Inside Large Language Models

Foundations to Production

Ritesh Modi

Inside Large Language Models: Foundations to Production

First Edition

First printing: 2026

To the curious,

the ones who refuse to treat a model as a black box

and insist on seeing how it actually works.

Preface

Large language models moved from research curiosity to everyday tool in the span of a few years, and yet most of the people using them every day have no idea how they actually work. They type a prompt, the model produces fluent text, and the mechanism behind it stays hidden. That is a problem. When you treat a model as magic, you cannot debug its failures, improve its outputs, or decide when to trust it. You are stuck asking someone else for answers.

This book exists because the gap between "I use a language model" and "I understand how a language model works" is far smaller than it looks from the outside. The ideas at the core, tokens, embeddings, attention, transformer blocks, training loss, sampling, are explainable from first principles. You do not need a research background. You do not need a graduate math class. You need someone willing to walk through each concept slowly, with concrete numbers, real examples, and the patience to explain what every symbol means and why every design choice exists.

That is the book I wanted when I was learning. Every chapter starts with intuition before mechanics. Every concept gets a worked numerical example before any formula. Every piece of code comes with a line-by-line explanation and an execution-flow trace showing what happens under the hood. The book moves from the simplest building blocks, what a token is, what an embedding does, through single-head attention, multi-head attention, the complete transformer block, training loops, inference, alignment, and fine-tuning for real applications.

By the end of this book, you will have built a small transformer-based language model from scratch, trained it on real text, run inference on your own inputs, and fine-tuned a pretrained model for four concrete tasks. More importantly, when a new model is announced next year, you will be able to read its paper and know exactly which parts are genuinely new and which parts are familiar ideas wearing a new name.

What This Book Is About

This book teaches large language models from the ground up. Every abstraction is built from what came before it. The book starts with text, the raw material a language model consumes, and explains how text becomes numbers, how numbers become vectors in a learned space, how those vectors interact through attention, and how a stack of such interactions produces the next-token predictions that form the basis of every modern LLM.

The book is organized into nine parts. Part I (Chapters 1 through 3) builds the foundation: what a language model is, what the architectural families are, and the mathematical objects every subsequent chapter assumes. Part II (Chapters 4 and 5) introduces positional encoding and single-head attention, the mechanism at the heart of the transformer. Part III (Chapters 6 and 7) assembles the full transformer

block, with multi-head attention, residual connections, layer normalization, the feed-forward network, and the language-model head.

Part IV (Chapters 8 and 9) shifts from architecture to training: the loss function, backpropagation, optimizers, the data pipeline, and a capstone chapter that puts every piece together to train a complete GPT-style model from scratch. Part V (Chapter 10) covers inference, how a trained model actually produces text, token by token, with the optimizations that make it feasible in production. Part VI (Chapter 11) covers post-training alignment: supervised fine-tuning, reward modeling, and reinforcement learning from human feedback.

Part VII (Chapters 12 and 13) covers fine-tuning theory and the inference optimizations used to serve these models at scale. Part VIII (Chapters 14 through 17) is the applied portion: four hands-on fine-tuning projects taking a pretrained model through a contract-classification task, a legal-document assistant, a natural-language-to-SQL system, and a function-calling agent. Part IX (Chapter 18) ties every concept in the book into a single coherent picture so you leave with one mental model, not eighteen fragmented ones.

What This Book Is Not

This book is not a framework tutorial. It is not a manual for any specific library, and it deliberately keeps the teaching independent of whichever framework happens to be popular this year. Frameworks change. The ideas in this book do not. When a code example needs to run, it runs, but the point of the code is always to illustrate a concept, not to showcase an API.

This book is not a research survey. It does not attempt to catalogue every model variant or every paper published in the last five years. It covers the ideas that have survived long enough to matter, the core architectural choices, the training objectives, the alignment techniques, and the fine-tuning methods that appear again and again across systems. Once you understand those, reading any recent paper becomes a matter of spotting what is new and what is familiar.

This book is not an API reference. It does not list every argument to every function. It does not enumerate every configuration option. Reference material is best consumed as reference material, on a web page, searchable, up to date. This book teaches the concepts the reference assumes you know.

This book is not about deploying models in production infrastructure. It explains how inference works, including the optimizations that production systems rely on, but questions about cluster orchestration, autoscaling, request routing, and cost accounting belong to a different discipline with its own dedicated resources.

Who This Book Is For

This book is for anyone who wants to genuinely understand how large language models work. You do not need prior machine learning experience. You do not need a deep math background. You should be comfortable with Python at an intermediate level, functions, classes, list comprehensions, and working with libraries , and you should have enough high-school algebra to follow along when a concept uses a formula. Every mathematical object that appears in the book is explained from scratch the first time it is introduced.

Software engineers who have been told "add an LLM to the product" will find this book invaluable because it teaches you what you are actually adding. Data scientists and analysts working adjacent to modeling will find the concept-first approach lets them move from using models to building and tuning them. Recent graduates and career switchers will find a self-contained path from basic Python to a working understanding of the architecture that underpins every major LLM released today.

If you are already building with LLMs, calling APIs, writing prompts, stitching together retrieval-augmented applications, this book fills in the mechanisms behind the tools you use every day. The chapters on fine-tuning (Chapters 12 through 17) are particularly relevant if you want to move beyond prompt engineering into actually customizing models for your own domain.

How to Use This Book

The book is designed to be read front to back. Each chapter builds on what came before. The cross-references are specific, when a chapter points you back to "Chapter 5 (Single-Head Attention), Section 5.3," it means exactly that section, and your understanding of the current chapter will be deeper if you have that earlier material in mind.

That said, the book supports different reading paths. If you already understand the transformer architecture at a high level but want to see every component taught from first principles, read Chapters 1 through 9 carefully and skim the applied fine-tuning chapters (14 through 17) based on interest. If you are here primarily to fine-tune models for your own tasks, read Chapter 11 (Post-Training) and Chapter 12 (Fine-Tuning Theory) carefully, then pick the applied chapter closest to your problem. If you want to understand inference and serving, Chapter 10 and Chapter 13 are self-contained enough to read as a pair.

Every chapter ends with a set of hands-on exercises ("Try It Yourself") and a set of scenario-based knowledge-check questions. The questions are deliberately not definitional, they present a situation and ask you to reason about what the model or the training loop will do. Answers for every knowledge-check question across all eighteen chapters are in Appendix E. Work through the questions before checking the answers. Attempting a question and getting it wrong builds more understanding than reading the correct answer directly.

Every non-trivial code block in the book is followed by an Execution Flow trace , a small step card that shows the call tree the runtime follows when your code runs, with the tensor shapes at each step. The traces are how you build intuition for what is actually happening underneath the one-line API calls. Reading them is one of the fastest ways to internalize the architecture.

Conventions Used in This Book

Typography

Body text is set in Garamond. Code in the body text appears in `Courier New`, for example, softmax(), d_model, and attention_mask. When you see a term in monospace font, it refers to a specific variable name, function, tensor, or configuration in the code.

Code Blocks

Standalone code examples appear in numbered listings with a color-coded bar on the left that indicates the semantic role of each section, setup, forward pass, loss computation, or output. Every listing number follows the pattern "Listing [chapter].[sequence]", Listing 5.3 is the third code block in Chapter 5. When the text refers to a specific listing, it uses this numbering so you can find it quickly.

Figures and Diagrams

Diagrams are numbered per chapter, Figure 9.2 is the second diagram in Chapter 9. The book uses a consistent four-color system across every diagram, table, and value display. Every chapter includes a color legend at the bottom, but the system is summarized here in full:

Color System

	Color	Meaning	Used For
0	Blue	Input / original / source	Input tensors, token IDs, source values, cached values
1	Amber	Computed / transformed / derived	Intermediate activations, scores, derived metrics
2	Green	Output / result / final	Final tensors, predicted tokens, loss

			values, results
3	Purple	Aggregated / architectural	Batch-level summaries, module outputs, architecture elements

The color coding is semantic, not decorative. When you see a blue-shaded value in a step card, it came from the input. Amber means it was computed in that step. Green is the final result. Purple is an aggregated or architectural-level value. The colors make multi-step transformations traceable at a glance.

Callout Boxes

Three kinds of callout boxes appear throughout the book, each with a distinct visual treatment:

> *Insight boxes (green left border) surface deeper connections, real-world applications, and the moments where a concept clicks into place. They connect what you just learned back to the bigger picture of how a language model works.*

> *Gotcha boxes (amber left border) warn about common mistakes, surprising behavior, and the traps that catch experienced engineers. When a gotcha says "this will silently produce wrong outputs," it means exactly that, pay attention.*

> *Error boxes (red left border) show the exact error message you will see when something goes wrong, followed by what caused it and how to fix it. These are the errors you will actually encounter in practice, with the diagnoses you will find yourself reaching for again and again.*

Step Cards and Execution Flow Traces

Multi-step processes, tokenization pipelines, attention computations, training loops, are shown as a sequence of step cards. Each card has a numbered pill badge (Step 1, Step 2, and so on) and a title describing what happens in that step. The pill color signals the nature of the step: amber for computation, blue for input or cached data, green for output, and purple for aggregated or architectural operations.

After every non-trivial code example you will find an Execution Flow trace, a step card with a green pill that shows the complete call tree underneath the code. The traces include the tensor shapes at each step so you can follow not just what is being called but what is flowing through. They are the fastest way to build real intuition for how these models actually execute.

Tables and Data Displays

Tables are numbered per chapter, Table 10.1 is the first table in Chapter 10. When showing tensor contents or intermediate values, the book uses styled tables with color-coded cells rather than raw printouts, because the semantic coloring carries meaning about where each value came from and where it is going.

Cross-References

When the book points back to material covered elsewhere it always includes the chapter number, chapter name, and section: “as we saw in Chapter 5 (Single-Head Attention), Section 5.3.” This specificity is deliberate. It lets you flip to the exact spot if you need a refresher, and it lets you skip the reference if the concept is already fresh. The book never says “as mentioned earlier” without telling you exactly where.

Companion Code and Data

All code examples from this book are available in a companion code repository. Each chapter has its own directory with independently runnable Python scripts. The scripts include any setup code that is omitted from the book to keep the listings focused on the concept being taught. Setup blocks are clearly marked so you can tell them apart from the teaching examples.

The repository also includes the small datasets used throughout the book, the toy corpora used for tokenization examples, the short training set used in the capstone chapter, and the task-specific datasets used in the applied fine-tuning projects. A requirements file pins the exact library versions that the book was tested against, so reproducing any example is a matter of cloning the repository, installing the requirements, and running the script.

To run the examples, you need Python 3.10 or later and a machine with at least 16 GB of RAM. Most of the code can be executed on a laptop CPU, and where a GPU makes a meaningful difference the book calls it out and provides a lighter-weight fallback. The capstone training chapter and the larger fine-tuning projects are written to work on either a modest consumer GPU or a free-tier cloud notebook, with configuration options for faster hardware when you have it.

How to Contact Us

Getting the details right matters. If you find a technical error, a code example that does not run as written, an explanation that confuses more than it clarifies, or a typo that makes you stop reading, I want to hear about it. Errata and corrections are tracked in the companion repository's issues section. Before opening a new issue, check the existing ones to see if somebody has already reported the same thing.

For general questions about the book, suggestions for future editions, or to share something you built after reading it, you can reach me at www.riteshmodi.com or through the companion repository. Messages from readers who shipped something real after working through the book are the ones I enjoy most. I read every one.

If you are using this book in a course, training program, or reading group and would like supplementary material, extra exercises, slide decks, or dataset variations, please reach out. I am glad to support educators teaching the next round of engineers who are going to build with these models.

Acknowledgments

A book like this is shaped by more people than any list can name. It is shaped by every engineer who asked a question that made me realize I did not understand something as well as I thought. It is shaped by every junior colleague who pushed back on an explanation until I had to rebuild it from first principles. It is shaped by every debate over a whiteboard about what a particular gradient actually meant.

Thank you to the open-source communities that built and continue to build the tools the whole field runs on. The pace at which ideas turn into working code in this space is only possible because thousands of contributors have chosen to share their work freely. Much of what this book teaches was only made teachable because somebody else built a reference implementation you could read.

Thank you to the reviewers who read early drafts, ran every example, challenged my explanations, and caught the errors that would otherwise have shipped. The mistakes that remain are mine, not theirs.

Thank you to every reader who chose this book over the many alternatives now available. Writing a technical book is an act of optimism, a bet that your particular way of framing an idea will click for someone in a way that other resources did not. If this book helps you build something you are proud of, debug a problem that had you stuck, or land an opportunity you wanted, that is the entire point.

And finally, thank you to my family, who tolerated the long evenings and early mornings this book demanded, who listened patiently while I explained attention mechanics over dinner, and who never suggested, even once, that maybe I had enough hobbies already.

Table of Contents

Part V: Inference

How a trained model actually produces text, token by token.

Chapter 10

LLM Inference: From Prompt to Generated Text

Welcome to Volume II. In Volume I, you built a complete transformer-based language model from scratch and trained it to predict the next token, finishing in Chapter 9 with a working GPT you understood end to end. Training was fast, parallel, and elegant: the model saw entire sequences at once, computed loss across every position simultaneously, and updated its weights in one big backward pass. But now that training is done, a question looms that most tutorials gloss over: how do you actually use this trained model to generate text? The answer turns out to be surprisingly different from training, and understanding that difference is the key to understanding why language models are slow, why they are expensive, and why an entire industry of optimization techniques exists to make them practical. This chapter takes you inside the **inference** engine, the machinery that turns a trained model into the text-generating systems you interact with every day when you use ChatGPT, Claude, or any other **Large Language Model (LLM)**.

By the end of this chapter, you will understand the **autoregressive generation** loop and why inference must generate tokens one at a time instead of all at once. You will learn about the **Key-Value (KV) cache**, the single most important inference optimization, and we will compute a complete two-step generation by hand with real matrices and real numbers, so the KV cache becomes something you can verify with a calculator rather than an abstract concept you have to take on faith. You will master five sampling strategies (**greedy decoding**, **temperature** scaling, **top-K sampling**, **top-P sampling**, and **beam search**) that control whether generated text is deterministic, creative, or somewhere in between. And you will understand why Graphics Processing Unit (GPU) inference is **memory-bound** rather than **compute-bound**, which explains every production optimization from **Grouped Query Attention (GQA)** to **paged attention** to **speculative decoding**.

We will start with the fundamental difference between training and inference, then introduce the KV cache with a concrete scenario and a step-by-step algorithm. Sections 10.3 and 10.4 are the heart of this chapter: full worked examples tracing the **prefill** and **decode** phases through every matrix multiplication with actual numbers. From there we will cover the sampling strategies that convert raw model scores into actual token choices, then move to the hardware reality of why inference is so wasteful and the production optimizations that fix it. Building on the transformer architecture you built in Volume I, the self-attention mechanism from Chapter 5 (Single-Head Attention), the multi-head architecture from Chapter 6 (Multi-Head Attention, Residuals and Layer Norm), the complete transformer from Chapter 7 (Feed-Forward Network, LM Head and The Complete Transformer), and the training pipeline from Chapter 8 (Training:

Loss, Backpropagation, Optimizers and Data Pipeline), this chapter completes your understanding of the full LLM lifecycle from training to text generation.

10.1 Autoregressive Generation: How Inference Differs from Training

Before we look at any code or any optimization, we need to understand a fundamental constraint that shapes everything about inference. Here is the problem: during training, the model sees the entire sequence at once. If you are training on the sentence "The cat sat on the mat," the model receives all six tokens simultaneously and predicts the next token at every position in parallel. This is possible because of a trick called teacher forcing, which we covered in Chapter 8 (Training, Section 8.3): the model always receives the correct previous token as input, regardless of what it would have predicted. Teacher forcing lets training be massively parallel; the GPU can crunch through all positions at the same time, keeping its thousands of arithmetic cores busy.

Inference does not have that luxury. During inference, there is no ground truth, no teacher providing the correct answer at each step. The model must generate tokens one at a time, and each new token depends on every token that came before it. You cannot predict the fifth word until you know the fourth, and you cannot know the fourth until the model has predicted it. Think of it like writing a story: you cannot write the ending before you have written the middle, because the ending depends on what happened in the middle. This sequential dependency is not a software limitation we could fix with better code; it is a fundamental property of **autoregressive** language models. The word "autoregressive" literally means "self-feeding": each output becomes the input for the next step. This one constraint, the inability to generate tokens in parallel, is the reason inference is slower than training, the reason it is more memory-intensive, and the reason every optimization in this chapter exists.

To make this concrete, imagine you ask a chatbot "What is the capital of France?" The model does not instantly produce "The capital of France is Paris." in one step. Instead, it first processes your entire question (this is called the prefill phase), then generates "The" as the first output token. Then it takes your question plus "The" and generates "capital." Then it takes the growing sequence and generates "of." Then "France." Then "is." Then "Paris." Then a period. Then a special end-of-sequence token that tells it to stop. Each step requires a full forward pass through the model, every layer, every attention head, every weight matrix. For a 7-billion-parameter model, that means loading and processing 14 gigabytes of weights for every single token. And the model has to do this dozens or hundreds of times to generate a response. That is the inference problem in a nutshell, and it is why inference optimization matters so much.

Step 1 Training vs Inference: The Fundamental Difference

The table below summarizes every major difference between training and inference. Understanding these differences explains why inference requires its own set of optimization techniques.

	Property	Training	Inference	Why It Differs
0	Token processing	All positions in parallel	One new token per step	Teacher forcing vs autoregressive dependency
1	Causal mask	Explicit, applied in software	Automatic, future tokens don't exist	Nothing to mask when tokens haven't been generated
2	Batch size	Typically 128-1024 sequences	Often 1; continuous batching for many users	Real-time latency vs throughput trade-off
3	Compute profile	Compute-bound (GPU cores busy)	Memory-bound (weights must be loaded)	Per-step work is tiny; bandwidth is the bottleneck
4	K, V matrices	Recomputed fresh every forward pass	Cached and reused across steps	K, V for past tokens never change
5	Loss computation	Cross-entropy across all positions	None, pure forward pass only	No labels during generation

Reading the table top to bottom, the takeaway is that almost every property training and inference share is the same except for what the model has access to and what it spends its compute on; the rest of this section makes that asymmetry concrete with a teaching analogy.

Training is like a classroom exam where every student gets the answer key alongside the questions. They can all work in parallel because they are not waiting on each other; the teacher has given them all the information they need. Inference is like an improv comedy show: each performer's line depends on what the previous performer just said. Nobody can rehearse their line in advance because it depends on whatever just happened on stage. That sequential dependency is what makes improv unpredictable and entertaining, but it is also what makes it slow compared to a rehearsed performance. The same is true for LLM inference.

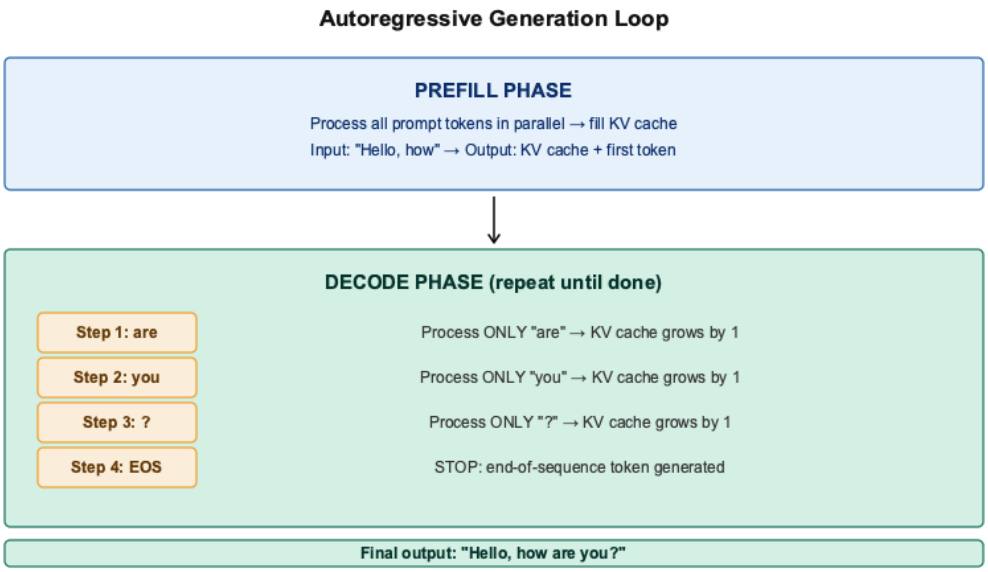
The Two Phases of Inference: Prefill and Decode

Every inference request naturally splits into two computationally distinct phases, and understanding this split is the single most important thing you can learn in this section. The first phase is called prefill. During prefill, the model processes your entire input prompt, all of it at once, in parallel, just like training. If you type a 500-word question, all 500 words (roughly 600-700 tokens) get processed simultaneously in one forward pass. The model computes the Key and Value vectors for every prompt token and stores them in what will become the KV cache. Prefill also produces the first output token. Prefill is fast because it is compute-bound: the GPU is doing big matrix-matrix multiplications that keep its arithmetic cores busy, just like during training.

The second phase is decode. During decode, the model generates one new token at a time. Each decode step processes exactly one token, the token that was just generated in the previous step. The model computes Q, K, and V for this single new token, then uses the KV cache to attend to all previous tokens without recomputing their K and V vectors. Decode is slow because it is memory-bandwidth-bound: at each step, the GPU loads all 14 gigabytes of model weights (for a 7B model) to perform a tiny matrix-vector multiplication for just one token. The arithmetic takes microseconds; loading the weights takes milliseconds. The GPU spends over 90 percent of each decode step just waiting for data to arrive from memory. Prefill determines **Time To First Token (TTFT)**, how long the user waits before seeing any output. Decode determines tokens per second, how fast the response streams after that first token appears.

Before we go deeper into prefill and decode, keep one name in your head: the KV cache. It is the single optimization that makes modern inference practical, and the whole next section is dedicated to it. For now, know that when we say the model stores Key and Value vectors during prefill, we mean it is saving partial attention computations to disk-fast GPU memory so it does not have to redo that work for every future token. Section 10.2 unpacks what Keys and Values are and why caching them is legal.

Here is a helpful way to think about it. Prefill is like reading a letter; you can scan the whole thing at once. Decode is like writing a reply; you have to think of each word based on what you have written so far. You can read fast, but writing takes time. That is exactly the asymmetry between prefill and decode, and it explains why a 2000-token prompt processes in 120 milliseconds but generating a 200-token response takes 4 seconds. The response is not slower because something is broken; it is slower because each token in the response requires a separate forward pass through the entire model.



Autoregressive generation loop: prefill processes the prompt in parallel, then decode generates tokens one at a time.

The diagram shows the shape of the loop; now we need to see its skeleton in source form. Seeing the control flow as a small code sketch makes the two-phase structure concrete and gives you something to match against when we later trace real matrices through it.

The Autoregressive Generation Loop in Code

Now that you understand the two-phase structure conceptually, let us see it in code. The following pseudocode shows the complete generation process for producing "Hello, how are you?" from a 3-token prompt. Pay attention to how the prefill phase handles all prompt tokens at once while each decode step processes exactly one new token.

vocab_example.py: usage

```
# Prompt: "Hello, how"  -> token IDs e.g. [15496, 11, 703]

# === PREFILL PHASE === (process all prompt tokens at once)
prompt_tokens = tokenize("Hello, how")                    e.g. [15496, 11, 703]

logits, kv_cache = model.prefill(prompt_tokens)           parallel forward pass
next_token = sample(logits[-1])                           -> "are"

# === DECODE PHASE === (one new token per step)
```

```
generated = [next_token]
while next_token != EOS_TOKEN and len(generated) < max_new_tokens:
    # Process ONLY the new token; reuse all cached K, V

    logits, kv_cache = model.decode_step(next_token, kv_cache)
    next_token = sample(logits)                                  sample from vocab distribution

    generated.append(next_token)

# Step 1 (decode): "are"  -> only "are" processed, not "Hello, how"
# Step 2 (decode): "you"  -> only "you" processed, KV cache now has 4 positions
# Step 3 (decode): "?"    -> only "?" processed
# Step 4 (decode): <EOS>  -> stop
# Final output: "Hello, how are you?"
```

Let us walk through this code carefully, because every line teaches something important about how inference works. Line 1 converts the user's text into token IDs using the model's tokenizer, the same tokenizer used during training. Line 2 is the prefill phase: all three prompt tokens are fed to the model in a single forward pass, and the model returns **logits** (raw scores for every **token** in the **vocabulary**) plus a KV cache containing the Key and Value vectors for all three prompt positions. Line 3 samples the first output token from the logits at the last position; we only care about the last position's prediction because that is where the model predicts what comes after the full prompt. The sample() function is where all the sampling strategies we will cover later (greedy, temperature, top-K, top-P) come into play.

Lines 5-12 are the decode loop. Each iteration processes exactly one token. Line 8 calls model.decode_step() with just the single new token and the existing KV cache. Internally, the model computes Q, K, and V for this one token, appends the new K and V to the cache, and runs attention between the new query and all cached keys. Line 9 samples the next token from the resulting logits. Line 10 appends it to the growing list. The loop continues until either the model produces an End-of-Sequence (EOS) token (meaning the model has decided it is done) or we hit the maximum generation length. Without the max_new_tokens limit, a model could theoretically generate forever; the EOS token is what lets it "decide" it is finished.

How the autoregressive generation loop executes

```
generate("Hello, how", max_new_tokens=100)
  |
  +-- PREFILL
  |   +-- tokenize("Hello, how") -> [15496, 11, 703]
  |   +-- model.prefill([15496, 11, 703])
  |   |   +-- embed all 3 tokens              (3, d_model)
  |   |   +-- for each layer (32 layers):
  |   |   |   +-- attention(X)                 (3, d_model) -> (3, d_model)
  |   |   |   |   +-- Q, K, V = X @ W_Q/K/V   each (3, d_k) per head
  |   |   |   |   +-- scores = Q @ K.T / sqrt  (3, 3) attention matrix
  |   |   |   |   +-- apply causal mask
  |   |   |   |   +-- weights = softmax(scores)
  |   |   |   |   +-- output = weights @ V     (3, d_k)
  |   |   |   +-- kv_cache.store(K, V)         cache K,V for all 3
positions
  |   |   |   +-- ffn(x)                       (3, d_model) -> (3, d_model)
  |   |   +-- logits = lm_head(x[-1])          (vocab_size,)
  |   +-- sample(logits) -> "are"
  |
  +-- DECODE LOOP
      +-- Step 1: decode_step("are", kv_cache)
      |   +-- embed 1 token                    (1, d_model)
      |   +-- for each layer:
      |   |   +-- Q_new = x @ W_Q              (1, d_k) -- ONE query
      |   |   +-- K_new, V_new = x @ W_K/V     (1, d_k) -- ONE key, value
      |   |   +-- kv_cache.append(K_new, V_new) cache now has 4 positions
      |   |   +-- scores = Q_new @ K_all.T     (1, 4) -- attend to ALL
      |   |   +-- output = softmax(scores) @ V_all  (1, d_k)
      |   +-- logits = lm_head(x)              (vocab_size,)
      |   +-- sample -> "you"
      +-- Step 2: decode_step("you", kv_cache)  -> "?"
      +-- Step 3: decode_step("?", kv_cache)    -> <EOS>
      +-- STOP (EOS detected)
```

> *During training, we apply an explicit causal mask to prevent token i from attending to token $j > i$. During inference, this mask is unnecessary during decode; when generating the fifth token, tokens six, seven, and eight simply do not exist yet. There is nothing to mask. The causal constraint that required careful software engineering during training comes for free from the sequential structure of generation itself. The mask is still needed during prefill (where multiple tokens are processed at once), but during decode it vanishes entirely. Understanding this deepens your grasp of why inference code looks simpler than training code.*

The generation loop above shows the mechanical process: prefill, sample, append, repeat. But there is a massive efficiency problem hiding in this loop that we have not addressed yet. At each decode step, computing attention requires the Key and Value vectors for every previous token. Without any optimization, the model would recompute all of those K and V vectors from scratch at every step, work that grows quadratically with sequence length. That is where the KV cache comes in, and it is the single most important optimization in all of LLM inference. Let us see exactly why naive inference is so wasteful, and how the KV cache fixes it.

10.2 The KV Cache: Turning $O(n^2)$ Recomputation into $O(n)$

Imagine you are a student taking notes in a lecture. At the end of each minute, you need to review everything the professor has said so far to understand the new point being made. A reasonable approach: keep your notes and just add the new minute's content. A terrible approach: throw away all your notes every minute and re-listen to the entire lecture from the beginning. That terrible approach is exactly what naive LLM inference does, and the KV cache is the equivalent of keeping your notes. It is such a simple idea that it might seem obvious, but its impact is enormous: for generating 512 tokens from a 512-token prompt, the KV cache reduces total computation by roughly 500 times. That is the difference between a response that takes 45 seconds and one that takes under a second.

Here is the problem in plain English. At each decode step the model recomputes attention across every previous token, redoing work that never changes. By token five hundred the model is reprocessing all five hundred tokens just to add one new one; by token a thousand the per-step work has doubled. The total work across a five-hundred-token decode sums to roughly seven hundred eighty thousand attention operations. Every one of those operations after the first is wasted, because the K and V vectors for already-generated tokens never change. The KV cache makes that waste visible, then eliminates it.

Here is the problem in precise terms. In self-attention, computing attention for a sequence of length n requires multiplying the Query matrix (n × d_k) by the Key matrix (n × d_k) transposed: an (n × d_k) × (d_k × n) matrix multiplication producing an n × n attention score matrix. That is $O(n^2)$ work. Without caching, every decode step at position t must process the full sequence of length t. At step 512 alone (generating the 512th token from a 512-token prompt), the model would need to process a 1024-token sequence. Summing across all 512 decode steps, the total work is roughly $O(n \cdot m^2)$ where n is prompt length and m is generation length; that works out to approximately 786,000 matrix-vector operations. With caching, the total is about 1,500 operations. That is not a typo. The difference really is that large.

The Key Insight: Why K and V Never Change

The KV cache works because of a mathematical property that is worth understanding deeply, not just memorizing. In self-attention, the Key and Value vectors for a token at position i are computed as: K[i] = X[i] · W_K and V[i] = X[i] · W_V, where X[i] is the input embedding for token i. Here is the critical observation: once a token is committed to the sequence and placed at a specific position, its embedding X[i] never changes. The token "love" at position 1 will always have the same embedding vector, whether we are at decode step 3 or decode step 300. And since the weight matrices W_K and W_V are fixed after training (we are not updating weights during inference), the result of X[i] · W_K is always the same for a given token at a given position. There is literally no reason to recompute it.

Recall from Chapter 4 (Positional Encoding and Attention Projections) that we defined three separate projection matrices (W_Q, W_K, and W_V), each learning a different role. The Query projection asks "what am I looking for?" The Key projection says "what information do I contain?" The Value projection says "what information should I pass along if someone attends to me?" The KV cache exploits the fact that a token's answers to "what information do I contain?" and "what should I pass along?" never change once that token is placed in the sequence. Only the Query changes at each step, because the query represents "what is the current token looking for right now," and the current token is always different.

Think of it like a library. The books (Values) and their catalog entries (Keys) do not change every time a new patron (Query) walks in. The library does not reshelve and re-catalog every book for each visitor. It catalogs them once, and each new visitor simply looks up what they need. The KV cache does the same thing: it computes each token's K and V vectors once, stores them, and lets each new token's Query look up what it needs from the stored Keys and Values. This is not an approximation or a shortcut; the cached values are mathematically identical to what a full recomputation would produce. The cache is provably correct.

Step 2 **What the KV Cache Stores**

For each transformer layer (say 32 for a 7B model), for each attention head (say 32 heads), for each position seen so far (grows by 1 each decode step), the cache stores two vectors: K[i] of dimension d_k (typically 128) and V[i] of dimension d_v (typically 128). That is 2 vectors × d_k floats × 2 bytes (fp16) = 512 bytes per position per head per layer.

For LLaMA 2 7B: 32 layers × 32 heads × 128 dims × 2 (K+V) × 2 bytes = 524,288 bytes ≈ 0.5 MB per token. A 4096-token context window uses 0.5 MB × 4096 ≈ 2.1 GB of KV cache memory.

Knowing what sits in the cache explains the "what," but the payoff is on the compute side. The next card shows precisely which work disappears at every decode step because those stored vectors are already waiting.

Step 3 **What the KV Cache Saves at Each Decode Step**

At decode step t, the model needs K and V for positions 0 through t. Positions 0 through t−1 are already in the cache, zero recomputation. Only position t requires fresh K and V computation: just 2 matrix-vector products per head per layer. The attention computation is a dot product between one new query and all cached keys: O(t × d_k) work, linear in sequence length, not quadratic.

With the savings quantified, we can now pin down the exact algorithm every production engine runs at each decode step. The sequence below is short, but every line encodes a choice that keeps attention mathematically correct while reducing cost from quadratic to linear.

KV Cache Strategy: Step by Step

Here is exactly what happens at each decode step. This is the core algorithm behind every fast LLM inference engine, from vLLM (a high-performance open-source LLM serving engine) to TensorRT-LLM (NVIDIA's optimized inference library). Understanding these steps means you understand the heart of production LLM serving.

Step 4 **Decode Step t: The Complete Algorithm**

```
1. Embed: Get X[t] = token_embedding[new_token] + positional_encoding[t]
2. Compute Q: Q[t] = X[t] · W_Q   (dimension: 1 × d_k)
3. Compute K: K[t] = X[t] · W_K   (dimension: 1 × d_k)
4. Compute V: V[t] = X[t] · W_V   (dimension: 1 × d_v)
5. Append K[t] and V[t] to the cache  →  cache now has t+1 entries
6. Attention scores: score[j] = Q[t] · K[j] / √d_k   for j = 0..t
7. Softmax: weights = softmax(scores)   → how much to attend to each
position
8. Output: weighted sum of all cached V vectors: output = ∑ weights[j] ×
V[j]
```

Steps 2-4 each involve a single matrix-vector multiplication, cheap. Step 6 is a dot product between one query vector and all cached key vectors: $O(t \times d_k)$. Step 8 is a weighted sum over all cached value vectors: also $O(t \times d_v)$. Both grow linearly with sequence length. Not quadratically.

Those eight algorithmic steps describe what should happen; writing them in PyTorch reveals which details the framework hides for us and which ones we still have to handle. The implementation that follows is short enough to trace step by step, but it is structurally identical to what vLLM and TensorRT-LLM do under the hood.

KV Cache Implementation in PyTorch

The following code implements a minimal but complete KV cache and an attention function that handles both prefill (processing all prompt tokens at once) and decode (processing one new token at a time). Read this carefully; this is the exact pattern used by real inference engines, just simplified to its essential structure.

kvcache.py: KVCache (class)

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

class KVCache:
    """Stores K and V tensors for all previous positions in one attention head."""
    def __init__(self):

        self.k = None    # shape: (batch, num_cached_tokens, d_k)
        self.v = None    # shape: (batch, num_cached_tokens, d_v)

    def update(self, new_k, new_v):
        """Append new K, V vectors. Called once per decode step."""

        if self.k is None:
            self.k, self.v = new_k, new_v                     # first call (prefill)
        else:
            self.k = torch.cat([self.k, new_k], dim=1)        # append along seq dim
            self.v = torch.cat([self.v, new_v], dim=1)

        return self.k, self.v

def attention_with_kv_cache(x, W_Q, W_K, W_V, cache, d_k):
    """
    x:     (batch, seq_len, d_model) -- seq_len=N for prefill, seq_len=1 for decode
    cache: KVCache instance -- stores all previous K, V
    """

    Q     = x @ W_Q                                           # (batch, seq_len, d_k)
    K_new = x @ W_K                                           # (batch, seq_len, d_k)
    V_new = x @ W_V                                           # (batch, seq_len, d_v)
```

```
    # Append new K, V to cache and retrieve ALL cached K, V
    K_all, V_all = cache.update(K_new, V_new)        (batch, total_len, d_k)

    # Attention: Q attends to ALL cached K (including just-appended)
    scores = Q @ K_all.transpose(-2, -1) / d_k ** 0.5        (batch, seq_len, total_len)

    # Causal mask: each position can only attend to positions <= itself
    total_len = K_all.size(1)
    seq_len   = Q.size(1)

    # During decode (seq_len=1), the mask is trivially all-True

    # During prefill (seq_len=N), we need the standard causal mask
    if seq_len > 1:
        mask = torch.triu(torch.ones(seq_len, total_len),
                          diagonal=total_len - seq_len + 1).bool()

        scores.masked_fill_(mask, float('-inf'))

    weights = F.softmax(scores, dim=-1)        (batch, seq_len, total_len)
    output  = weights @ V_all        (batch, seq_len, d_v)

    return output

# --- Usage: prefill + decode ---
cache = KVCache()
d_model, d_k = 4096, 128
W_Q = torch.randn(d_model, d_k)
```

```
W_K = torch.randn(d_model, d_k)
W_V = torch.randn(d_model, d_k)

# Prefill: process all prompt tokens at once
prompt = torch.randn(1, 512, d_model)                    512-token prompt

out = attention_with_kv_cache(prompt, W_Q, W_K, W_V,
cache, d_k)

# cache.k is now (1, 512, 128) -- all 512 K vectors
stored

# Decode step 1: process ONE new token

new_token = torch.randn(1, 1, d_model)                   single token

out = attention_with_kv_cache(new_token, W_Q, W_K, W_V,
cache, d_k)

# cache.k is now (1, 513, 128) -- one K vector appended
# Q was (1, 1, 128) -- attended to all 513 K vectors
# Total work: 1 matrix-vector multiply + 513 dot
products
# Without cache: would recompute all 513 K and V vectors
from scratch
```

Before the usage block even runs, look at the shapes the comments promise. cache.k starts at None, becomes (1, 512, 128) after prefill, then grows to (1, 513, 128) after one decode step. The 512 is the prompt length, the 128 is d_k (the per-head key dimension), and the 1 is batch size. Every decode step adds exactly one row to dimension 1 and nothing else. That is the entire story of KV caching in one shape.

A subtle but important design choice is that KVCache stores nothing about which layer or which head it belongs to. One cache holds K and V for a single attention head in a single layer. In a real model you need num_layers × num_heads caches (or, more commonly, one cache per layer holding all heads stacked along an extra dimension). The class is intentionally minimal so you can see the mechanics; production code trades this simplicity for batch dimensions, head dimensions, and a fixed-size tensor allocated up front to avoid the concat-in-a-loop pattern that allocates fresh memory on every single decode step.

The usage block also slips in a common beginner mistake. Notice that W_Q, W_K, W_V are created with torch.randn but never wrapped in torch.no_grad() and never flagged requires_grad=False. In a real inference pipeline you would always disable gradient tracking; otherwise PyTorch builds an autograd graph for every token generated, which does nothing useful during inference, consumes memory linearly with sequence length, and silently slows decode by a factor of two or more. The first time you deploy a model

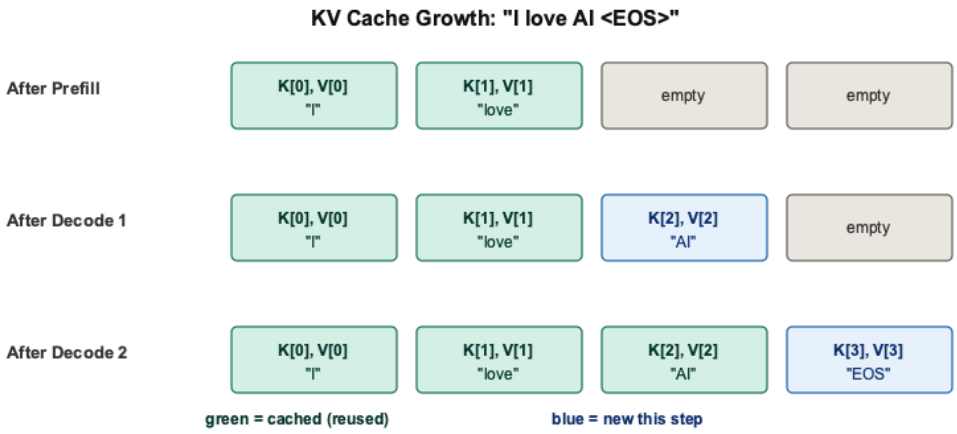
without wrapping generation in torch.no_grad(), you will discover this the hard way when OOM errors start appearing at the 2000-token mark.

Let us walk through this code carefully. The KVCache class is remarkably simple: it just stores two tensors (k and v) and provides an update() method that either initializes them (on the first call, during prefill) or appends new entries (during decode). The torch.cat() operation on line 13 concatenates the new K vector along the sequence dimension, growing the cache by one position. In production, systems pre-allocate contiguous memory and use index-based writes instead of repeated concatenation to avoid allocation overhead; we will see this when we cover **PagedAttention** later in the chapter.

The attention_with_kv_cache() function handles both prefill and decode with the same code path. During prefill, x has shape (batch, N, d_model) where N is the prompt length, so Q, K_new, and V_new all have N rows. During decode, x has shape (batch, 1, d_model), so only one row is computed. The key insight is on line 25: after updating the cache, K_all and V_all contain the full history: all cached positions plus the new one(s). The **attention scores** on line 28 compute Q @ K_all.T, which means each query attends to the full cached history. During decode with seq_len=1, the **causal masking** on lines 32-35 is skipped entirely because there is only one query position, and it can attend to everything in the cache, since all positions are in the past. This is exactly the insight we discussed earlier: the causal mask vanishes during decode because future tokens do not exist.

How a single decode step executes with KV cache

```
attention_with_kv_cache(new_token, W_Q, W_K, W_V, cache, d_k)
  |
  +-- Q     = new_token @ W_Q          (1, 1, 128)   -- 1 query vector
  +-- K_new = new_token @ W_K          (1, 1, 128)   -- 1 key vector
  +-- V_new = new_token @ W_V          (1, 1, 128)   -- 1 value vector
  |
  +-- cache.update(K_new, V_new)
  |   +-- self.k = cat([old_k, K_new]) (1, 513, 128) -- cache grows by 1
  |   +-- self.v = cat([old_v, V_new]) (1, 513, 128)
  |
  +-- scores = Q @ K_all.T / sqrt(128) (1, 1, 513)   -- 1 query vs 513 keys
  +-- seq_len=1, so causal mask SKIPPED (nothing to mask)
  +-- weights = softmax(scores)        (1, 1, 513)   -- attention weights
  +-- output  = weights @ V_all        (1, 1, 128)   -- weighted sum of 513
values
```



KV cache growth: green slots are cached from previous steps, blue slots are newly computed.

The prefill/decode split in the code is worth highlighting. During prefill, use_cache=True with no existing cache: all positions are processed at once, and K/V are stored. During decode, the existing cache gets passed in, only the new token's K/V are appended, and the query attends to the full cached history. Prefill is a matrix-matrix multiply (compute-bound, fast). Decode is a matrix-vector multiply (memory-bandwidth-bound, slow). This asymmetry is fundamental and drives every optimization later in this chapter.

> *Common mistake: Forgetting to pass the KV cache between decode steps. If you create a new empty cache at each step, you lose all cached K and V vectors, and the model effectively has no memory of previous tokens. The output will be incoherent garbage. The cache must persist across all decode steps for a given generation request.*

Understanding why K and V are immutable tells us that caching is mathematically safe, and reading the code shows us how caching is implemented, but numbers on a page can still feel abstract until you trace them yourself. In the next two sections, we will walk through the complete prefill and decode phases with concrete values, working through every matrix multiplication using real embeddings, real weight matrices, and real attention scores. By the end, you will be able to point at any number in the KV cache and explain exactly where it came from.

10.3 Worked Example: The Prefill Phase, Processing "I love"

Every number in this section is worked out by hand. We will process a 2-token prompt ("I love") with a 2-head attention layer, compute all Q, K, V matrices, run causal attention, and store the KV cache for the decode phase. This is exactly what a real transformer does, just with d_model = 4 instead of 4096. The mechanics are identical to what happens inside a 70-billion-parameter model; only the dimensions are smaller so every number fits on the page.

Why are we doing this? Because understanding the KV cache in the abstract is one thing. Seeing it work with actual numbers is another. By the end of this section, you will be able to point at a specific cell in the attention score matrix and explain exactly how it was computed from the weight matrices and input

embeddings. More importantly, you will see exactly which K and V vectors get stored in the cache, and why we will never need to recompute them during the decode steps that follow.

Setup: Vocabulary, Weights, and Input

We are using a minimal but complete example: a vocabulary of 5 tokens, d_model = 4, 2 attention heads with d_k = 2 each. The weights below are pre-trained and loaded from a checkpoint; no gradient computation happens during inference. Think of these as the numbers that training produced, frozen forever.

Step 1 **Vocabulary and Embedding Matrix E (5 × 4)**

```
E[0] = [ 0.0,  0.0,  0.0,  0.0]   <PAD>
E[1] = [ 0.8,  0.3, -0.2,  0.5]   "I"
E[2] = [ 0.4,  0.6,  0.7, -0.3]   "love"
E[3] = [ 0.5, -0.4,  0.6,  0.8]   "AI"
E[4] = [ 0.0,  0.0,  0.0,  0.0]   <EOS>
```

Positional Encodings PE (simplified):

```
PE[0] = [0.0, 1.0, 0.0, 1.0]   (position 0)
PE[1] = [0.8, 0.5, 0.1, 1.0]   (position 1)
PE[2] = [0.9,-0.4, 0.2, 1.0]   (position 2)
```

Input X[i] = E[token_id] + PE[position]

With the vocabulary, embeddings, and positional encodings laid out, we can assemble the first object the attention layer will see: the input matrix X for the two prompt tokens.

Step 1: Build Input Matrix X from "I love"

Each token gets its embedding vector added to its positional encoding, producing the input matrix X. This is the same embedding + positional encoding operation from Chapter 4 (Positional Encoding and Attention Projections, Section 4.1).

Step 2 **Build X: Embedding + Positional Encoding**

```
X[0] = E[1] + PE[0] = [0.8, 0.3, -0.2, 0.5] + [0.0, 1.0, 0.0, 1.0] = [0.8, 1.3, -0.2, 1.5]

X[1] = E[2] + PE[1] = [0.4, 0.6,  0.7,-0.3] + [0.8, 0.5, 0.1, 1.0] = [1.2, 1.1,  0.8, 0.7]


X = [[0.8, 1.3, -0.2, 1.5],    # "I" at position 0
     [1.2, 1.1,  0.8, 0.7]]    # "love" at position 1
```

Now that we have X, the rest of the layer is a chain of matrix multiplications. The next step runs the first three of them, projecting each input row into the query, key, and value subspaces that Head 1 will use.

Step 2: Compute Q1, K1, V1 for Head 1

Each position is projected through the pre-trained weight matrices W_Q1, W_K1, W_V1 (each 4×2) to produce query, key, and value vectors. These are exactly the Q, K, V projections from Chapter 5 (Single-Head Attention, Section 5.2). The difference is that here we are computing them during inference, not training, no gradients, no weight updates, just pure forward computation.

Step 3 **Q, K, V Projections for Head 1**

W_Q1, W_K1, W_V1 are (4×2) projection matrices (pre-trained)

```
Q1[0,0] = 0.8×0.2 + 1.3×0.1 + (-0.2)×0.3 + 1.5×0.4 = 0.16+0.13-0.06+0.60 = 0.83

Q1[0,1] = 0.8×0.3 + 1.3×0.4 + (-0.2)×0.2 + 1.5×0.1 = 0.24+0.52-0.04+0.15 = 0.87


Q1[1,0] = 1.2×0.2 + 1.1×0.1 + 0.8×0.3 + 0.7×0.4 = 0.24+0.11+0.24+0.28 = 0.87

Q1[1,1] = 1.2×0.3 + 1.1×0.4 + 0.8×0.2 + 0.7×0.1 = 0.36+0.44+0.16+0.07 = 1.03


Q1 = [[0.83, 0.87],    # query for "I"
      [0.87, 1.03]]    # query for "love"
```

Similarly for K1 and V1:

```
K1 = [[0.73, 1.31],    # key for "I"
      [0.93, 1.15]]    # key for "love"

V1 = [[1.37, 0.79],    # value for "I"
      [1.19, 0.95]]    # value for "love"
```

With Q1, K1, and V1 in hand for both positions, we can compute how strongly each position wants to attend to every other position. That is the job of the attention score matrix.

Step 3: Attention Scores = Q1 @ $K1^T$ / √d_k

Each query vector is dotted with every key vector, then divided by √2 ≈ 1.414 to stabilize the **softmax** (as we learned in Chapter 5 (Single-Head Attention), Section 5.3). This produces a 2×2 attention score matrix, how much each position wants to attend to every other position.

Step 4 **Attention Scores (Head 1)**

```
Score[0,0] = (0.83×0.73 + 0.87×1.31) / 1.414 = (0.606 + 1.140) / 1.414 =
1.235
Score[0,1] = (0.83×0.93 + 0.87×1.15) / 1.414 = (0.772 + 1.001) / 1.414 =
1.254


Score[1,0] = (0.87×0.73 + 1.03×1.31) / 1.414 = (0.635 + 1.349) / 1.414 =
1.403
Score[1,1] = (0.87×0.93 + 1.03×1.15) / 1.414 = (0.809 + 1.185) / 1.414 =
1.410


Scores = [[1.235, 1.254],
          [1.403, 1.410]]
```

Raw scores still include one illegal interaction, position 0 peeking at position 1, which would leak future information. The next step zeros that entry out with the causal mask, then turns the surviving scores into a proper probability distribution via softmax.

Step 4: Apply Causal Mask then Softmax

Position 0 can only attend to itself (Score[0,1] is masked to $-\infty$). Position 1 can attend to both positions. This is the same causal mask from Chapter 5 (Single-Head Attention, Section 5.4), preventing tokens from peeking at future tokens during the prefill phase.

Step 5 **Masked Scores and Softmax Weights**

```
Masked = [[1.235,  -∞  ],    # position 0 cannot see position 1
          [1.403, 1.410]]    # position 1 can see 0 and 1

# Row 0: only one valid score → weight = 1.0 for position 0
Attn[0] = softmax([1.235, -∞]) = [1.000, 0.000]

# Row 1: softmax([1.403, 1.410])
exp(1.403) = 4.067,  exp(1.410) = 4.096,  sum = 8.163
Attn[1] = [4.067/8.163, 4.096/8.163] = [0.498, 0.502]

Attention Weights = [[1.000, 0.000],    # "I" attends only to itself
                     [0.498, 0.502]]    # "love" attends ~equally to both
```

The softmax gave us the weights; now we multiply those weights against the value vectors to collapse the whole row of attention into a single output per position. This is where the head actually produces its contextual representation.

Step 5: Output = Attention Weights @ V1

Each row of attention weights is a weighted sum over the value vectors, producing Head 1's output.

Step 6 **Attention Output (Head 1)**

```
Out1[0] = 1.000×[1.37,0.79] + 0.000×[1.19,0.95] = [1.370, 0.790]


Out1[1] = 0.498×[1.37,0.79] + 0.502×[1.19,0.95]
        = [0.682,0.393]     + [0.597,0.477]
        = [1.279, 0.870]


Output_Head1 = [[1.370, 0.790],
                [1.279, 0.870]]
```

Head 1's attention output is now written to memory, but there is one more bookkeeping step before we move on to Head 2. We need to save the K1 and V1 matrices so the decode phase can retrieve them without recomputation.

Store KV Cache for Head 1

The computed K1 and V1 matrices are stored in the KV cache so they can be reused during decode steps without recomputation. This is the moment the KV cache earns its keep, these six numbers (K1 has four values, V1 has four values) will never be computed again for this generation request.

Step 7 **KV Cache After Prefill (Head 1)**

```
KV_cache[head=1] = {
    K: [[0.73, 1.31],    # position 0 "I"
        [0.93, 1.15]],   # position 1 "love"
    V: [[1.37, 0.79],
        [1.19, 0.95]]
}
```

Head 1 is complete and cached. Head 2 follows the same mechanical path with its own weight matrices, and once both heads are concatenated and passed through the rest of the block, we finally reach the next-token prediction. The following card collapses those remaining steps end to end so you can see how the prefill phase produces its first generated token.

Head 2 and Final Output

Head 2 follows the identical computational path but with its own independently learned projection matrices W_Q2, W_K2, and W_V2. Each head captures different aspects of the input, Head 1 might

focus on syntactic relationships while Head 2 captures semantic similarity. For brevity, we show the results directly:

Step 8 Head 2 Results and Final Prediction

```
Q2: [[0.81, 0.93], [1.02, 0.89]]
K2: [[0.95, 1.28], [1.08, 1.03]]
V2: [[1.28, 0.92], [1.15, 1.07]]
Attn2: [[1.0, 0.0], [0.503, 0.497]]
Output_Head2: [[1.280, 0.920], [1.215, 0.994]]
```

Concatenate Heads and Project Through W_O (4×4):

```
Concat[0] = [1.370, 0.790, 1.280, 0.920]  ← heads joined
Concat[1] = [1.279, 0.870, 1.215, 0.994]

MHA_output[0] = Concat[0] @ W_O = [0.939, 1.082, 0.873, 1.011]
MHA_output[1] = Concat[1] @ W_O = [0.921, 1.054, 0.856, 0.989]
```

After Residual + LayerNorm + Feed-Forward Network (FFN) (see Chapter 6 (Multi-Head Attention, Residuals & Layer Norm) and Chapter 7 (Feed-Forward Network, LM Head & The Complete Transformer) for full detail):

```
Layer_output[0] = [0.856, 1.123, 0.945, 1.034]
Layer_output[1] = [0.834, 1.089, 0.921, 1.008]  ← use this for the LM head
```

LM Head → Vocabulary Probabilities (position 1 only):

```
Logits = Layer_output[1] @ W_LM = [0.933, 1.002, 0.919, 1.272, 0.933]
Probs  = softmax(Logits) = [0.183, 0.196, 0.181, 0.257, 0.183]
                           PAD    I      love   AI     EOS

Predicted next token: "AI" (25.7% probability)
```

> *During prefill, we compute logits for every position, but we only use the last position's logits to generate the next token. Why? Because each position's logits predict the token that comes after that position. Position 0's logits predict what comes after "I" (which is "love", already known). Position 1's logits predict what comes after "I love", that is the token we actually need to generate. In general, only the last prompt token's logits are used for generation. The logits at other positions are computed as a side effect of the parallel forward pass but are discarded.*

Step 9 Color-Coded Summary: Prefill Inputs to Outputs

Reading guide: blue rows are inputs to the layer, amber rows are intermediate computations the layer produces along the way, and green rows are the outputs that get used downstream (passed to the next block, written to cache, or fed into the LM head).

`X[0] ("I"):`	0.8	1.3	-0.2	1.5
`X[1] ("love"):`	1.2	1.1	0.8	0.7

`Q1[0]:`	0.83	0.87
`Q1[1]:`	0.87	1.03
`K1[0]:`	0.73	1.31
`K1[1]:`	0.93	1.15
`V1[0]:`	1.37	0.79

V1[1]:	1.19	0.95
Scores row 0:	1.235	1.254
Scores row 1:	1.403	1.410
Attn weights row 0:	1.000	0.000
Attn weights row 1:	0.498	0.502

Out1[0] (head 1):	1.370	0.790
Out1[1] (head 1):	1.279	0.870

Layer_output[1]:	0.834	1.089	0.921	1.008

Probs over vocab:	0.183	0.196	0.181	0.257	0.183

The maximum probability (0.257) lands on vocab index 3 ("AI"), which becomes the first generated token.

The prefill phase is now complete. The KV cache holds K and V vectors for both prompt positions across both heads, and the model has selected "AI" as its first generated token. Now we enter the decode phase, where the KV cache will earn its keep by allowing us to process the new token "AI" without recomputing anything for "I" or "love."

10.4 Worked Example: The Decode Phase, KV Cache in Action

The prefill phase is done. We processed both prompt tokens ("I" and "love") in a single parallel forward pass, stored their K and V vectors in the cache, and selected "AI" as the first generated token. Now comes the decode phase, where the KV cache earns its keep. Instead of reprocessing the entire growing sequence from scratch, we will compute Q, K, and V for just the one new token and retrieve everything else from memory. This section walks through two full decode steps with the same concrete numbers from the prefill section. Watch how the cache grows by exactly one K vector and one V vector per step, while the computational cost stays constant regardless of how long the sequence has become.

The difference from prefill is immediately visible in the scale of work: instead of computing Q, K, and V projections for two tokens (six matrix-vector products per head), we compute them for exactly one (three matrix-vector products per head). Every other K and V value, for "I" at position 0 and "love" at position 1, comes straight from the cache. Zero recomputation. The savings are proportional to the number of cached positions, and as we will see in the speed-up table at the end, those savings get staggering at realistic sequence lengths.

Step 1: Embed the New Token "AI"

Token embedding plus positional encoding produces the input vector at position 2. Note that we are computing a single vector, not a matrix, this is a 1×4 vector, not a 2×4 matrix like during prefill.

Step 1 **Embed "AI" at Position 2**

```
X[2] = E[3] + PE[2] = [0.5, -0.4, 0.6, 0.8] + [0.9, -0.4, 0.2, 1.0] = [1.4, -0.8, 0.8, 1.8]
```

With the new token embedded, we now need its Q, K, and V, but only for position 2. Every earlier position is already in the cache, so we avoid the six matrix-vector products that the naive approach would require.

Step 2: Compute Q, K, V for Position 2 Only

Only 3 matrix-vector products instead of the 6 that would be needed if we recomputed everything. This is the KV cache savings made concrete: one token, three multiplications, done.

Step 2 **Q, K, V for Position 2 (Head 1)**

```
# (3 matrix-vector products, vs 6 in the naive approach)
```