\|\, p_\theta).$$

The entropy $H(\hat{p}_{\text{data}})$ of the empirical distribution does not depend on θ at all — it is a fixed property of the training corpus once it is collected (for a corpus of N distinct examples, $\hat{p}_{\text{data}}$ is uniform

over those N points, so $H(\hat{p}_{\text{data}}) = \log N$). Consequently, minimizing $\text{NLL}(\theta)$ over θ, minimizing $H(\hat{p}_{\text{data}}, p_\theta)$ over θ, and minimizing $D_{\text{KL}}(\hat{p}_{\text{data}} \| p_\theta)$ over θ are all identical optimization problems — they differ only by the additive, θ-independent constant $H(\hat{p}_{\text{data}})$, which affects the numerical value of the loss but not the location of its minimizer. This closes the loop opened in Section 1: fitting p_θ to a corpus by maximum likelihood is, exactly, minimizing the *forward* KL divergence from the empirical data distribution to the model — the mass-covering direction identified in Section 2 — and, as the corpus grows large enough that $\hat{p}_{\text{data}}$ converges to the true generating distribution p (a consistency argument, not pursued formally here), this becomes asymptotically equivalent to minimizing $D_{\text{KL}}(p \| p_\theta)$, the population-level forward KL between the true distribution of language and the model.

This equivalence is the precise mathematical content behind the causal language modeling objective used throughout pretraining, developed operationally in Chapter 11 (Pretraining: Objectives, Data, and Practice): the loss minimized at every gradient step is a per-token cross-entropy, which by the identity above is simultaneously (a) the empirical negative log-likelihood of the training batch, (b) an unbiased Monte Carlo estimate of the population cross-entropy $H(p, p_\theta)$, and (c), up to the constant $H(\hat{p}_{\text{data}})$, the forward KL divergence $D_{\text{KL}}(\hat{p}_{\text{data}} \| p_\theta)$ from data to model. Three descriptions, one number.

N-gram models and their limits

Section 1 noted that any tractable language model must restrict how $p_\theta(x_t \mid x_{<t})$ depends on the history $x_{<t}$, since tabulating a distinct distribution for every possible history is combinatorially impossible. The n-gram model is the oldest and simplest such restriction, and although it has been superseded by neural approaches for essentially every serious application, it remains the clearest illustration of *why* a fixed-window, count-based approach fails at scale — an argument this section makes quantitative, because the same argument, run in reverse, is the strongest one-paragraph motivation for the neural architectures introduced starting in Chapter 4.

The Markov assumption and the model. An n-gram language model approximates the exact conditional $p(x_t \mid x_{<t})$ by conditioning only on the $n-1$ most recent tokens, an order-$(n-1)$ Markov assumption:

$$p(x_t \mid x_{<t}) \;\approx\; p(x_t \mid x_{t-n+1}, \dots, x_{t-1}). \tag{2.8}$$

(The terminology is that an "n-gram" model conditions on a window of $n-1$ prior tokens to predict the n-th; a *bigram* model, $n = 2$, conditions on one preceding token; a *trigram* model, $n = 3$, on two.) The joint probability of a full sequence is then approximated as a product of these fixed-window conditionals, $p(x_1, \dots, x_n) \approx \prod_{t=1}^{n} p(x_t \mid x_{t-n+1:t-1})$, which is a genuine simplification of the exact chain-rule factorization from Section 1 (recovering it only in the degenerate case $n \to \infty$).

Maximum likelihood estimation by counting. Given the Markov assumption, the natural estimator for $p(x_t \mid \text{context})$ is the relative frequency observed in a training corpus:

$$\hat{p}(x_t \mid c) = \frac{\text{count}(c, x_t)}{\text{count}(c)}, \tag{2.9}$$

where $c = (x_{t-n+1}, \dots, x_{t-1})$ is the conditioning context, $\text{count}(c, x_t)$ is the number of times the n-gram (c, x_t) appears in the training corpus, and $\text{count}(c) = \sum_{x' \in \mathcal{V}} \text{count}(c, x')$ is the number of times the context c appears (followed by anything). This is exactly the maximum likelihood estimator of a categorical distribution: it is straightforward to show, by Lagrange-multiplier maximization of

the multinomial log-likelihood subject to the constraint that probabilities for a fixed context sum to 1, that the relative-frequency estimator is the unique MLE — the same principle developed abstractly in Section 4, specialized to a model whose parameters are literally the count table.

Smoothing. The counting estimator has a fatal flaw: any n-gram (c, x_t) that never occurs in the training data gets estimated probability exactly zero. Since a full sequence's log-probability under the chain rule is a *sum* of individual conditional log-probabilities, a single zero-probability n-gram anywhere in an evaluation sequence sends $\log p_\theta(x_1, \dots, x_n) \to -\infty$ and the perplexity of Section 3 to $+\infty$ — for any held-out text of realistic length, this happens almost certainly, since novel combinations of even common words are the norm rather than the exception in naturally occurring text. Every practical n-gram model therefore applies *smoothing*, which reallocates some probability mass from observed n-grams to unobserved ones. The simplest scheme, additive (Laplace, or more generally add-k) smoothing, adds a constant $k > 0$ to every count before normalizing:

$$\hat{p}_{\text{add-}k}(x_t \mid c) = \frac{\text{count}(c, x_t) + k}{\text{count}(c) + k\,|\mathscr{V}|},$$

which guarantees every n-gram receives strictly positive probability (the denominator's $k|\mathscr{V}|$ term is exactly what is needed so the smoothed probabilities still sum to 1 over $\mathscr{V}$), at the cost of taking probability mass away from frequent, genuinely well-attested n-grams and handing it uniformly to n-grams the model has never seen, whether or not they are plausible. More refined schemes improve on this by making the reallocation depend on structure rather than treating every unseen n-gram identically. Katz backoff, for instance, redistributes the discounted probability mass from a higher-order model to a lower-order one — if the trigram (x_{t-2}, x_{t-1}, x_t) was never observed, back off to the bigram estimate $\hat{p}(x_t \mid x_{t-1})$, discounted and rescaled so the overall distribution still sums to 1 — on the reasoning that a bigram estimate, built from far more data, is a better guess than a uniform default. Kneser-Ney smoothing refines this further by weighting the lower-order backoff distribution not by raw frequency but by *how many distinct contexts* a word follows in, which better captures a word's general versatility rather than its raw frequency (a word like "Francisco" is frequent but follows almost only "San," and should receive a low backoff weight despite its high unigram count). These refinements matter enormously for the practical quality of n-gram models, but none of them address the structural problem developed next, which is a matter of parameter counting rather than estimator cleverness.

The combinatorial explosion, made quantitative. An n-gram model's parameters are, in effect, one probability value for every (context, next-token) pair — the count table itself is the model. The number of distinct contexts of length $n-1$ is $|\mathscr{V}|^{n-1}$, and for each context the model must specify a distribution over the next token, i.e., (up to the one normalization constraint per context) on the order of $|\mathscr{V}|$ further numbers, so the total number of free parameters scales as $\Theta(|\mathscr{V}|^n)$. Table 2.1 makes this concrete for a modern subword vocabulary of $|\mathscr{V}| = 50{,}000$ tokens (the rough size used by GPT-2-style byte-pair encodings, introduced formally in Chapter 5):

n (n-gram order)	context space $\lvert\mathscr{V}\rvert^{n-1}$	parameters $\sim \lvert\mathscr{V}\rvert^n$
2 (bigram)	5.0×10^4	2.5×10^9
3 (trigram)	2.5×10^9	1.25×10^{14}
4	1.25×10^{14}	6.25×10^{18}
5	6.25×10^{18}	3.125×10^{23}

Even the bigram model — conditioning on a single preceding token — already implies 2.5 billion parameters, comparable to a mid-sized modern neural language model, just to store one probability per

(previous-token, next-token) pair; a 5-gram model's parameter count, 3.125×10^{23}, exceeds Avogadro's number and is many orders of magnitude larger than the token count of any training corpus that has ever been assembled (the largest published LLM pretraining corpora are on the order of 10^{12}–10^{13} tokens). The overwhelming majority of the $|\mathcal{V}|^{n-1}$ possible contexts of any real length will therefore be seen zero or one times in even a very large corpus — the count table is catastrophically sparse — which is precisely why unsmoothed n-gram models assign zero probability almost everywhere on unseen text, and why even the best smoothing techniques can only patch over, not solve, the underlying sparsity: no amount of clever redistribution manufactures information about a context the model has never encountered.

There is a second, structural limitation independent of parameter count. The counting estimator treats each n-gram as an atomic, discrete symbol with no notion of similarity to any other n-gram: having observed "the cat sat on the mat" gives the model no information whatsoever about the probability of "the dog sat on the mat," even though "cat" and "dog" play near-identical distributional roles in English, because the contexts (the, cat, sat, on, the) and (the, dog, sat, on, the) are, to a count table, two unrelated keys with no shared structure. Fixing this requires exactly what an n-gram count table cannot provide: a representation in which similar contexts are *numerically* close, so that statistical evidence collected for one context automatically transfers to related ones. That is precisely the function of the distributed, continuous token embeddings introduced in Chapter 6 (Embeddings and Positional Encoding) and the neural conditional models built on top of them starting in Chapter 4 — replacing an exponentially large, sparse, unshared count table with a compact, shared, continuously parameterized function that can generalize across contexts it has never seen verbatim. The n-gram model's two failures — combinatorial parameter blowup and zero parameter sharing across similar contexts — are exactly the two problems the rest of Part I and Part II are built to solve.

Python: entropy, cross-entropy, perplexity, and a bigram model from scratch

The setup below is used for both code listings in this chapter; later chapters assume `torch`, `torch.nn as nn`, `torch.nn.functional as F`, `numpy as np`, and `math` are already imported and do not repeat this block.

```
import math

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

The first listing implements entropy, cross-entropy, KL divergence, and perplexity directly from the definitions in Sections 2 and 3, using only `numpy` and `math`, and checks them against the worked example computed by hand above.

```
def entropy(p, base=math.e):
    """Shannon entropy H(p) = -sum_x p(x) log p(x), in nats by default (base
        =math.e)."""
    p = np.asarray(p, dtype=np.float64)
```

```
    support = p > 0 # enforce the convention 0 * log(0) := 0
    return -np.sum(p[support] * np.log(p[support]) / np.log(base))


def cross_entropy(p, q, base=math.e):
    """Cross-entropy H(p, q) = E_{x~p}[-log q(x)]. Assumes q(x) > 0 wherever
        p(x) > 0."""
    p = np.asarray(p, dtype=np.float64)
    q = np.asarray(q, dtype=np.float64)
    support = p > 0
    return -np.sum(p[support] * np.log(q[support]) / np.log(base))


def kl_divergence(p, q, base=math.e):
    """D_KL(p || q) = H(p, q) - H(p), the decomposition derived in the text.
        """
    return cross_entropy(p, q, base=base) - entropy(p, base=base)


def perplexity_from_nats(cross_ent_nats):
    """Perplexity = exp(cross-entropy), when cross-entropy is measured in
        nats."""
    return math.exp(cross_ent_nats)


# Toy vocabulary {a, b, c, d}: true distribution p versus model prediction
    q
p = np.array([0.5, 0.25, 0.125, 0.125])
q = np.array([0.4, 0.3, 0.2, 0.1])

h_p = entropy(p)
h_pq = cross_entropy(p, q)
kl_pq = kl_divergence(p, q)

print(f"H(p)   = {h_p:.4f} nats")
print(f"H(p, q) = {h_pq:.4f} nats")
print(f"D_KL(p||q) = {kl_pq:.4f} nats")
print(f"PPL(p, q) = {perplexity_from_nats(h_pq):.4f}")
print(f"PPL(p, p) = {perplexity_from_nats(h_p):.4f}")
```

Running this prints `H(p)= 1.2130 nats`, `H(p, q)= 1.2481 nats`, `D_KL(p||q)= 0.0351 nats`, `PPL(p, q)= 3.4839`, and `PPL(p, p)= 3.3636`, matching the by-hand computation in Section 3 to four decimal places and confirming numerically that $\mathrm{PPL}(p, q) > \mathrm{PPL}(p, p)$: no model can beat the source's own irreducible perplexity, exactly as Gibbs' inequality guarantees.

The second listing builds a complete bigram language model — counts, additive smoothing, and held-out perplexity — implementing the estimator of Section 5 and the perplexity formula of Section 3 without any library shortcuts.

```
class BigramLanguageModel:
    """A bigram (order-1 Markov) language model with add-k smoothing."""

    def __init__(self, k: float = 1.0):
        self.k = k
        self.vocab = set()
        self.unigram_counts: dict[str, int] = {}
        self.bigram_counts: dict[tuple[str, str], int] = {}

    def fit(self, corpus: list[list[str]]) -> "BigramLanguageModel":
        """corpus: a list of token lists, each already wrapped with <bos> and
             <eos>."""
        for tokens in corpus:
            for i in range(len(tokens) - 1):
                w_prev, w_next = tokens[i], tokens[i + 1]
                self.vocab.add(w_prev)
                self.vocab.add(w_next)
                self.unigram_counts[w_prev] = self.unigram_counts.get(w_prev, 0)
                     + 1
                key = (w_prev, w_next)
                self.bigram_counts[key] = self.bigram_counts.get(key, 0) + 1
        return self

    def prob(self, w_prev: str, w_next: str) -> float:
        """p_theta(w_next | w_prev) under add-k smoothing (Section 5)."""
        v = len(self.vocab)
        num = self.bigram_counts.get((w_prev, w_next), 0) + self.k
        den = self.unigram_counts.get(w_prev, 0) + self.k * v
        return num / den

    def sequence_log_prob(self, tokens: list[str]) -> float:
        """sum_t log p_theta(x_t | x_{t-1}) over a <bos>/<eos>-wrapped token
            sequence."""
        return sum(
            math.log(self.prob(tokens[i], tokens[i + 1]))
            for i in range(len(tokens) - 1)
        )

    def perplexity(self, corpus: list[list[str]]) -> float:
        """Corpus-level perplexity: exp of the average per-token NLL (Section
             3)."""
        total_log_prob = 0.0
```

```
        total_tokens = 0
        for tokens in corpus:
            total_log_prob += self.sequence_log_prob(tokens)
            total_tokens += len(tokens) - 1 # predicted tokens, excludes
                leading <bos>
        avg_nll = -total_log_prob / total_tokens
        return math.exp(avg_nll)


def tokenize(sentence: str) -> list[str]:
    return ["<bos>"] + sentence.split() + ["<eos>"]


train_sentences = [
    "the cat sat on the mat",
    "the dog sat on the log",
    "the cat chased the dog",
    "the dog chased the cat",
]
held_out_sentences = [
    "the cat sat on the log",
    "the dog chased the mat",
]

train_corpus = [tokenize(s) for s in train_sentences]
held_out_corpus = [tokenize(s) for s in held_out_sentences]

bigram_lm = BigramLanguageModel(k=1.0).fit(train_corpus)
print(f"Vocabulary size |V| = {len(bigram_lm.vocab)}")
print(f"Train perplexity = {bigram_lm.perplexity(train_corpus):.3f}")
print(f"Held-out perplexity = {bigram_lm.perplexity(held_out_corpus):.3f}")
```

Running this listing gives `Vocabulary size |V| = 10`, `Train perplexity = 4.747`, and `Held-out perplexity = 4.943`. Two things are worth connecting back to the derivations above. First, the held-out perplexity is higher than the training perplexity, exactly the pattern Section 4's analysis predicts: the model's counts, and hence its estimated probabilities, are fit directly to the training corpus, so training-set bigrams like `(the, mat)` and `(sat, on)` are well-attested, while the held-out sentence "the cat sat on the log" contains the bigram `(on, the)`→`(the, log)` combination in a configuration the model has only seen partially, so its estimated probability — and hence held-out cross-entropy and perplexity — is somewhat worse than on data it was fit to; on a corpus of any realistic size this training/held-out gap becomes far more pronounced than in this ten-word-vocabulary toy example, precisely because of the sparsity argument made quantitative in Section 5. Second, note that `k=1.0` (add-one smoothing) is what keeps every held-out bigram's probability strictly positive and the perplexity finite at all — setting `k=0.0` and evaluating on any held-out bigram absent from the training counts would immediately produce a `ValueError` from `math.log(0.0)` inside `sequence_log_prob`, a direct, executable demonstration of the zero-probability problem Section 5 describes analytically.

Summary

- The chain rule $p(x_1, \ldots, x_n) = \prod_{t=1}^{n} p(x_t \mid x_{<t})$ is an exact identity, not an approximation; it converts one intractable joint-distribution estimation problem into n tractable-in-principle conditional (categorical) estimation problems, at the cost of relocating the difficulty into modeling dependence on an unboundedly large space of histories.
- Language is not exchangeable — order changes meaning — so no permutation-invariant (bag-of-words, de Finetti mixture) representation can capture p exactly; this is why positional information must be injected explicitly into permutation-equivariant architectures like self-attention (Chapters 6–7).
- Entropy $H(p)$, cross-entropy $H(p, q)$, and KL divergence obey the exact decomposition $H(p, q) = H(p) + D_{\mathrm{KL}}(p\|q)$; $D_{\mathrm{KL}}(p\|q) \geq 0$ always (Gibbs' inequality, via Jensen), so cross-entropy is bounded below by the true entropy.
- D_{KL} is asymmetric: forward KL $D_{\mathrm{KL}}(p\|q)$ is mass-covering/mean-seeking (penalizes q heavily for missing any of p's support), while reverse KL $D_{\mathrm{KL}}(q\|p)$ is mode-seeking/zero-forcing; standard language model training minimizes the forward direction.
- Perplexity is defined as $\mathrm{PPL} = \exp(H(p, q))$, empirically estimated as exp of the average per-token negative log-likelihood on held-out text; it has the operational meaning of an effective branching factor and is bounded below by $\exp(H(p))$, the irreducible perplexity of the data source.
- Minimizing negative log-likelihood over a training corpus is algebraically identical to minimizing the cross-entropy $H(\hat{p}_{\mathrm{data}}, p_\theta)$ between the empirical data distribution and the model, and — dropping a θ-independent constant — identical to minimizing $D_{\mathrm{KL}}(\hat{p}_{\mathrm{data}}\|p_\theta)$; this is the exact sense in which MLE-based LLM pretraining is cross-entropy/forward-KL minimization.
- n-gram models restrict $p(x_t \mid x_{<t})$ to a fixed window via a Markov assumption and estimate probabilities by counting, requiring smoothing (add-k, Katz backoff, Kneser-Ney) to avoid zero probabilities on unseen n-grams; their parameter count scales as $\Theta(|\mathscr{V}|^n)$, making even trigram models over realistic subword vocabularies computationally and statistically infeasible, which motivates the shared, distributed, continuous representations developed starting in Chapter 4.

Exercises

1. Prove that for a fixed context c, the relative-frequency estimator $\hat{p}(x_t \mid c) = \mathrm{count}(c, x_t)/\mathrm{count}(c)$ is the maximum likelihood estimator of a categorical distribution, by maximizing the multinomial log-likelihood subject to the constraint $\sum_x \hat{p}(x \mid c) = 1$ using a Lagrange multiplier.
2. Show algebraically that $D_{\mathrm{KL}}(p\|q) = D_{\mathrm{KL}}(q\|p)$ if and only if, for every $x \in \mathscr{V}$, $p(x) = q(x)$ or the ratio $p(x)/q(x)$ takes a specific relationship you should derive — or, more simply, construct a three-outcome numerical example (other than the one in this chapter) where the two directions differ by more than a factor of 2, and explain in words which distribution's support is being "protected" by the smaller of the two.
3. A held-out sequence of length $n = 1{,}000$ tokens is scored by a model and found to have average per-token cross-entropy $\hat{H} = 4.5$ nats. Compute the model's perplexity, its effective branching factor in bits-per-token (i.e., convert to log base 2), and the total number of nats required to encode the entire sequence under this model.
4. Extend the `BigramLanguageModel` class in Section 6 to a general `NGramLanguageModel(n, k)` that conditions on the preceding $n - 1$ tokens, with add-k smoothing, and reduces to

`BigramLanguageModel` when `n=2`. Train trigram ($n = 3$) and 4-gram models on a larger corpus of your choosing (e.g., a public-domain text file), and compare held-out perplexity across $n \in \{2, 3, 4\}$ — explain any perplexity *increase* you observe at higher n in terms of the sparsity argument from Section 5.

5. Using the parameter-counting argument from Section 5, compute the exact number of free parameters (accounting for the one normalization constraint per context, i.e., $|\mathcal{V}| - 1$ free values per context rather than $|\mathcal{V}|$) for a bigram, trigram, and 4-gram model over a character-level vocabulary of size $|\mathcal{V}| = 256$ (e.g., raw bytes) instead of the 50,000-token subword vocabulary used in the text. At what value of n does the character-level model's parameter count exceed that of the $|\mathcal{V}| = 50{,}000$ bigram model?
6. Derive the identity $H(p, q) = H(p) + D_{\mathrm{KL}}(p \| q)$ starting instead from the definition $D_{\mathrm{KL}}(p \| q) = \sum_x p(x) \log(p(x)/q(x))$ and solving for $H(p, q)$, to confirm you obtain the same relationship as the derivation in Section 2 that starts from $H(p, q)$ and splits the logarithm.