GPT Explained

A hands-on guide to transformer architecture

Ju Lin

Table of contents

Welcome 6

License 7

Preface 8

- Who This Book is For? . . . 8
- How to Use This Book . . . 8
- What the book covers . . . 9
- Should I Buy This Book? . . . 9

Contributors 10

I Foundations 11

1 Introduction 13

- 1.1 What Is GPT? . . . 13
- 1.2 How GPT Works . . . 14
- 1.3 A Brief History of GPT . . . 16

2 Notation and Definitions 19

- 2.1 Scalars . . . 19
- 2.2 Capital Sigma and Capital Pi . . . 20
- 2.3 Vectors . . . 21
- 2.4 Dot Product . . . 26
- 2.5 Matrix Multiplication . . . 29
- 2.6 Softmax . . . 32
- 2.7 Logarithm and Exponential . . . 34
- 2.8 Mean and Variance . . . 35
- 2.9 Derivative . . . 38
- 2.10 Key Takeaways . . . 40

II Representations 42

3 Tokens — Text to Numbers 44
3.1 The Idea 44
3.2 Byte Pair Encoding (BPE) 45
3.3 The Matrix: A Tiny Vocabulary Table 45
3.4 The Code: A Micro-BPE in Python 47
3.5 Real World Use 51
3.6 Key Takeaways 52

4 Embeddings — Numbers to Meaning 54
4.1 The Idea 54
4.2 Why Learned Vectors? 54
4.3 The Math 55
4.4 The Matrix: Worked Example 56
4.5 The Code: Embedding in Python 57
4.6 Key Takeaways 58

5 Positional Encoding — Giving Order to Meaning 61
5.1 The Idea 61
5.2 Sinusoidal Positional Encoding 62
5.3 The Math 62
5.4 The Matrix: Worked Example 63
5.5 The Code: Positional Encoding in Python 64
5.6 Learned vs Sinusoidal Positional Embeddings 67
5.7 Key Takeaways 67

III Transformer Core 68

6 Attention — Tokens Talking to Each Other 70
6.1 The Idea 70
6.2 The Math 70
6.3 The Matrix: Worked Example 72
6.4 The Code: Scaled Dot-Product Attention in Python 75
6.5 Key Takeaways 76

7 RoPE: Position Inside Attention 78
7.1 The Idea 78
7.2 The Math 79
7.3 A Small Example 79
7.4 The Code Structure 80
7.5 Takeaways 82

8 Multi-Head Attention — Many Conversations at Once **83**
8.1 The Idea . . . 83
8.2 The Math . . . 84
8.3 The Matrix: Worked Example . . . 85
8.4 The Code: Multi-Head Attention in Python . . . 86
8.5 Why Multi-Head Attention Works . . . 87
8.6 Key Takeaways . . . 88

9 Feed-Forward Network — The Model's Memory **89**
9.1 The Idea . . . 89
9.2 The Math . . . 90
9.3 GELU Deep Dive . . . 90
9.4 The Matrix: Worked Example . . . 91
9.5 The Code: FFN in Python . . . 92
9.6 The FFN as a Key-Value Memory . . . 94
9.7 Key Takeaways . . . 95

10 The Transformer Block — Putting It Together **96**
10.1 The Idea . . . 96
10.2 The Math . . . 97
10.3 The Residual Stream . . . 97
10.4 The Matrix: Worked Example . . . 98
10.5 The Code: Transformer Block in Python . . . 99
10.6 Why N Blocks? . . . 102
10.7 Key Takeaways . . . 102

IV Prediction and Learning **103**

11 Vocabulary Projection — From Vectors to Words **105**
11.1 The Idea . . . 105
11.2 The Math . . . 105
11.3 The Generation Loop . . . 106
11.4 The Matrix: Worked Example . . . 107
11.5 The Code: Full microGPT Forward Pass in Python . . . 109
11.6 Weight Tying in Detail . . . 111
11.7 Key Takeaways . . . 111
11.8 The Complete Picture . . . 112

12 Loss — How the Model Learns **113**
12.1 The Idea . . . 113
12.2 The Problem: Evaluating a Probability Distribution . . . 113
12.3 Cross-Entropy Loss . . . 114

12.4 The Matrix: Worked Example . . . 116
12.5 Python Implementation . . . 117
12.6 Key Takeaways . . . 119

13 Training — Teaching the Model **120**
13.1 The Idea . . . 120
13.2 The Goal: Move the Loss Down . . . 120
13.3 The Chain Rule . . . 121
13.4 Backprop Through Softmax and Cross-Entropy . . . 122
13.5 Backprop Through a Linear Layer . . . 122
13.6 Accumulating Gradients Across the Sequence . . . 123
13.7 The Gradient Descent Step . . . 123
13.8 One Training Step . . . 124
13.9 Watching the Loss Fall . . . 124
13.10 Key Takeaways . . . 125

V Modern Extensions **126**

14 Modern GPT **128**
14.1 KV Cache . . . 128
14.2 Multi-Query and Grouped-Query Attention . . . 129
14.3 Flash Attention . . . 129
14.4 Alternative Architectures . . . 130
14.5 Key Takeaways . . . 131

Appendices **132**

A microGPT in Python — Complete Runnable Code **132**
A.1 Files . . . 132
A.2 Installation & Running . . . 132
A.3 End-to-End Demo . . . 132
A.4 microGPT Architecture Summary (Reference) . . . 133

Welcome

A hands-on guide to transformer architecture, from tokens to text generation, with math, matrices, Python, and diagrams.

This is the website for *GPT Explained*, an open book about how GPT-style language models actually work — from the inside out.

This book will teach you how transformers are built and trained. Not at a high level — at the level where you can read the code, follow the math, and trace a single token all the way through a real model.

You will learn tokenization, embeddings, positional encoding, attention, RoPE, multi-head attention, feed-forward networks, transformer blocks, vocabulary projection, cross-entropy loss, and backpropagation. Each concept arrives as a focused chapter: a plain-language explanation, the math in full, a Python implementation you can run, and diagrams generated from code.

In this book, you will build a small but complete GPT from scratch. Not a toy metaphor — actual matrix multiplications, actual softmax, actual multi-head attention, actual gradient descent. By the final chapter you will have a working miniature language model you can inspect line by line, and a clear mental model of every step a modern LLM takes between reading your prompt and writing its reply. The only prerequisites are basic Python and enough math to follow an equation when it is explained in words next to it. No machine-learning background is assumed.

GPT Explained is free to read online under the Creative Commons Attribution-NonCommercial-ShareAlike 3.0 license. You may share and adapt the material for non-commercial purposes as long as you give credit and keep the same license. Ebook and PDF versions are available for download on the releases page.

Contributions are welcome — whether that is a typo fix, a clearer explanation, a better diagram, or a new section. Open an issue or a pull request on GitHub. Please read the Code of Conduct before contributing; we want this project to be a welcoming place for everyone learning how these models work.

License

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License. To view a copy of this license, visit https://creativecommons.org/licenses/by-nc-sa/3.0/.

Preface

Who This Book is For?

This book is for programmers who want to understand how GPT-style language models actually work — not just use them.

If you have written Python before and can follow basic algebra, you have enough background to read every page. You do not need prior experience with machine learning, deep learning, or neural networks. The book introduces every concept from scratch, explains the math in plain language before writing it as a formula, and implements each idea in code you can run and inspect.

If you are already a machine-learning practitioner, this book gives you a clean, self-contained reference for the transformer architecture. The implementations are deliberately simple — no frameworks, no abstractions beyond plain Python — so nothing hides what is actually happening.

How to Use This Book

Read the chapters in order on the first pass. Each chapter builds on the previous one: tokens before embeddings, embeddings before attention, attention before the full transformer block. Skipping ahead is possible but you may find yourself returning to fill gaps.

Every code snippet is part of a complete, runnable program. The source lives in `src/python/` and can be run with `python3 src/python/run_book_code.py`. Every diagram is generated from code in `src/matplotlib/`, `src/figures/`, or directly inside the chapter files. If a diagram surprises you, open its source and change the numbers.

Work through the exercises at the end of each chapter. The exercises are not optional decoration — they are the fastest way to verify that you understood the material rather than just read it.

What the book covers

The book covers fourteen chapters and one appendix:

Foundations. Chapter 1 gives an overview of GPT and its history. Chapter 2 introduces the math notation used throughout: vectors, matrices, dot products, softmax, logarithms, mean, and variance.

Representations. Chapter 3 explains tokenization and byte-pair encoding. Chapter 4 covers word embeddings and the embedding lookup. Chapter 5 introduces sinusoidal positional encoding.

Transformer Core. Chapter 6 derives scaled dot-product attention step by step. Chapter 7 shows how modern GPT-style models put position directly inside attention. Chapter 8 extends it to multi-head attention. Chapter 9 covers the feed-forward sub-layer. Chapter 10 assembles these pieces into a full transformer block with residual connections and layer normalization.

Prediction and Learning. Chapter 11 covers vocabulary projection. Chapter 12 derives cross-entropy loss and perplexity. Chapter 13 traces backpropagation and the Adam optimizer through the full model.

Modern GPT. Chapter 14 describes the changes in current models: RoPE positional encoding, grouped-query attention, SwiGLU activations, and RMSNorm.

Appendix A presents the complete micro-GPT implementation — the same code from across the chapters assembled into one readable file.

Should I Buy This Book?

The full text is free to read online at the book's website. Nothing is paywalled.

If you find the book useful and want to support the work, you can purchase the EPUB or PDF edition from the releases page. Buying a copy is a way of saying thank you — but it is entirely optional. The content is identical to what you are reading now.

Contributors

Contributors as of 71a43b5:

- Ju Lin

Part I

Foundations

This part establishes the vocabulary used throughout the rest of the book. It starts with the high-level idea of GPT-style models, then introduces the compact math notation needed to read the diagrams and Python code.

- In Chapter 1, you will see what a GPT model does and how the main pieces fit together.
- In Chapter 2, you will learn the vector, matrix, probability, and calculus notation used in the later chapters.

1 Introduction

This chapter answers two questions before you touch any math or code: what is GPT, and how does it work? We keep the language concrete and the diagrams simple. The details come later.

1.1 What Is GPT?

GPT stands for *Generative Pre-trained Transformer*.

Generative — it produces new text, one token at a time, rather than classifying or retrieving.

Pre-trained — before you ever use it, the model was trained on a vast corpus of text from the internet, books, and code. It learned the statistical patterns of language from billions of examples.

Transformer — the underlying neural network architecture. The transformer replaced recurrent networks in 2017 and became the backbone of every large language model since.

A GPT model is a function:

$$P(\text{next token} \mid \text{context tokens}) = f_\theta(\text{context})$$

Given a sequence of tokens (integers representing text), it outputs a probability distribution over the vocabulary — every possible next token gets a score. The model then samples from this distribution to generate the next word. Repeat, and you get fluent text.

> **Note**
>
> *What is a token?* A token is the smallest unit of text the model processes. It is not always a whole word — "unbelievable" might be two tokens: "unbel" and "ievable". Chapter 3 explains exactly how text is split into tokens.

1.2 How GPT Works

Imagine texting with a really smart friend who has read every book, article, and website ever written. You send a message, they think for a split second, and they reply with exactly the right words. GPT works like that friend, except instead of a brain, it uses a chain of math operations called a *pipeline*.

Every word GPT produces goes through these seven steps, in order:

Step 1 — Tokenization (Chapter 3). Before GPT can think about your text, it has to chop it into small pieces called *tokens*. Think of tokens like LEGO bricks: you break a sentence apart, and GPT works with the individual bricks. The phrase `"the cat"` might become three bricks with serial numbers like `[1, 428, 3797]`.

Step 2 — Embedding (Chapter 4). A number like `428` is just an ID — it has no meaning by itself. So GPT looks each ID up in a giant table and swaps it for a list of hundreds of numbers called a *vector*. Picture a map where every word has its own location. Similar words end up close together: "cat" and "kitten" are neighbors, while "cat" and "spaceship" are far apart.

Step 3 — Positional Encoding (Chapter 5). Here is a puzzle: does "dog bites man" mean the same thing as "man bites dog"? Obviously not — but if GPT only sees individual words and ignores their order, it cannot tell the difference. Positional encoding adds a tiny "position tag" to each word's vector so GPT knows which word came first, second, and so on.

Step 4 — Transformer Blocks (Chapter 6 through Chapter 10). This is where the real thinking happens. GPT passes the vectors through a stack of identical layers called *transformer blocks*. Inside each block, two things happen back-to-back:

- *Attention* — every word looks at every other word and asks "which of you matters most to me right now?" For example, the word "it" in "I kicked the ball and it bounced" needs to figure out that "it" refers to "ball".
- *Feed-forward* — each word's vector gets processed through a small math circuit that stores facts from training.

There can be dozens of these blocks stacked on top of each other. Each layer builds a slightly deeper understanding of the text.

Step 5 — Vocab Projection (Chapter 11). After passing through all the blocks, GPT has a rich, updated vector for the last word in your prompt. Now it needs to turn that into a real prediction: "what word should come next?" It does this by scoring every word in its vocabulary — all 50,000-plus of them. These raw scores are called *logits*.

Step 6 — Softmax (Chapter 11). Raw scores are hard to work with, so GPT converts them into proper probabilities that add up to 100%. The highest-probability word is the model's best guess. It samples from the top options to avoid always giving the same robotic answer.

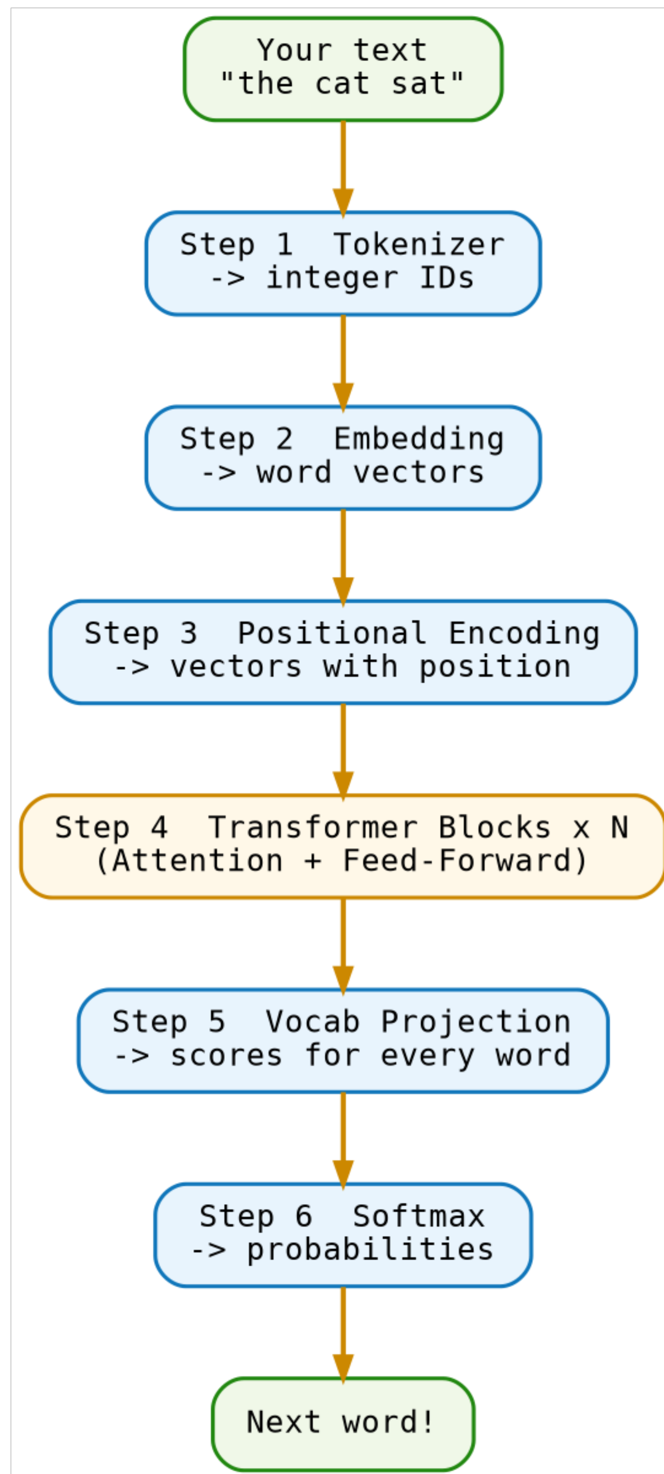

Figure 1.1: The GPT pipeline: from input text to predicted next word, in seven steps

Step 7 — Loss and Training (Chapter 12 and Chapter 13). GPT did not start out smart. During training, it read billions of sentences and kept trying to predict the next word — getting it wrong most of the time at first. Each mistake triggered a feedback signal called *backpropagation* that nudged millions of internal numbers a tiny bit in the right direction. After trillions of these small nudges, the model got very good at the guessing game.

That's GPT from start to finish. The rest of this book builds each step from scratch — in math, in Python code, and in step-by-step diagrams.

1.3 A Brief History of GPT

GPT did not appear out of nowhere. It is the result of more than sixty years of research: a long chain of ideas, failures, and breakthroughs. Here is the story, told simply.

1950s–1980s — The Birth of Machine Learning

Early computers followed strict rules written by programmers. If you wanted a computer to recognise a cat, you had to describe a cat in code: "four legs, pointy ears, fur…" That worked for simple things but fell apart for anything complicated.

Researchers started asking a different question: what if the computer could *learn the rules itself* by looking at lots of examples? That idea became *machine learning*. Instead of coding rules by hand, you feed the machine thousands of cat photos, label them "cat", and let it figure out the pattern.

1943–1986 — Neural Networks

Inspired by the human brain, researchers built *neural networks* — programs made of many simple math nodes connected together, loosely like neurons. Each node takes some numbers in, does a small calculation, and passes the result on.

In 1986, a key paper by Geoffrey Hinton and others showed how to train these networks using an algorithm called *backpropagation*. Think of it like a teacher who marks your test, finds every mistake, and tells you exactly how much to adjust each answer. This was huge — but computers at the time were too slow for it to become widely useful yet.

2012 — The Deep Learning Explosion

Fast-forward to 2012. A neural network called AlexNet entered an image-recognition contest and demolished the competition. It won by a margin so wide that the whole field changed overnight. The secret ingredient was *deep learning* — stacking many layers of neurons on top of each other — combined with powerful graphics cards (GPUs) that could run the math fast enough.

Suddenly every company poured money into neural networks.

2013 — Word Vectors (Word2Vec)

A team at Google figured out how to teach a neural network to read text. They trained it to predict which words tend to appear near each other. The surprising result: the network assigned each word a list of numbers (a *vector*), and similar words ended up with similar numbers. You could even do word arithmetic: “king” − “man” + “woman” “queen”. Language had become math.

2014–2015 — Sequence Models and Attention

To translate a sentence — say, from English to French — models needed to process words in order. Researchers used networks called *RNNs* (Recurrent Neural Networks) and *LSTMs* for this. But they had a problem: long sentences broke them. By the time the model reached the end of a paragraph, it had mostly forgotten the beginning.

In 2015, researchers introduced *attention*. Instead of forcing the model to squeeze everything into one memory slot, attention let it look back at any earlier word at any time, like being allowed to flip back to any page of your notes during an exam. This made translation much better.

2017 — “Attention Is All You Need”

A team at Google published a paper with that bold title. They asked: what if we got rid of the old word-by-word processing entirely and used *only* attention? The result was the *transformer* — the architecture that GPT is built on. The transformer could look at all the words in a sentence at the same time, in parallel, which made it dramatically faster to train and better at long-range reasoning.

2018 — GPT-1 and BERT

Two important models appeared within months of each other.

OpenAI released *GPT-1*, which trained a transformer to do one simple task: read text and predict the next word. After seeing enough text, the model picked up grammar, facts, and reasoning as a side effect. It had 117 million *parameters* — the tunable numbers inside the network (imagine dials on a mixing board).

Google released *BERT*, which trained a transformer to predict missing words in the middle of a sentence instead of just the next one. BERT became the backbone of Google Search for years.

2019 — GPT-2

OpenAI scaled GPT-1 up to 1.5 billion parameters and fed it text scraped from the internet. The result was striking: GPT-2 could write paragraphs that sounded like a real person wrote them. OpenAI were nervous enough about misuse that they released it in stages rather than all at once.

2020 — GPT-3

Another scale jump: 175 billion parameters, trained on a huge chunk of the internet, books, and code. GPT-3 could write essays, answer questions, translate, summarise, and generate code — without being specifically trained for any of those things. It had just absorbed so much human writing that it learned to generalise. Developers started building products on top of it via an API, and a new industry was born.

2023 — LLaMA and the Open-Source Wave

Until now, the most powerful models were locked behind the walls of big companies. Meta changed that by releasing *LLaMA* — a family of models that could run on a single powerful laptop. Researchers and hobbyists could finally experiment with a state-of-the-art model without needing a data centre. Hundreds of variants appeared within weeks: Alpaca, Vicuna, Mistral, and many more. The open-source AI ecosystem exploded.

2022–2023 — ChatGPT and GPT-4

OpenAI added a training step called *reinforcement learning from human feedback* (RLHF). Human raters scored the model's replies, and the model learned to be more helpful and less likely to say harmful things. The result was *ChatGPT*, released in November 2022 — the fastest app to reach 100 million users in history.

GPT-4 followed in 2023. It understood images as well as text, scored at or above human level on many standardised exams, and became the foundation for countless products and services.

Now — A Fast-Moving Field

Since then, new models have appeared every few months: Claude (Anthropic), Gemini (Google), Mistral, Qwen, DeepSeek, and many others. Each brings new ideas: longer context windows, better reasoning, cheaper inference, multilingual support.

The race is still on. But no matter how fancy the model, the fundamentals stay the same. Everything you learn in this book — tokens, embeddings, attention, transformer blocks, training — is the engine underneath all of them.

2 Notation and Definitions

This chapter is a compact reference for the core math used at the start of the book. It focuses on notation, definitions, and small numerical examples.

2.1 Scalars

A *scalar* is a single number. Examples include -2, 0, 0.5, and 3.14.

Scalars can be added, subtracted, multiplied, divided, squared, and compared. They are the simplest objects in this chapter. Vectors and matrices are built by arranging scalars into larger shapes.

> **Note**
>
> *Math Minute – Variable*
> A variable is a name for a value. In $x = 3.0$, the variable is x and its value is 3.0. In formulas, variables let us describe a whole pattern without writing a separate equation for every possible number.

2.1.1 Scalar Operations

For two scalars $a = 3.0$ and $b = 2.0$:

$$a + b = 5.0$$

$$ab = 6.0$$

$$a^2 = 9.0$$

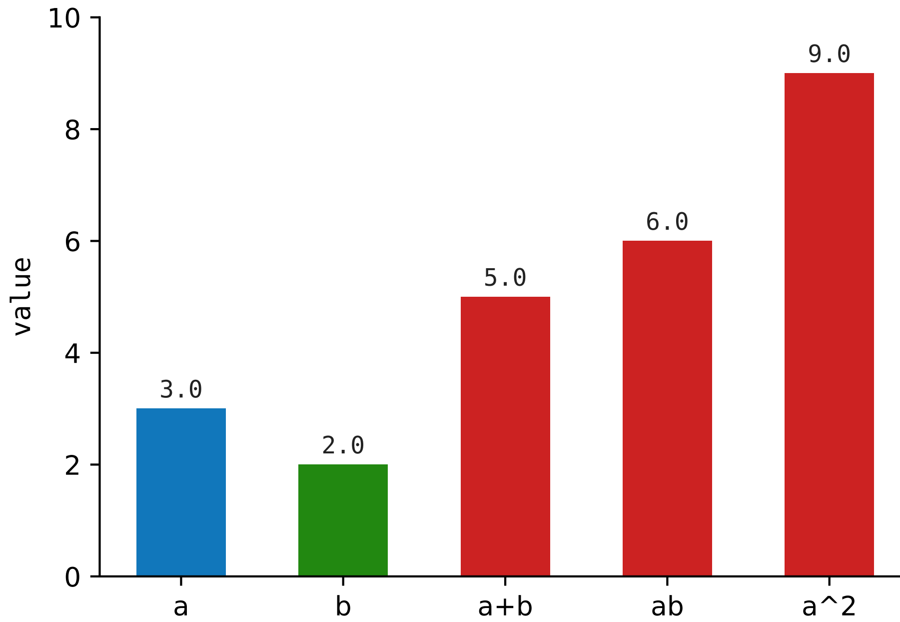

Figure 2.1: Scalar operations combine individual numbers

2.2 Capital Sigma and Capital Pi

Capital sigma $\sum$ means "add a sequence of terms." Capital pi $\prod$ means "multiply a sequence of terms."

2.2.1 Capital Sigma: Sum

The expression:

$$\sum_{i=1}^{4} x_i$$

means:

$$x_1 + x_2 + x_3 + x_4$$

For $x = [2, 3, 5, 7]$:

$$\sum_{i=1}^{4} x_i = 2 + 3 + 5 + 7 = 17$$

> **Note**
>
> *Math Minute – Subscript*
> A subscript is the small label written below and to the right of a symbol. In x_i, the subscript i selects one entry from the sequence x. So x_1 means "the first entry of x."

2.2.2 Capital Pi: Product

The expression:

$$\prod_{i=1}^{4} x_i$$

means:

$$x_1 x_2 x_3 x_4$$

For $x = [2, 3, 5, 7]$:

$$\prod_{i=1}^{4} x_i = 2 \cdot 3 \cdot 5 \cdot 7 = 210$$

2.3 Vectors

A *vector* $v \in \mathbb{R}^n$ is an ordered list of n real numbers: $v = [v_1, v_2, \dots, v_n]$. Vectors store many related measurements as one object. Once values are written as vectors, we can add, scale, compare, and transform them with ordinary arithmetic.

> **Note**
>
> *Math Minute — Element Of*
> The symbol $\in$ means "is an element of" or, more casually, "is in." So $v \in \mathbb{R}^n$ says that v is one item from the set $\mathbb{R}^n$. In the above definition, v is a vector with n real-number components.

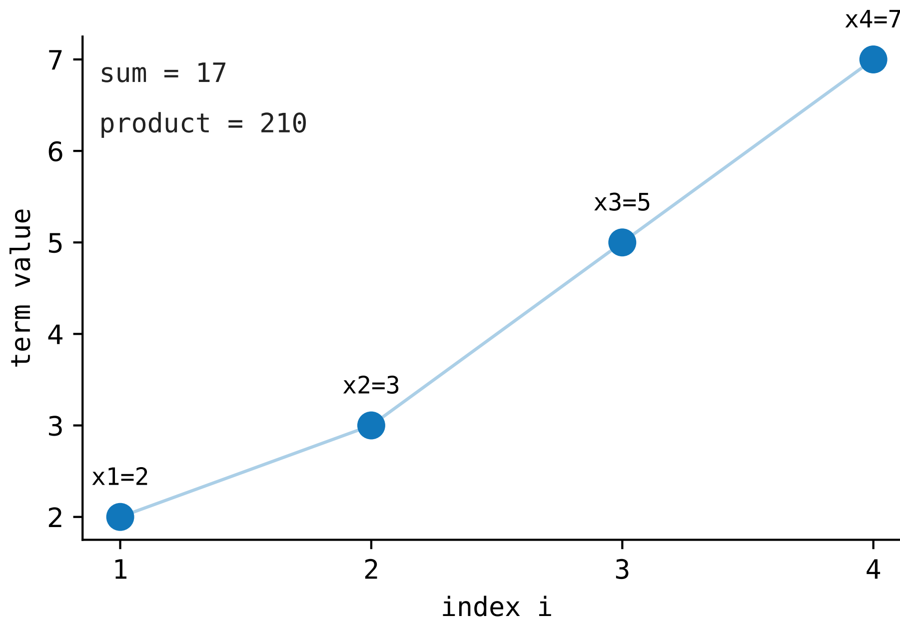

Figure 2.2: Capital sigma adds terms; capital pi multiplies terms

> **Note**
>
> *Math Minute — Real Number*
> A real number is any ordinary number on the number line: -2, 0, 0.5, $\sqrt{2}$, π, and so on. The notation $\mathbb{R}$ means "the set of all real numbers." So $\mathbb{R}^n$ means "all vectors with n real-number components."

In Python, the book represents a vector as a list of floating-point numbers:

```
Vector = list[float]
```

This simple representation is enough for the small implementations in the book. Packages like NumPy provide their own vector type, usually as an `ndarray`, with faster arithmetic, compact memory layout, and many built-in operations. For larger numerical programs, you would usually use NumPy, PyTorch, JAX, or another tensor library instead.

2.3.1 Vector Addition

When two vectors have the same length, we can add them by adding matching positions. The first component adds to the first component, the second to the second, and so on:

$$u + v = [u_1 + v_1, u_2 + v_2, ...]$$

The Python version follows that sentence directly: walk through both vectors together and add each pair.

```
def add_vectors(left: Vector, right: Vector) -> Vector:
    return [a + b for a, b in zip(left, right)]
```

For example, take $u = [3.0, 4.0]$ and $v = [1.0, -2.0]$. The first components give $3.0 + 1.0 = 4.0$, and the second components give $4.0 + (-2.0) = 2.0$. So:

$$u + v = [3.0 + 1.0, 4.0 + (-2.0)] = [4.0, 2.0]$$

Figure 2.3 shows the same addition geometrically. Dashed connectors show how u and v meet at $u + v$.

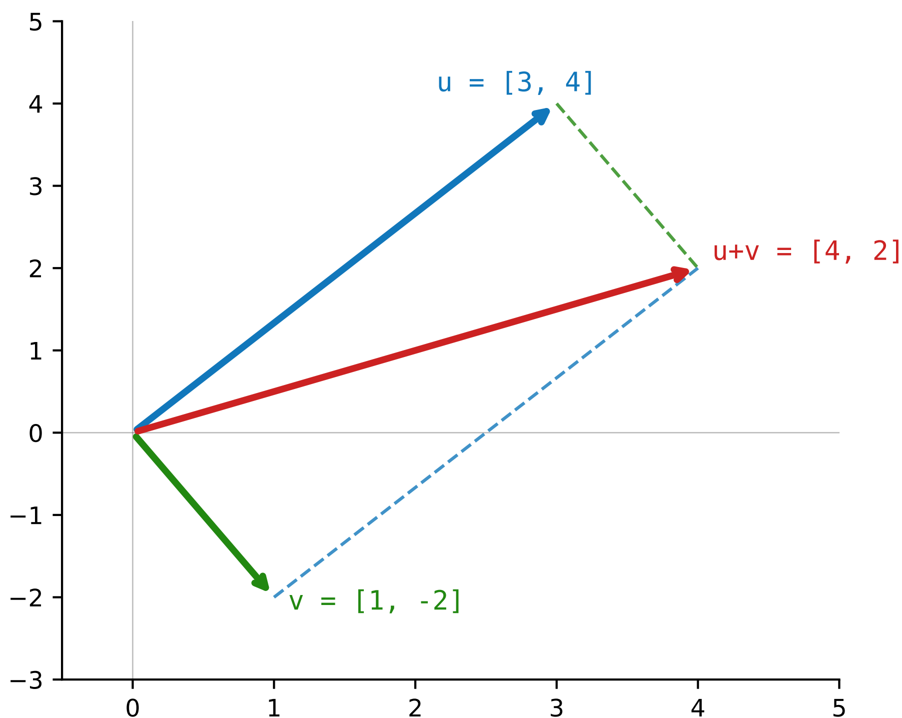

Figure 2.3: Vector addition in 2D

2.3.2 Scalar Multiplication

A scalar is a single number. Multiplying a vector by a scalar stretches or shrinks the whole vector without changing how many components it has:

$$c \cdot v = [cv_1, cv_2, ...]$$

In code, this means multiplying every value in the list by the same scalar.

```
def scale_vector(vector: Vector, scalar: float) -> Vector:
    return [scalar * value for value in vector]
```

For $u = [3.0, 4.0]$, multiplying by 2.0 doubles both components:

$$2.0 \cdot u = [6.0, 8.0]$$

2.3.3 L2 Norm

The L2 norm is the length of a vector. For a two-dimensional vector, this is the Pythagorean theorem; for longer vectors, the same idea extends across all components:

$$\|v\| = \sqrt{v_1^2 + v_2^2 + ... + v_n^2}$$

The implementation squares every component, adds the squares, then takes the square root.

```
def vector_norm(vector: Sequence[float]) -> float:
    return math.sqrt(sum(value * value for value in vector))
```

For $u = [3.0, 4.0]$, the length is:

$$\|u\| = \sqrt{3.0^2 + 4.0^2} = \sqrt{9.0 + 16.0} = 5.0$$

Figure 2.4 shows the vector $[3, 4]$ as a right triangle with length 5.

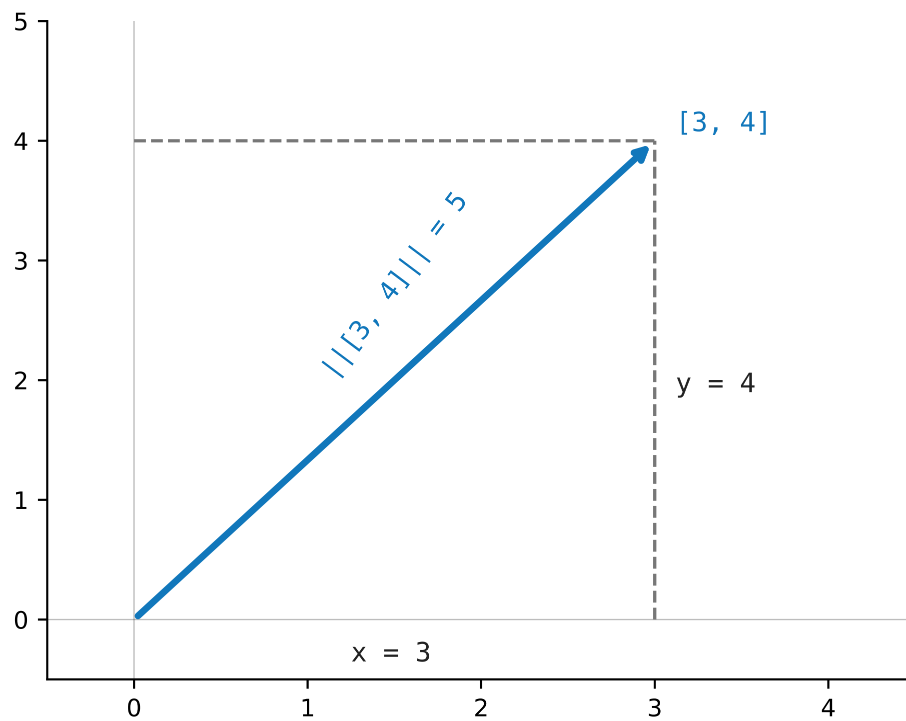

Figure 2.4: The vector [3, 4] has L2 norm 5

2.3.4 Unit Vector

A unit vector keeps the direction of the original vector but scales its length to 1. To get one, divide the vector by its own norm:

$$\hat{v} = v/\|v\|$$

The helper below combines the two earlier operations: compute the norm, then scale by its reciprocal.

```
def unit_vector(vector: Vector) -> Vector:
    return scale_vector(vector, 1.0 / vector_norm(vector))
```

For $u = [3.0, 4.0]$, the norm is 5.0, so each component is divided by 5.0:

$$[3.0/5.0, 4.0/5.0] = [0.6, 0.8]$$

2.4 Dot Product

The *dot product* (also called inner product) combines two vectors into one scalar. It is the basic operation for asking, "how aligned are these two vectors?"

2.4.1 Dot Product

The dot product multiplies matching components and then adds the results:

$$u \cdot v = \sum_{i=1}^{n} u_i v_i = u_1 v_1 + u_2 v_2 + \ldots + u_n v_n$$

That is also the most direct way to write it in Python. The `zip` pairs matching components, and `sum` adds the pairwise products.

```
def dot(left: Sequence[float], right: Sequence[float]) -> float:
    return sum(a * b for a, b in zip(left, right))
```

With $u = [3.0, 4.0, 0.0]$ and $v = [1.0, -2.0, 5.0]$, the matching products are $3.0 \cdot 1.0$, $4.0 \cdot (-2.0)$, and $0.0 \cdot 5.0$. Adding them gives:

$$u \cdot v = (3.0)(1.0) + (4.0)(-2.0) + (0.0)(5.0) = 3.0 - 8.0 + 0.0 = -5.0$$

The sign matters. A positive dot product means the vectors point somewhat in the same direction. A negative dot product means they point somewhat against each other. If $u \cdot v = 0$, the vectors are orthogonal (perpendicular).

2.4.2 Cosine Similarity

The dot product mixes two things: vector length and direction. Cosine similarity removes the length part so we can compare direction alone. Starting from the geometric form:

$$u \cdot v = \|u\|\|v\| \cos\theta$$

we solve for $\cos\theta$:

$$\cos\theta = \frac{u \cdot v}{\|u\|\|v\|}$$

The implementation is just that formula: dot product divided by the two vector lengths.

```
def cosine_similarity(left: Sequence[float], right: Sequence[float]) -> float:
    return dot(left, right) / (vector_norm(left) * vector_norm(right))
```

Using the same u and v as above, $\|u\| = 5.0$ and $\|v\| = \sqrt{30.0}$. Since the dot product is -5.0, the cosine similarity is:

$$\frac{u \cdot v}{\|u\|\|v\|} = \frac{-5.0}{5.0\sqrt{30.0}} \approx -0.183$$

The value is below zero, so the vectors point in moderately opposite directions.

Figure 2.5 shows the dot product as a scalar projection in 2D.

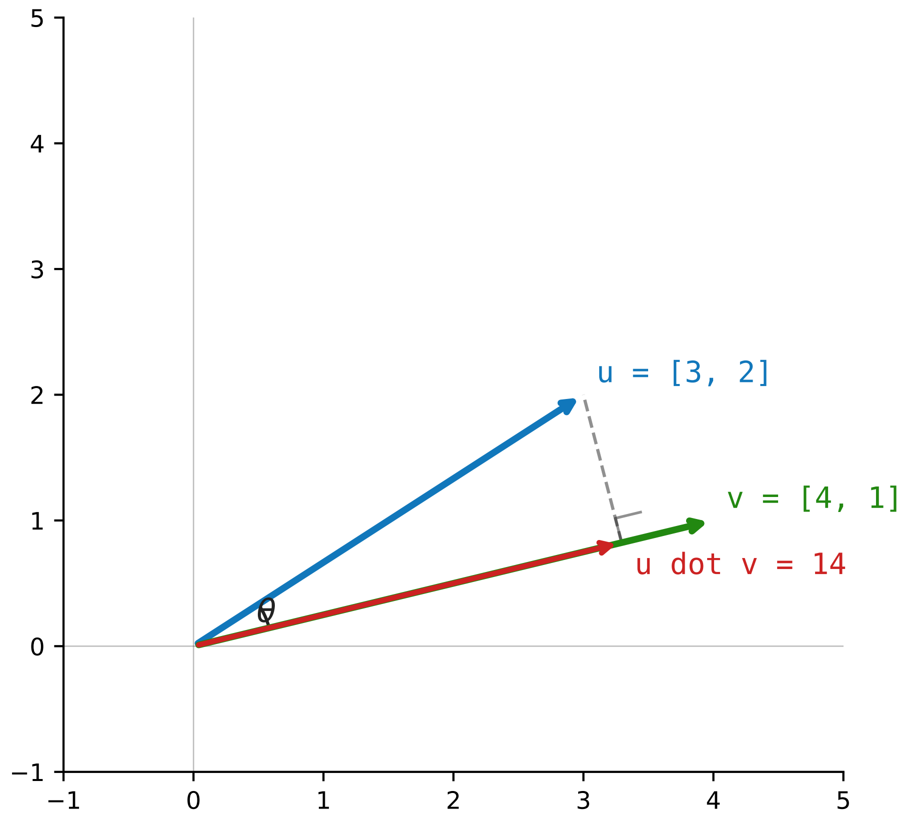

Figure 2.5: Dot product as scalar projection in 2D

2.5 Matrix Multiplication

A matrix is a rectangular grid of numbers. Matrices make it possible to organize many vectors or many linear equations at once. Matrix multiplication combines rows from one matrix with columns from another. The book represents a matrix as a list of rows, where each row is a list of floats:

```
Matrix = list[list[float]]
```

> **Note**
>
> *Math Minute – Matrix Shape and Indexing*
> A matrix with m rows and n columns has shape $m \times n$. The notation $A \in \mathbb{R}^{m \times n}$ means every entry of A is a real number, and A has m rows and n columns. The notation A_{ij} means "the entry in row i, column j of matrix A."

For example, this matrix:

$$A = \begin{bmatrix} 1.0 & 2.0 & 3.0 \\ 4.0 & 5.0 & 6.0 \end{bmatrix}$$

has shape 2×3: two rows and three columns. In code, the same matrix is represented as a list of rows:

```
A = [
    [1.0, 2.0, 3.0],
    [4.0, 5.0, 6.0],
]
```

2.5.1 Transpose

Transposing a matrix swaps rows and columns. The first row becomes the first column, the second row becomes the second column, and so on. If $A \in \mathbb{R}^{m \times n}$, then $A^T \in \mathbb{R}^{n \times m}$.

In Python, `zip(*matrix)` groups together the first item from every row, then the second item from every row, and so on. Converting each group back to a list gives the transposed rows.

```
def transpose(matrix: Matrix) -> Matrix:
    return [list(col) for col in zip(*matrix)]
```

For:

$$A = \begin{bmatrix} 1.0 & 2.0 \\ 3.0 & 4.0 \end{bmatrix}$$

the transpose is:

$$A^T = \begin{bmatrix} 1.0 & 3.0 \\ 2.0 & 4.0 \end{bmatrix}$$

2.5.2 Matrix Multiplication

Matrix multiplication combines rows from the left matrix with columns from the right matrix. Given $A \in \mathbb{R}^{m \times k}$ and $B \in \mathbb{R}^{k \times n}$, their product $C = AB$ has shape $m \times n$, and each entry is:

$$C_{ij} = \sum_{r=1}^{k} A_{ir} B_{rj}$$

That equation says: take row i from A, take column j from B, and compute their dot product. The implementation transposes the right matrix first so its columns become easy to iterate over as rows.

```
def matrix_multiply(left: Matrix, right: Matrix) -> Matrix:
    right_t = transpose(right)
    return [[dot(row, col) for col in right_t] for row in left]
```

For:

$$A = \begin{bmatrix} 1.0 & 2.0 \\ 3.0 & 4.0 \end{bmatrix}$$

and:

$$B = \begin{bmatrix} 5.0 & 6.0 \\ 7.0 & 8.0 \end{bmatrix}$$

the four output cells are:

$$C_{00} = 1.0 \cdot 5.0 + 2.0 \cdot 7.0 = 19.0$$

$$C_{01} = 1.0 \cdot 6.0 + 2.0 \cdot 8.0 = 22.0$$

$$C_{10} = 3.0 \cdot 5.0 + 4.0 \cdot 7.0 = 43.0$$

$$C_{11} = 3.0 \cdot 6.0 + 4.0 \cdot 8.0 = 50.0$$

So:

$$AB = \begin{bmatrix} 19.0 & 22.0 \\ 43.0 & 50.0 \end{bmatrix}$$

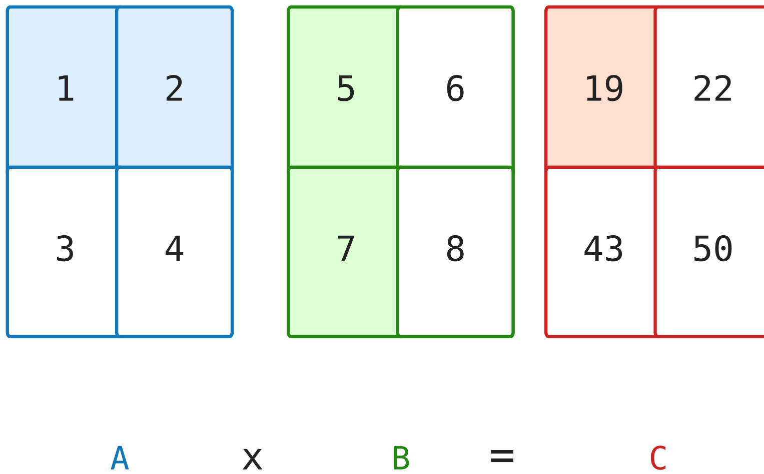

Figure 2.6: Matrix multiplication: each output cell is a dot product of a row with a column

Figure 2.6 illustrates how each output cell is a dot product of a row with a column.

2.6 Softmax

The *softmax* function turns arbitrary real numbers into probabilities. The input numbers are called logits. The output values are all positive and sum to 1.

2.6.1 Vector Softmax

Softmax exponentiates each logit and divides by the total exponentiated mass:

$$\text{softmax}(z)_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

In code, we subtract the maximum logit before exponentiating. This does not change the final probabilities, because the same constant shift appears in the numerator and denominator. It does prevent very large exponentials from overflowing.

$$\text{softmax}(z)_i = \frac{e^{z_i - \max(z)}}{\sum_j e^{z_j - \max(z)}}$$

```
def softmax(logits: Sequence[float]) -> Vector:
    max_logit = max(logits)
    exp_values = [math.exp(value - max_logit) for value in logits]
    total = sum(exp_values)
    return [value / total for value in exp_values]
```

For $z = [2.0, 1.0, 0.1]$, the maximum is 2.0, so the shifted values are $[0.0, -1.0, -1.9]$. After exponentiating, we get approximately $[1.000, 0.368, 0.150]$, whose sum is 1.518. Dividing each value by that sum gives:

$$\text{softmax}(z) \approx [0.659, 0.242, 0.099]$$

The largest logit gets the largest probability, but the smaller logits still receive nonzero probability.

2.6.2 Row-wise Softmax

Sometimes the input is a matrix of scores. Each row can be normalized into its own probability distribution. Row-wise softmax applies the vector softmax independently to every row.

```
def softmax_rows(matrix: Matrix) -> Matrix:
    return [softmax(row) for row in matrix]
```

After this step, each row sums to 1.

Figure 2.7 shows how softmax maps raw logits to a probability distribution.

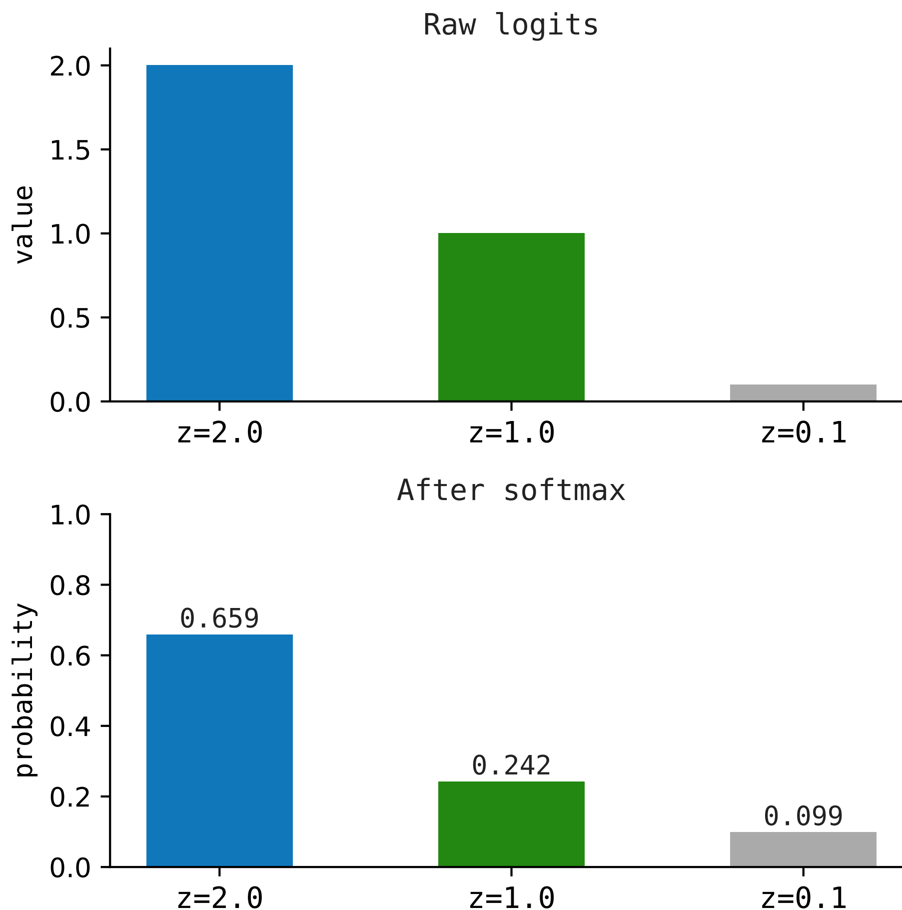

Figure 2.7: Softmax turns logits into probabilities — higher values dominate, none reach zero

2.7 Logarithm and Exponential

The natural exponential e^x and natural logarithm $\ln x$ are inverse operations:

$$e^{\ln x} = x \quad \text{and} \quad \ln(e^x) = x$$

These functions convert between additive and multiplicative scales. They are useful whenever growth, decay, ratios, or repeated multiplication need to be written on an additive scale.

2.7.1 Natural Exponential

The natural exponential raises e to a power:

$$\exp(x) = e^x$$

In Python, this is `math.exp`.

```
def natural_exp(value: float) -> float:
    return math.exp(value)
```

Two anchor values appear often enough to keep in mind. If the exponent is 0.0, the result is 1.0. If the exponent is 1.0, the result is e:

$$e^{0.0} = 1.0$$

$$e^{1.0} \approx 2.718$$

2.7.2 Natural Logarithm

The natural logarithm asks the reverse question: what power of e gives this value?

$$\ln x = y \quad \text{means} \quad e^y = x$$

In Python, this is `math.log`.

```
def natural_log(value: float) -> float:
    return math.log(value)
```

Some useful values:

$$\ln(1.0) = 0.0$$

$$\ln(e) = 1.0$$

$$\ln(0.5) \approx -0.693$$

$$\ln(0.1) \approx -2.303$$

For inputs between 0 and 1, the natural logarithm is negative. Negating it gives a positive value that grows as the input moves closer to zero, as shown in Figure 2.8. For example:

$$-\ln(0.01) = 4.605$$

Two identities appear often when simplifying logarithms:

$$\ln(ab) = \ln a + \ln b$$

$$\ln(a^n) = n \ln a$$

2.8 Mean and Variance

Mean and variance summarize a vector of numbers. They tell us where the values are centered and how spread out they are.

2.8.1 Mean

The mean is the average value. Add all values, then divide by how many values there are:

$$\mu = \frac{1}{n} \sum_{i=1}^{n} x_i$$

The implementation is just that calculation.

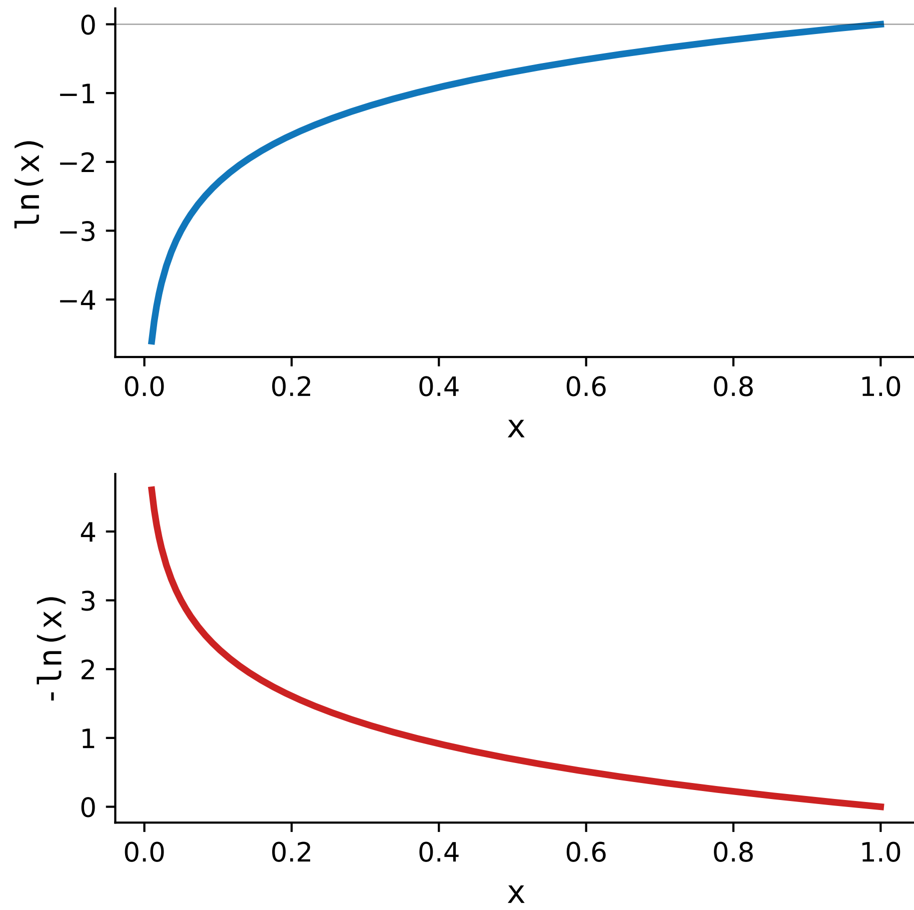

Figure 2.8: The natural log and its negative over inputs between 0 and 1

```python
def mean(values: Sequence[float]) -> float:
    return sum(values) / len(values)
```

For $x = [2.0, 4.0, 4.0, 4.0, 5.0, 5.0, 7.0, 9.0]$, the sum is 40.0 and there are 8 values. So:

$$\mu = \frac{40.0}{8.0} = 5.0$$

2.8.2 Variance

Variance measures how spread out the values are. First subtract the mean from each value. Then square those deviations so negative and positive deviations do not cancel out. Finally, average the squared deviations:

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2$$

The Python helper computes the mean first, then follows the formula.

```python
def variance(values: Sequence[float]) -> float:
    avg = mean(values)
    return sum((value - avg) ** 2 for value in values) / len(values)
```

Continuing with $x = [2.0, 4.0, 4.0, 4.0, 5.0, 5.0, 7.0, 9.0]$, the mean is 5.0. The squared deviations are:

$$9.0, 1.0, 1.0, 1.0, 0.0, 0.0, 4.0, 16.0$$

Their average is:

$$\sigma^2 = \frac{9.0 + 1.0 + 1.0 + 1.0 + 0.0 + 0.0 + 4.0 + 16.0}{8.0} = 4.0$$

2.8.3 Standard Deviation

Standard deviation puts variance back on the original scale by taking the square root:

$$\sigma = \sqrt{\sigma^2}$$

```
def standard_deviation(values: Sequence[float]) -> float:
    return math.sqrt(variance(values))
```

In this example, the variance is 4.0, so the standard deviation is:

$$\sigma = \sqrt{4.0} = 2.0$$

Figure 2.9 illustrates the example data with mean and standard deviation marked.

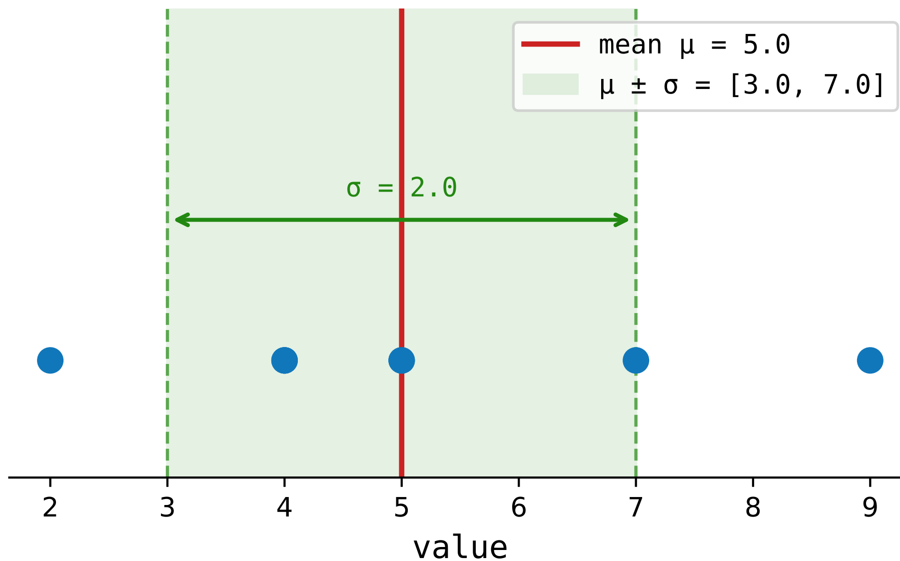

Figure 2.9: Data points with mean and standard deviation

2.9 Derivative

A *derivative* measures how fast one value changes as another value changes. For a function $f(x)$, the derivative is written:

$$\frac{df}{dx}$$

This reads as "the derivative of f with respect to x." It tells us the slope of the function at a particular input.

Note

Math Minute – Function
A function maps an input to an output. If $f(x) = x^2$, then $f(3) = 9$. The input is x and the output is $f(x)$.

Note

Math Minute – Slope
Slope means "change in vertical value divided by change in horizontal value." Between two points, slope is:

$$\frac{\Delta y}{\Delta x}$$

A derivative is the slope when the two points get infinitely close together.

2.9.1 Example

For:

$$f(x) = x^2$$

the derivative is:

$$\frac{df}{dx} = 2x$$

At $x = 3$, the slope is:

$$2x = 2 \cdot 3 = 6$$

Some useful derivative facts:

|=== | Operation | Mathematical form | Derivative

`a + b` | $a + b$ | $\frac{\partial}{\partial a} = 1, \quad \frac{\partial}{\partial b} = 1$
`a * b` | $a \cdot b$ | $\frac{\partial}{\partial a} = b, \quad \frac{\partial}{\partial b} = a$
`a ** n` | a^n | $\frac{\partial}{\partial a} = n \cdot a^{n-1}$
`log(a)` | $\ln(a)$ | $\frac{\partial}{\partial a} = \frac{1}{a}$
`exp(a)` | e^a | $\frac{\partial}{\partial a} = e^a$
`relu(a)` | $\max(0, a)$ | $\frac{\partial}{\partial a} = \mathbf{1}_{a>0}$

|===

For ReLU, the derivative at exactly $a = 0$ is undefined; implementations usually choose a fixed convention.

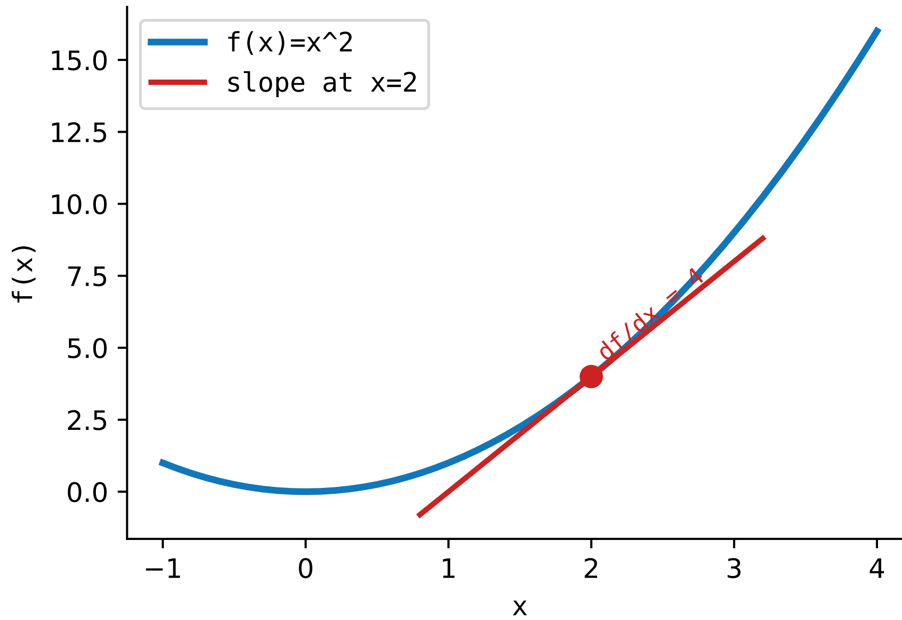

Figure 2.10: A derivative is the local slope of a function

2.10 Key Takeaways

This chapter defined a compact mathematical toolkit.

Concept	Main idea
Scalar	A single number
Capital Sigma/Pi	Compact notation for repeated addition or multiplication
Vector	An ordered list of numbers
Dot product	A scalar measure of alignment between two vectors
Matrix multiply	Row-by-column dot products arranged into a new matrix
Softmax	A way to convert real-valued scores into probabilities

Concept	Main idea
Logarithm	The inverse of exponentiation
Mean/Variance	Measurements of center and spread
Derivative	The local rate of change of a function

Together, these tools cover the notation and arithmetic used throughout the examples in this chapter.

Part II

Representations

This part turns raw text into vectors the model can compute with. The goal is to make each row of the model's input matrix carry both identity and position.

- In Chapter 3, you will see how text becomes token IDs, and how a small byte-pair encoding tokenizer works.
- In Chapter 4, token IDs become learned vectors through an embedding matrix.
- In Chapter 5, those vectors gain order so the model can distinguish "dog bites man" from "man bites dog."

3 Tokens — Text to Numbers

A language model does not read text. It reads *integers*. The first job of any GPT pipeline is to convert a string of characters into a sequence of integers. That conversion is called *tokenization*, and the integers it produces are called *token IDs*.

This chapter explains what tokens are, how they are produced, and what they look like as data structures. We implement a tiny byte-pair encoding tokenizer in Python.

3.1 The Idea

Consider the sentence:

```
the cat likes tokenization
```

A human reads four words. A GPT model reads something like:

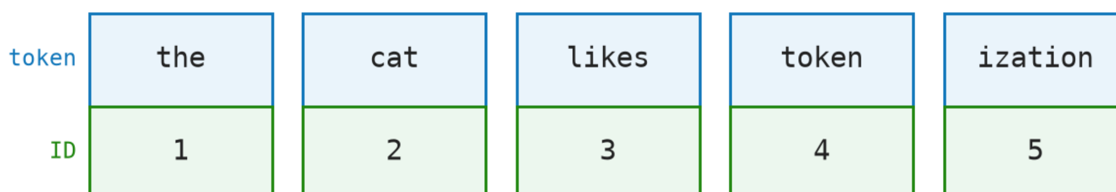

Figure 3.1: Each token has a text piece and a vocabulary ID

Where each two-cell block is a token and the number is its ID in a vocabulary table. The model never sees the full word `"tokenization"` as one unit here — it sees the IDs for `"token"` and `"ization"`.

Why not just use characters? You could map each letter to an integer: `a→1, b→2, …`. The problem is that the model would need to learn that `c`, `a`, `t` together mean something different from `c`, `a`, `r`. It would have to rediscover every word from scratch. This makes learning extremely sample-inefficient.

Why not use whole words? The opposite extreme. Map every word to an integer. This works for common words, but English has hundreds of thousands of words, and new ones appear constantly. The vocabulary would be enormous, and rare words would have almost no training data.

Subword tokenization finds the middle ground. Common whole words become single tokens. Rare words get split into recognizable pieces. The word `"tokenization"` might become `["token", "ization"]`. The word `"running"` might become `["run", "ning"]`.

3.2 Byte Pair Encoding (BPE)

The most common tokenization algorithm used in GPT models is *Byte Pair Encoding*. The algorithm has two phases:

Training phase (happens once, offline):

1. Start with a vocabulary of individual characters (or bytes).
2. Count every adjacent pair of tokens in the training corpus.
3. Merge the most frequent pair into a single new token.
4. Repeat until the vocabulary reaches the target size (e.g., 50,000).

Encoding phase (happens at inference time):

1. Start with the input string split into individual characters.
2. Greedily merge adjacent pairs according to the trained merge rules, highest-priority first.
3. Return the final token IDs.

3.2.1 The Math

Let $V_0 = \{c_1, c_2, ...\}$ be the initial character vocabulary. After each merge step k:

$$V_{k+1} = V_k \cup \{(a, b) \mid \text{freq}(a, b) \text{ is max over all adjacent pairs}\}$$

The merge order defines a priority list $M = [(a_1, b_1), (a_2, b_2), ...]$.

3.3 The Matrix: A Tiny Vocabulary Table

This section traces through a micro-example. Suppose our entire training corpus is:

```
low lower newest
```

3.3.1 Step 0 — Character Vocabulary

Token	l	o	w	e	r	n	s	t	_
ID	0	1	2	3	4	5	6	7	8

_ marks the end of a word.

Initial tokenized corpus: `[l,o,w,_]` `[l,o,w,e,r,_]` `[n,e,w,e,s,t,_]`

3.3.2 Step 1 — Choose a Frequent Pair to Merge

Count adjacent token pairs across the corpus:

Token pair	(l,o)	(o,w)	(w,_)	(w,e)
Count Picked?	2	2	1	1

Here `(l,o)` and `(o,w)` are tied. This trace chooses `(l,o)` first, merges it into `lo`, and gives `lo` the next ID: 9.

Token	l	o	w	e	r	n	s	t	_	lo
ID	0	1	2	3	4	5	6	7	8	9

3.3.3 Step 2 — Apply That Merge to the Corpus

Every `l,o` pair becomes one `lo` token:

`[lo,w,_]` `[lo,w,e,r,_]` `[n,e,w,e,s,t,_]`

Count adjacent token pairs again:

Token pair	(lo,w)	(w,_)	(w,e)	(e,r)
Count Picked?	2	1	1	1

The frequent pair is (lo,w), so the next merge creates low with ID 10.

Token	l	o	w	e	r	n	s	t	_	lo	low
ID	0	1	2	3	4	5	6	7	8	9	10

After a few more merges, encoding "low" returns [10] — a single token.

3.4 The Code: A Micro-BPE in Python

3.4.1 The Vocabulary Abstraction

```
def make_vocab() -> dict[str, int]:
    return {}
```

make_vocab starts with an empty token-to-ID mapping.

```
def vocab_extend(vocab: dict[str, int], token: str, token_id: int) -> dict[str, int]:
    return vocab | {token: token_id}
```

vocab_extend returns a vocabulary with one more token ID.

```
def vocab_lookup(vocab: dict[str, int], token: str) -> int | None:
    return vocab.get(token)
```

vocab_lookup retrieves the ID for a token, or None if the token is unknown. Together, these helpers create, extend, and query the mapping from string tokens to integer IDs.

3.4.2 Characters and Corpus

```
def word_to_tokens(word: str) -> list[str]:
    return list(word) + ["_"]
```

word_to_tokens turns one word into characters and appends "_" as the end-of-word marker.

```python
def words_to_corpus(words: Sequence[str]) -> list[list[str]]:
    return [word_to_tokens(word) for word in words]
```

`words_to_corpus` applies that conversion to every word in the training corpus.

3.4.3 Counting Pairs

```python
from collections import Counter
from typing import Sequence
```

This section uses `Counter` to tally pair frequencies and `Sequence` for read-only sequence type hints.

```python
def adjacent_pairs(tokens: Sequence[str]) -> list[tuple[str, str]]:
    return list(zip(tokens, tokens[1:]))
```

`adjacent_pairs` lists every neighboring token pair in one token sequence.

```python
def count_pairs(corpus: Sequence[Sequence[str]]) -> Counter[tuple[str, str]]:
    return Counter(pair for tokens in corpus for pair in adjacent_pairs(tokens))
```

`count_pairs` counts those neighboring pairs across the whole corpus.

```python
def most_frequent(counts: Counter[tuple[str, str]]) -> tuple[str, str]:
    return max(counts, key=counts.get)
```

`most_frequent` chooses the pair with the highest count.

3.4.4 Applying a Merge

```python
def merge_pair(tokens: Sequence[str], left: str, right: str) -> list[str]:
    merged = []
    i = 0
    while i < len(tokens):
        if i + 1 < len(tokens) and tokens[i] == left and tokens[i + 1] == right:
            merged.append(left + right)
```

```
            i += 2
        else:
            merged.append(tokens[i])
            i += 1
    return merged
```

`merge_pair` scans one token sequence and replaces every matching adjacent pair with the joined token.

```
def merge_corpus(corpus: Sequence[Sequence[str]], left: str, right: str) -> list[list[str]]:
    return [merge_pair(tokens, left, right) for tokens in corpus]
```

`merge_corpus` applies the same merge to every token sequence in the corpus.

3.4.5 The Training Loop

```
def unique_tokens(corpus: Sequence[Sequence[str]]) -> list[str]:
    return sorted({token for tokens in corpus for token in tokens})
```

`unique_tokens` collects the current vocabulary items from the tokenized corpus.

```
def train_bpe(words: Sequence[str], target_size: int) -> tuple[dict[str, int], list[tuple[st
    corpus = words_to_corpus(words)
    vocab = {token: token_id for token_id, token in enumerate(unique_tokens(corpus))}
    merges = []

    while len(vocab) < target_size:
        counts = count_pairs(corpus)
        if not counts:
            break
        left, right = most_frequent(counts)
        new_token = left + right
        if new_token not in vocab:
            vocab[new_token] = len(vocab)
        merges.append((left, right))
        corpus = merge_corpus(corpus, left, right)

    return vocab, merges
```

`train_bpe` repeatedly counts pairs, merges the most frequent pair, records the merge rule, and adds each new token to the vocabulary.

3.4.6 Encoding

```
def encode(word: str, merges: Sequence[tuple[str, str]], vocab: dict[str, int]) -> list[int]
    tokens = word_to_tokens(word)
    for left, right in merges:
        tokens = merge_pair(tokens, left, right)
    missing = [token for token in tokens if token not in vocab]
    if missing:
        raise ValueError(f"unknown token(s): {missing}")
    return [vocab[token] for token in tokens]
```

`encode` applies the trained merge rules to a new word and returns the final token IDs.

3.4.7 Demo

```
def demo() -> dict[str, object]:
    words = ["low", "lower", "newest", "widest"]
    vocab, merges = train_bpe(words, 20)
    return {
        "vocab": vocab,
        "merges": merges,
        "low": encode("low", merges, vocab),
        "lower": encode("lower", merges, vocab),
    }
```

`demo` trains a tiny tokenizer and encodes two example words. Run this file directly with `python3 src/python/micro_bpe.py`.

Output:

```
Vocabulary:
  0 -> _
  1 -> d
  2 -> e
  3 -> i
  4 -> l
  5 -> n
  6 -> o
  7 -> r
  8 -> s
```

```
  9 -> t
  10 -> w
  11 -> lo
  12 -> low
  13 -> es
  14 -> est
  15 -> est_
  16 -> low_
  17 -> lowe
  18 -> lower
  19 -> lower_

Encoding "low":   [16]
Encoding "lower": [19]
```

3.5 Real World Use

In real GPT applications, you usually do not train a tokenizer yourself. You use the tokenizer that matches the model. For OpenAI models, the common Python library for inspecting tokenization is `tiktoken`.

Install it with:

```
python3 -m pip install tiktoken
```

A simple use looks like this:

```
import tiktoken

encoding = tiktoken.get_encoding("cl100k_base")

text = "the cat likes tokenization"
token_ids = encoding.encode(text)
pieces = [encoding.decode([token_id]) for token_id in token_ids]

print(token_ids)
print(pieces)
```

Output:

```
[1820, 8415, 13452, 4037, 2065]
['the', ' cat', ' likes', ' token', 'ization']
```

`encode` turns text into token IDs. `decode` turns token IDs back into text. Decoding one ID at a time is a useful way to inspect what each token represents. Notice how the spaces are not separate end-of-word markers. They are folded into nearby tokens: `" cat"`, `" likes"`, and `" token"` each include the leading space.

For model-specific work, use the encoding for the model rather than choosing an encoding by hand:

```
import tiktoken

encoding = tiktoken.encoding_for_model("MODEL_NAME")
token_ids = encoding.encode("the cat likes tokenization")
```

Our micro-BPE implementation and `tiktoken` share the same broad idea: text becomes a sequence of token IDs through a fixed vocabulary and merge rules. The differences are important:

Our micro-BPE	`tiktoken`
Trains a tiny vocabulary from a few example words.	Ships with pretrained vocabularies and merge rules.
Starts from Python characters.	Works at the byte level, so it can handle arbitrary text robustly.
Uses `"_"` as an end-of-word marker in the teaching example.	Does not add an end-of-word token to every word; whitespace is encoded as part of the token stream.
Splits a list of words, not a full document.	Encodes normal text directly, including spaces and punctuation.
Raises an error if an unseen token remains after merging.	Is designed for production use and can encode real input text.

The main lesson carries over: the model receives integers, not raw strings. The production tokenizer just has a much larger vocabulary, carefully trained merge rules, and more robust handling of real text.

3.6 Key Takeaways

- Text enters the model as a *sequence of integers* (token IDs).

- Tokens are *subword units*, not characters or whole words.
- The *vocabulary* is the fixed lookup table mapping tokens to IDs.
- BPE builds the vocabulary by *greedily merging* the most frequent pairs.
- The tokenizer is trained *once*; at inference time it only applies the stored merge rules.

> **Note**
>
> We have a sequence of integers: `[10, 3, 4, 8, …]`. These integers are just row indices. To give them geometric meaning — so the model can do math on them — we need an *embedding matrix*. That is Chapter 4.

4 Embeddings — Numbers to Meaning

We have token IDs: `[10, 3, 4, 8]`. They are integers. The model cannot do useful math on raw integers — adding `10 + 3` gives `13`, which means nothing. To do useful math, each integer needs to become a *vector* — a list of numbers that encodes the token's meaning geometrically.

The mechanism is called *the embedding matrix*, and it is the model's first learned representation.

4.1 The Idea

To give each token a meaning the model can actually compute with, we replace its ID with a *list of hundreds of decimal numbers* — a vector. Think of it like this: the model keeps a big table, one row per word in the vocabulary. When it sees token `42`, it pulls out row `42` and uses those numbers as the token's representation going forward.

The key property that makes this useful is that the table is *learned*. After training on billions of words, tokens with similar meanings end up with similar rows. "Cat" and "kitten" cluster together. "Run" and "sprint" are nearby. "Cat" and "justice" are far apart. The model discovers these relationships entirely from how words appear together in text — no human labels required.

This learned table is the *embedding matrix*, and the step that converts a sequence of IDs into a sequence of vectors is called *embedding*.

4.2 Why Learned Vectors?

The values in the embedding matrix are not hand-crafted. They are initialized randomly and then *learned via gradient descent* during training, just like any other model parameter.

> **Note**
>
> *Math Minute — Gradient Descent*
> Gradient descent is the basic training rule: measure the model's error, compute which direction would increase that error, then nudge each parameter a small step in the opposite

direction. Repeated millions or billions of times, these tiny updates turn random numbers into useful weights. For embeddings, that means each token's row moves toward values that help the model predict text better.

The training objective is next-token prediction: given $t_1, t_2, \dots, t_n$, predict t_{n+1}. Because the model must do this well across billions of examples, it is forced to arrange embeddings such that tokens that appear in similar contexts land near each other in the embedding space.

This produces the famous arithmetic property:

$$\text{embedding}(\text{king}) - \text{embedding}(\text{man}) + \text{embedding}(\text{woman}) \approx \text{embedding}(\text{queen})$$

No one programmed this. It is a geometric consequence of how the vectors were shaped by the training signal.

4.3 The Math

The embedding matrix $E \in \mathbb{R}^{|V| \times d}$ holds one learned row per token — $|V|$ tokens wide, d_{model} dimensions deep. To embed token $i \in \{0, \dots, |V| - 1\}$, we pull out its row:

$$e_i = E_{i,\cdot} \in \mathbb{R}^d$$

For a sequence of T tokens $[i_1, \dots, i_T]$, we collect the corresponding rows into a matrix:

$$X = E[[i_1, \dots, i_T], :] \in \mathbb{R}^{T \times d}$$

Row indexing, no arithmetic. We can also write the same lookup as a matrix multiplication using a one-hot vector:

$$e_i = o_i^\top \cdot E, \quad \text{where } o_i \in \{0, 1\}^{|V|}, \quad (o_i)_j = \begin{cases} 1 & j = i \\ 0 & j \neq i \end{cases}$$

The one-hot form shows up in theoretical derivations, but implementations skip it and index directly — it is much faster.

> **Note**
>
> *Math Minute — One-Hot Vector*
> A one-hot vector has exactly one entry equal to 1 and all others 0. Token ID 2 in a vocabulary of size 5 becomes:

$$o_2 = [0, 0, 1, 0, 0]$$

Multiplying $o_2^\top$ by E selects row 2 — the same result as $E_{2,\cdot}$.

4.4 The Matrix: Worked Example

Take tiny numbers: $|V| = 5$, $d = 4$.

Embedding matrix E (5×4):

```
        col0   col1   col2   col3
tok 0: [ 0.10  -0.20   0.30  -0.40 ]
tok 1: [ 0.50   0.60  -0.70   0.80 ]
tok 2: [-0.90   0.10   0.20  -0.30 ]
tok 3: [ 0.40  -0.50   0.60  -0.70 ]
tok 4: [-0.10   0.80  -0.40   0.50 ]
```

Input token sequence: `"low lower"` → token IDs `[2, 3, 0]` (hypothetical).

Embedding lookup:

```
X = E[[2, 3, 0], :]

X[0] = E[2] = [-0.90,  0.10,  0.20, -0.30]   ← "low"
X[1] = E[3] = [ 0.40, -0.50,  0.60, -0.70]   ← "low" (part of "lower")
X[2] = E[0] = [ 0.10, -0.20,  0.30, -0.40]   ← "er"

Result X: shape [3 × 4]
```

This $[3 \times 4]$ matrix is what flows into the next stage.

Figure Figure 4.1 shows token IDs selecting row vectors from the embedding matrix.

Semantic similarity in vector space:

If we compute the dot product of two token embeddings, we get a scalar measuring how aligned they are:

```
dot(E[0], E[2]) = (0.10)(-0.90) + (-0.20)(0.10) + (0.30)(0.20) + (-0.40)(-0.30)
              = -0.09 - 0.02 + 0.06 + 0.12
              = 0.07   (slightly positive → slight similarity)
```

After training, semantically related tokens will have much higher dot products.

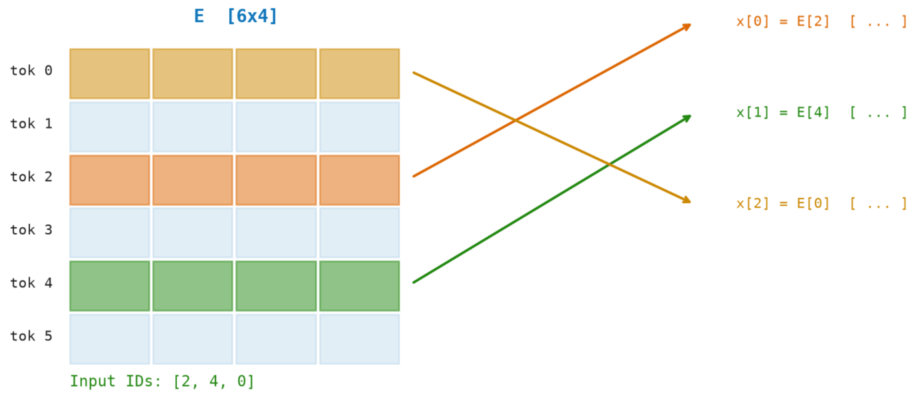

Figure 4.1: Token IDs mapped to embedding rows

4.5 The Code: Embedding in Python

Chapter 3 already showed how text becomes token IDs. Here we start from those IDs and look up vectors.

```
def make_matrix(rows: int, cols: int, fill: float = 0.0) -> Matrix:
    return [[fill for _ in range(cols)] for _ in range(rows)]


def shape(matrix: Matrix) -> tuple[int, int]:
    return len(matrix), len(matrix[0]) if matrix else 0
```

This helper creates a rows-by-cols grid of numbers. The embedding matrix uses the same matrix abstraction introduced earlier, so chapter 4 does not need a new data structure.

```
def random_matrix(
    rows: int,
    cols: int,
    rng: random.Random,
    scale: float | None = None,
) -> Matrix:
    scale = math.sqrt(2.0 / (rows + cols)) if scale is None else scale
    return [[rng.uniform(-scale, scale) for _ in range(cols)] for _ in range(rows)]
```

`random_matrix` fills that grid with small random values. With the default Xavier scale, the random values are sized by $\sqrt{2/(fan_{in} + fan_{out})}$ so activations are less likely to explode or vanish.

```
def embed_token(embedding: Matrix, token_id: int) -> Vector:
    return list(embedding[token_id])
```

The embedding of token i is row i of the matrix. This is a table lookup, not a calculation over the token ID itself.

```
def embed_sequence(embedding: Matrix, token_ids: Sequence[int]) -> Matrix:
    return [embed_token(embedding, token_id) for token_id in token_ids]
```

A sequence of T token IDs becomes T row lookups stacked into a $[T \times d]$ matrix. The vector helpers used to compare rows are the same ones from Chapter 2.

Demo. Run with `python3 src/python/chapter_demos.py`:

```
def chapter_04(seed: int = 4) -> dict[str, object]:
    rng = random.Random(seed)
    embedding = random_matrix(100, 8, rng)
    x = embed_sequence(embedding, [1, 2, 3])
    return {
        "sequence_shape": (len(x), len(x[0])),
        "dot": dot(embedding[1], embedding[2]),
        "cosine": cosine_similarity(embedding[1], embedding[2]),
    }
```

The demo starts with token IDs, creates an embedding table, and returns the embedded sequence shape. It also compares two embedding rows with dot product and cosine similarity, reusing the chapter 2 helpers instead of repeating them here.

Figure Figure 4.2 shows how each token ID indexes into one row of the embedding weight matrix.

Figure Figure 4.3 shows how semantic relationships can appear as vector directions in embedding space.

4.6 Key Takeaways

- An embedding matrix $E \in \mathbb{R}^{|V| \times d}$ maps token IDs to dense vectors.

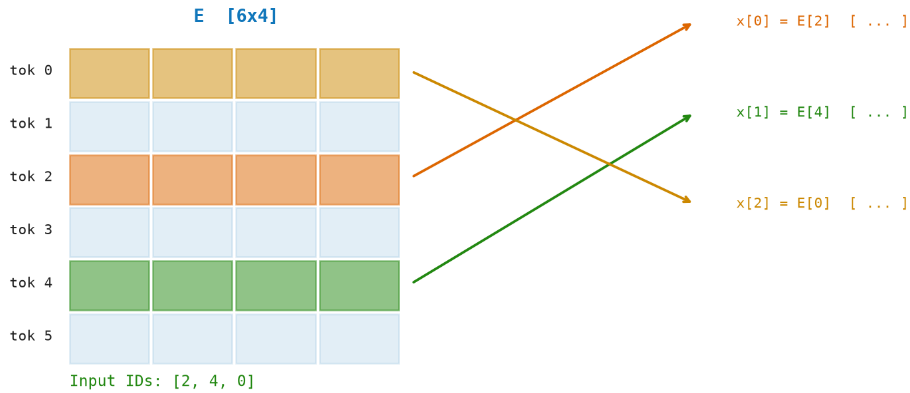

Figure 4.2: Embedding matrix lookup

- The embedding of token `i` is row `i` of `E` — a single *lookup*, no computation.
- For a sequence of `T` tokens, we get a $[T \times d]$ matrix `X`.
- The vectors are *learned*: training nudges similar tokens closer together.
- Semantic arithmetic ($\text{king} - \text{man} + \text{woman} \approx \text{queen}$) emerges from the training objective.

> **Note**
>
> *What's next?* We have a matrix `X` of shape $[T \times d]$. Each row knows *what* the token is, but nothing about *where* it sits in the sequence. Swapping two tokens would produce the same rows in a different order, which the model would treat identically. We need to inject *position information* — that is Chapter 5.

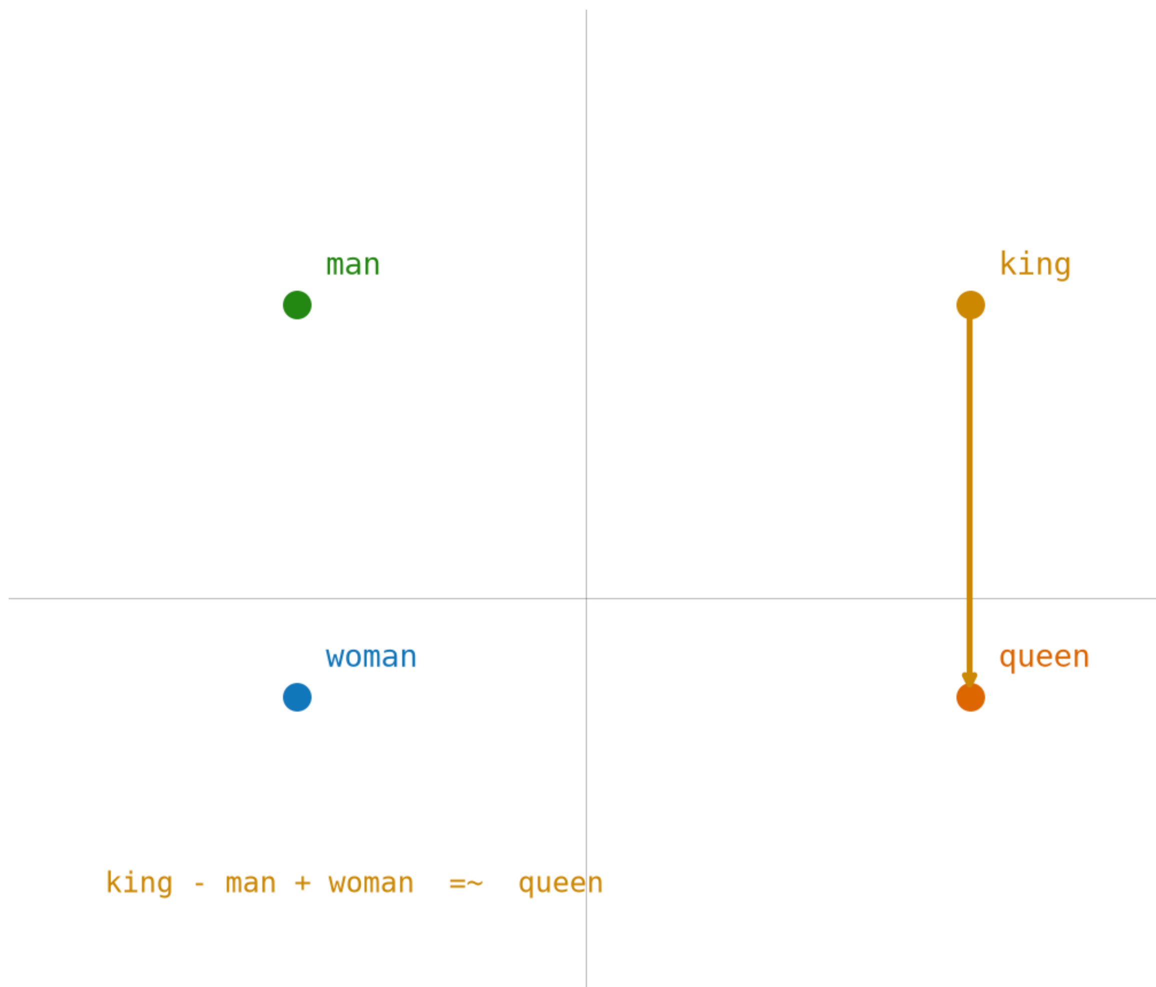

Figure 4.3: Semantic arithmetic in embedding space

5 Positional Encoding — Giving Order to Meaning

After the embedding step we have a matrix $X \in \mathbb{R}^{T \times d}$. Each row `X[t]` is a vector representing token `t`. But here is the problem: attention processes all tokens *simultaneously* — there is no inherent notion of "token 3 comes after token 2." If you shuffled the rows, the model would not know.

We fix this by *adding a position vector* to each token's embedding. The combined vector carries both *what* (the token identity) and *where* (the position).

5.1 The Idea

After embedding, every token is a vector. But those vectors carry no information about *where* in the sentence the token appears. "Dog bites man" and "man bites dog" produce the same three vectors, just in a different order — and attention would treat both sentences identically if we did nothing.

The fix is straightforward: before passing the vectors into attention, we *add a position signal* to each one. Token 0 gets a small nudge in one direction; token 1 gets a different nudge; token 50 gets yet another. The nudge is itself a vector the same size as the embedding, and it is designed so that no two positions produce the same nudge.

Think of it like stamping each page of a manuscript with its page number before sending it to the printer. The text on the page does not change — but now the order is recoverable.

There are two ways to design the position stamp:

- *Learned*: keep a second lookup table, one row per position, and let the model learn the best stamps from data. GPT-2 does this.
- *Sinusoidal*: compute the stamp from a fixed mathematical formula using waves of different frequencies. The original transformer paper did this. We study both, because the sinusoidal approach illuminates *why* position signals work.

5.2 Sinusoidal Positional Encoding

For position `t` and dimension `i` (where $0 \le i < d$):

$$PE(t,\, 2i) = \sin\!\left(\frac{t}{10000^{2i/d}}\right)$$
$$PE(t,\, 2i+1) = \cos\!\left(\frac{t}{10000^{2i/d}}\right)$$

Even-indexed dimensions use sine; odd-indexed dimensions use cosine. The denominator `10000^(2i/d)` controls the frequency — it grows exponentially with `i`, so low dimensions have high frequency (vary rapidly with `t`) and high dimensions have low frequency (vary slowly).

Why this formula works:

1. *Unique per position:* Every `t` produces a distinct vector.
2. *Bounded:* All values lie in `[-1, 1]`, so they do not dominate the token embeddings.
3. *Smooth:* Nearby positions have similar encodings, so the model can learn "close in position $\to$ related in meaning."
4. *Extrapolatable:* The formula works for any `t`, even positions not seen during training.

Relative position: The encoding for position `t+k` can be expressed as a linear transformation of the encoding for position `t`. The model can therefore attend to "the token 3 positions back" with a simple linear operation, without memorizing absolute positions.

5.3 The Math

Full formula for the positional encoding matrix $PE \in \mathbb{R}^{T\times d}$:

$$PE[t,i] = \sin(t \cdot \omega_{i/2}) \quad \text{if } i \text{ is even}$$
$$PE[t,i] = \cos(t \cdot \omega_{\lfloor i/2 \rfloor}) \quad \text{if } i \text{ is odd}$$

where $\omega_k = 1/10000^{2k/d}$.

After building `PE`, the position-encoded input is:

$$\tilde{X} = X + PE \quad \in \mathbb{R}^{T\times d}$$

The model now works with $\tilde{X}$ instead of `X`.

5.4 The Matrix: Worked Example

Let `T = 4` (4 tokens), `d = 8` (8-dimensional).

First compute the frequencies ω_k for `k = 0, 1, 2, 3`:

$$
\begin{aligned}
\omega_0 &= 1/10000^{0/8} = 1.0 \\
\omega_1 &= 1/10000^{2/8} \approx 0.1 \\
\omega_2 &= 1/10000^{4/8} = 0.01 \\
\omega_3 &= 1/10000^{6/8} \approx 0.001
\end{aligned}
$$

Now compute PE row by row (position `t = 0, 1, 2, 3`):

```
PE[0] = [sin(0·1.0), cos(0·1.0), sin(0·0.1), cos(0·0.1), ...]
      = [0.000,      1.000,      0.000,      1.000,      0.000, 1.000, 0.000, 1.000]

PE[1] = [sin(1·1.0), cos(1·1.0), sin(1·0.1), cos(1·0.1), ...]
      = [0.841,      0.540,      0.0998,     0.995,      0.00999, 0.99995, ...]

PE[2] = [sin(2·1.0), cos(2·1.0), sin(2·0.1), cos(2·0.1), ...]
      = [0.909,     -0.416,      0.198,      0.980,      0.01999, 0.99980, ...]

PE[3] = [sin(3·1.0), cos(3·1.0), sin(3·0.1), cos(3·0.1), ...]
      = [0.141,     -0.990,      0.296,      0.955,      0.02999, 0.99955, ...]
```

Notice: dimension 0-1 (high frequency) changes dramatically between positions. Dimension 6-7 (low frequency) barely changes. Together they form a *unique fingerprint* for each position.

The final input matrix $\tilde{X} = X + PE$:

```
X[t] = embed_vector[t] + PE[t]

For t=0: [-0.90, 0.10, 0.20, -0.30, ...]
       + [ 0.00, 1.00, 0.00,  1.00, ...]
       = [-0.90, 1.10, 0.20,  0.70, ...]
```

Figure Figure 5.1 shows the sinusoidal positional encoding values as a heatmap over positions and dimensions.

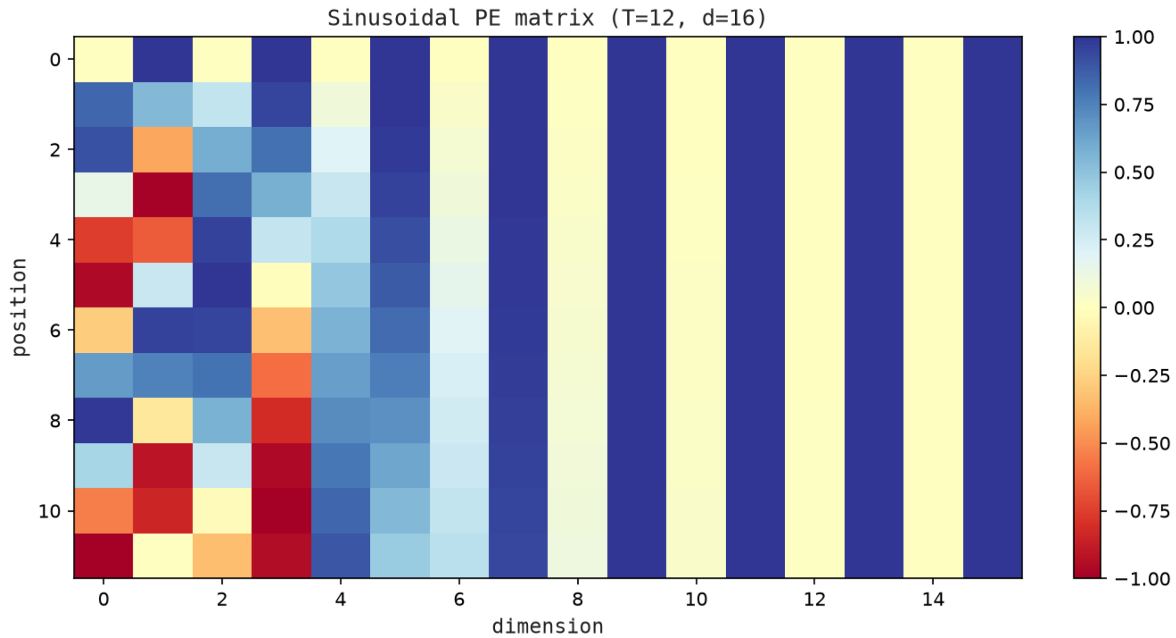

Figure 5.1: Sinusoidal positional encoding heatmap

5.5 The Code: Positional Encoding in Python

```
def make_matrix(rows: int, cols: int, fill: float = 0.0) -> Matrix:
    return [[fill for _ in range(cols)] for _ in range(rows)]


def shape(matrix: Matrix) -> tuple[int, int]:
    return len(matrix), len(matrix[0]) if matrix else 0
```

The matrix data abstraction from Chapter 2 carries over unchanged. `make_matrix` creates a fixed-size grid, and `shape` reads its dimensions.

```
def random_matrix(
    rows: int,
    cols: int,
    rng: random.Random,
    scale: float | None = None,
) -> Matrix:
    scale = math.sqrt(2.0 / (rows + cols)) if scale is None else scale
    return [[rng.uniform(-scale, scale) for _ in range(cols)] for _ in range(rows)]
```

`random_matrix` initializes numeric weights with small random values.

```
def sinusoidal_position(position: int, d_model: int) -> Vector:
    row = []
    for i in range(d_model):
        k = i // 2
        omega = 1.0 / (10000.0 ** (2.0 * k / d_model))
        row.append(math.sin(position * omega) if i % 2 == 0 else math.cos(position * omega))
    return row
```

`sinusoidal_position` encodes one position as a vector of alternating sine and cosine values. Dimension pair k oscillates at frequency $\omega_k = 1/10000^{2k/d}$, so each position gets a unique fingerprint.

```
def sinusoidal_encoding(max_length: int, d_model: int) -> Matrix:
    return [sinusoidal_position(position, d_model) for position in range(max_length)]
```

`sinusoidal_encoding` builds the full $[T \times d]$ matrix by stacking one row per position.

```
def add_positional_encoding(tokens: Matrix, positions: Matrix) -> Matrix:
    return matrix_add(tokens, positions[: len(tokens)])
```

`add_positional_encoding` sums the token embedding matrix X with the PE matrix element-wise, giving $\tilde{X} = X + PE$.

```
def chapter_05() -> dict[str, object]:
    encoding = sinusoidal_encoding(4, 6)
    return {
        "shape": (len(encoding), len(encoding[0])),
        "first": encoding[0],
        "second": encoding[1],
    }
```

The demo builds a small positional encoding matrix and returns its shape plus the first two rows. Run it with `python3 src/python/chapter_demos.py`.

Figure Figure 5.2 shows the positional encoding matrix whose rows are added to token embeddings.

Figure Figure 5.3 shows how sinusoidal waves at different frequencies encode positions across dimensions.

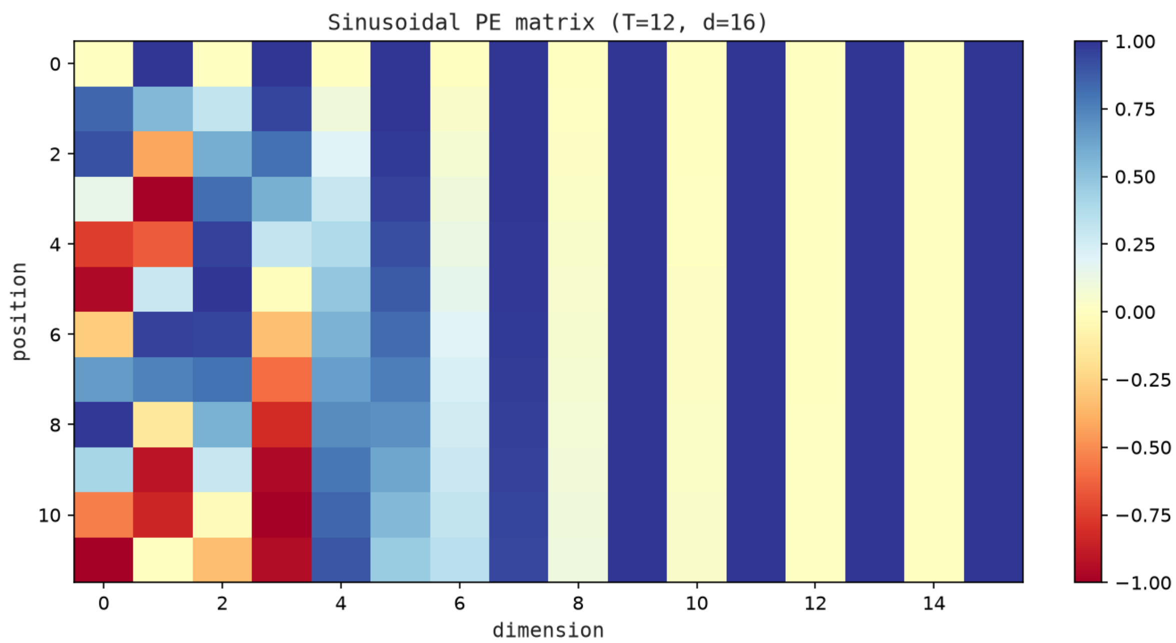

Figure 5.2: Positional encoding matrix

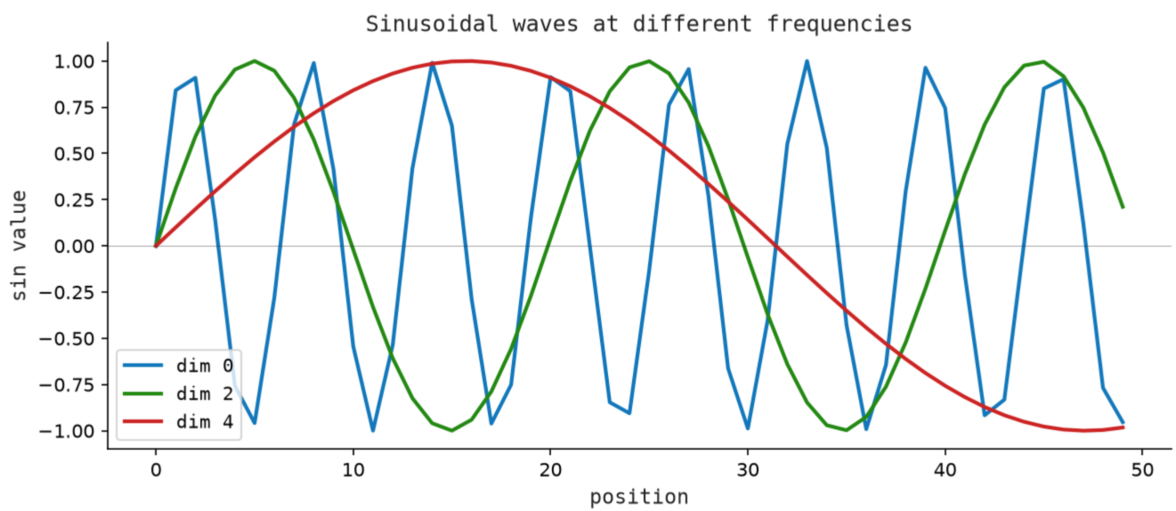

Figure 5.3: Sinusoidal positional frequencies

5.6 Learned vs Sinusoidal Positional Embeddings

GPT-2 and GPT-3 use *learned* positional embeddings — a second embedding matrix $P \in \mathbb{R}^{T_{max} \times d}$, trained alongside the token embedding matrix.

Pros:

- Can adapt to the specific patterns in the training data.

Cons:

- Cannot generalize to sequences longer than T_{max} (the training context length).

Sinusoidal encoding is deterministic and extrapolates naturally. Modern architectures (LLaMA, Mistral) use a more sophisticated variant called *Rotary Positional Encoding (RoPE)* which applies the positional information inside the attention computation rather than at the input stage.

So far, position has been added to the token representation. Modern GPT-style models put position into attention itself rather than adding it at the input. That idea is called RoPE.

5.7 Key Takeaways

- Without position information, the model is *permutation-invariant* — it cannot distinguish `"dog bites man"` from `"man bites dog"`.
- Sinusoidal PE encodes position using sine/cosine at exponentially-spaced frequencies.
- The encoding is *added* to the token embedding: $\tilde{X} = X + PE$.
- Nearby positions have high cosine similarity; distant positions have lower similarity.
- Modern models (GPT-2: learned; LLaMA: RoPE) vary the method but the purpose is the same.

Note

What's next? We now have $\tilde{X} \in \mathbb{R}^{T \times d}$ — a matrix that knows both what each token is and where it sits. Next: *attention*, the mechanism that lets tokens share information with each other. See Chapter 6.

Part III

Transformer Core

This part builds the transformer block from its major components. The chapters move from one attention head to a full block that can be stacked into a GPT model.

- In Chapter 6, each token learns which other tokens to read from.
- In Chapter 7, position moves inside the attention score through rotary position encoding.
- In Chapter 8, several attention patterns run in parallel.
- In Chapter 9, each token representation is transformed by a position-wise neural network.
- In Chapter 10, attention, feed-forward layers, residual connections, and normalization become one reusable block.

6 Attention — Tokens Talking to Each Other

Attention is the engine of the transformer. It is the mechanism that lets each token look at all (or some) of the other tokens and decide which ones to borrow information from. Without attention, each token would be processed independently, oblivious to context. With attention, the word "bank" can tell whether the surrounding tokens are about rivers or finance.

This chapter builds scaled dot-product attention from first principles.

6.1 The Idea

Imagine you are at a library. You arrive with a *query*: "I'm looking for books about Renaissance painting." Every book in the library has a *key* on its spine — a short descriptor of its contents. You compare your query against every key, giving each book a *score*. Then you take a weighted blend of the books' *values* (actual content) — spending more time on high-scoring books.

In the attention mechanism:

- *Query (Q):* "What am I looking for?" — produced from the current token.
- *Key (K):* "What do I contain?" — produced from every token in the sequence.
- *Value (V):* "What information can I give?" — produced from every token in the sequence.

The output for each token is a *weighted sum* of all value vectors, where the weights come from comparing that token's query against every token's key.

6.2 The Math

6.2.1 Step 1: Project to Q, K, V

Given input $X \in \mathbb{R}^{T \times d}$, we learn three weight matrices:

$$W_q \in \mathbb{R}^{d \times d_k}, \quad W_k \in \mathbb{R}^{d \times d_k}, \quad W_v \in \mathbb{R}^{d \times d_v}$$

Compute:

$$Q = XW_q \in \mathbb{R}^{T \times d_k} \quad K = XW_k \in \mathbb{R}^{T \times d_k} \quad V = XW_v \in \mathbb{R}^{T \times d_v}$$

In standard single-head attention, $d_k = d_v = d$.

6.2.2 Step 2: Compute Attention Scores

$$S = QK^\top / \sqrt{d_k} \in \mathbb{R}^{T \times T}$$

`S[i,j]` measures how relevant token `j` is to token `i`. The division by $\sqrt{d_k}$ prevents the dot products from growing too large.

6.2.3 Step 3: Apply Causal Mask (for autoregressive models)

GPT is *autoregressive*: when predicting token `t`, it must not look at tokens `t+1, t+2, …`. We enforce this by masking the upper triangle:

$$M[i,j] = \begin{cases} 0 & \text{if } j \le i \\ -\infty & \text{if } j > i \end{cases}$$

$$S_{\text{masked}} = S + M$$

Adding $-\infty$ before softmax effectively zeroes out those attention weights.

6.2.4 Step 4: Softmax → Attention Weights

$$A = \text{softmax}(S_{\text{masked}}) \in \mathbb{R}^{T \times T}$$

`A[i,j]` is now a probability: how much token `i` attends to token `j`.

6.2.5 Step 5: Weighted Sum of Values

$$\text{Output} = AV \in \mathbb{R}^{T \times d_v}$$

`Output[i]` is a weighted blend of all value vectors, guided by how much token `i` attends to each position.

Full formula:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}} + M\right) \cdot V$$

6.3 The Matrix: Worked Example

Let `T = 3` tokens, `d = 4`, $d_k = d_v = 4$. Input (position-encoded):

```
X = [[ 1.0,  0.0,  1.0,  0.0],   ← token 0: "the"
     [ 0.0,  1.0,  0.0,  1.0],   ← token 1: "cat"
     [ 1.0,  1.0,  0.0,  0.0]]   ← token 2: "sat"
```

Weight matrices (simplified, identity-like): $W_q = W_k = W_v = I_4$ (4×4 identity, so Q=K=V=X).

Step 1 — Q, K, V (same as X here):

```
Q = K = V = X
```

Step 2 — Scores $S = QK^\top/\sqrt{4}$:

```
QK [0,0] = [1,0,1,0]·[1,0,1,0] = 2
QK [0,1] = [1,0,1,0]·[0,1,0,1] = 0
QK [0,2] = [1,0,1,0]·[1,1,0,0] = 1

S = QK  / 2 =
  [[1.0,  0.0,  0.5],
   [0.0,  1.0,  0.5],
   [0.5,  0.5,  1.0]]
```

Step 3 — Causal mask:

```
S_masked =
  [[1.0,   -∞,   -∞],    ← token 0 can only attend to itself
   [0.0,  1.0,   -∞],    ← token 1 can attend to 0 and 1
   [0.5,  0.5,  1.0]]    ← token 2 can attend to all
```

Step 4 — Softmax:

```
A[0] = softmax([1.0, -∞, -∞]) = [1.000, 0.000, 0.000]
A[1] = softmax([0.0, 1.0, -∞]) = [0.269, 0.731, 0.000]
A[2] = softmax([0.5, 0.5, 1.0]) = [0.211, 0.211, 0.578]
```

Step 5 — Output = AV:

```
Output[0] = 1.000·V[0]                                = [1.0, 0.0, 1.0, 0.0]
Output[1] = 0.269·[1,0,1,0] + 0.731·[0,1,0,1]       = [0.269, 0.731, 0.269, 0.731]
Output[2] = 0.211·[1,0,1,0] + 0.211·[0,1,0,1] + 0.578·[1,1,0,0]
          = [0.789, 0.789, 0.211, 0.211]
```

Token 2 (“sat”) is blending information from all three tokens, with most weight on itself.

Figure Figure 6.1 shows Q, K, and V as three separate linear projections of the same input matrix X.

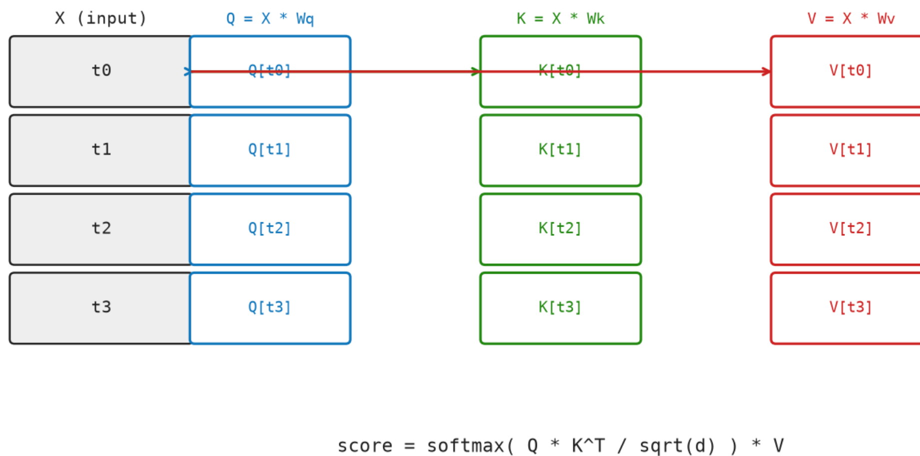

Figure 6.1: Query, key, and value projections

Figure Figure 6.2 shows causal attention weights, where rows are queries, columns are keys, and future tokens are masked.

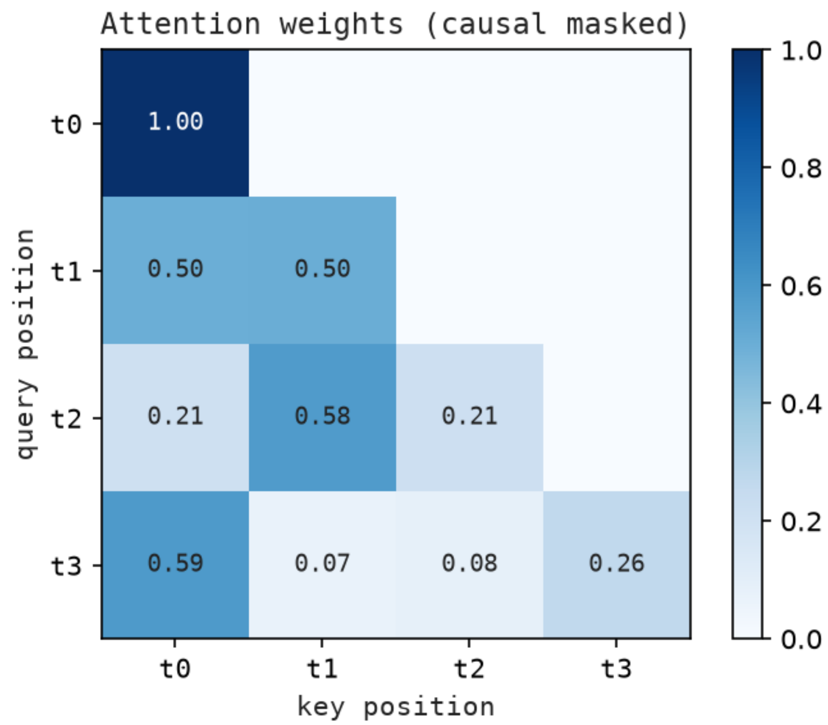

Figure 6.2: Causal attention weight heatmap

6.4 The Code: Scaled Dot-Product Attention in Python

```
def transpose(matrix: Matrix) -> Matrix:
    return [list(col) for col in zip(*matrix)]
```

`transpose` swaps rows and columns so attention can compare every query row with every key row.

```
def matrix_add(left: Matrix, right: Matrix) -> Matrix:
    return [[a + b for a, b in zip(row_a, row_b)] for row_a, row_b in zip(left, right)]
```

`matrix_add` adds two matrices element by element. Attention uses it to add the causal mask to the score matrix.

```
def matrix_scale(matrix: Matrix, scalar: float) -> Matrix:
    return [[scalar * value for value in row] for row in matrix]
```

`matrix_scale` multiplies every entry by the same scalar. Attention uses this for the $1/\sqrt{d_k}$ scaling factor.

```
def matrix_multiply(left: Matrix, right: Matrix) -> Matrix:
    right_t = transpose(right)
    return [[dot(row, col) for col in right_t] for row in left]
```

`matrix_multiply` computes the query-key score matrix and later the weighted sum of values.

```
def hstack(matrices: Sequence[Matrix]) -> Matrix:
    return [sum((matrix[row] for matrix in matrices), []) for row in range(len(matrices[0]))]
```

`hstack` concatenates matrices column-wise. It is used later by multi-head attention after each head has produced its own output.

```
def softmax(logits: Sequence[float]) -> Vector:
    max_logit = max(logits)
    exp_values = [math.exp(value - max_logit) for value in logits]
    total = sum(exp_values)
    return [value / total for value in exp_values]
```

`softmax` converts a vector of raw scores into a probability distribution. It subtracts the maximum value before exponentiating to prevent overflow.

```
def softmax_rows(matrix: Matrix) -> Matrix:
    return [softmax(row) for row in matrix]
```

`softmax_rows` applies the same conversion independently to every row of the attention score matrix.

```
def causal_mask(size: int) -> Matrix:
    return [[0.0 if j <= i else -1.0e9 for j in range(size)] for i in range(size)]
```

`causal_mask` fills the upper triangle with -10^9 so future tokens cannot be attended to.

```
def scaled_dot_product_attention(query: Matrix, key: Matrix, value: Matrix) -> tuple[Matrix,
    d_key = shape(query)[1]
    scores = matrix_scale(matrix_multiply(query, transpose(key)), 1.0 / math.sqrt(d_key))
    masked_scores = matrix_add(scores, causal_mask(len(scores)))
    weights = softmax_rows(masked_scores)
    return matrix_multiply(weights, value), weights


def self_attention(x: Matrix, wq: Matrix, wk: Matrix, wv: Matrix) -> tuple[Matrix, Matrix]:
    return scaled_dot_product_attention(
        matrix_multiply(x, wq),
        matrix_multiply(x, wk),
        matrix_multiply(x, wv),
    )
```

`scaled_dot_product_attention` is a direct translation of the attention formula: scores, mask, softmax, weighted sum. `self_attention` first projects the same input matrix into queries, keys, and values, then calls SDPA.

Run with `python3 src/python/chapter_demos.py`.

6.5 Key Takeaways

- Attention asks: for each token, *which other tokens are most relevant*?
- Q, K, V are *learned linear projections* of the input.
- The attention score $S[i,j] = Q_i \cdot K_j / \sqrt{d_k}$ measures relevance.
- Causal masking prevents tokens from seeing future positions (critical for text generation).

- The output is a *soft, weighted average* of value vectors.
- Full formula: $\text{Attention}(Q, K, V) = \text{softmax}(QK^\top/\sqrt{d_k} + \text{mask})V$.

> **Note**
>
> *Up next: RoPE.* Attention now has queries, keys, values, scores, masks, and weighted sums. Modern GPT-style models often add one more refinement before scaling to many heads: they put position directly into the query-key comparison. That mechanism is *RoPE* — Chapter 7.

7 RoPE: Position Inside Attention

So far, position has been added to the token representation. Sinusoidal and learned positional embeddings both create a position vector and add it to the token embedding before attention begins.

Modern GPT-style models often do something more subtle: they put position into attention itself. That idea is called *Rotary Positional Encoding*, or *RoPE*.

7.1 The Idea

Standard positional encoding changes the input stream:

$$\tilde{X} = X + PE$$

Attention then projects $\tilde{X}$ into Q, K, and V.

RoPE changes a different point in the pipeline. It first computes the usual query, key, and value vectors:

$$Q = XW_q, \quad K = XW_k, \quad V = XW_v$$

Then it rotates each query and key according to that token's position. Values are not rotated.

The attention score is still a dot product, but now it compares rotated vectors:

$$S[i, j] = \frac{(R_i q_i) \cdot (R_j k_j)}{\sqrt{d_k}}$$

This lets position affect *which tokens attend to which other tokens*. Instead of saying "position is another feature in the token vector," RoPE says "position changes the geometry of query-key matching."

7.2 The Math

RoPE treats a vector as pairs of dimensions. For each pair, it applies a 2D rotation.

For a vector pair (x, y) and angle θ:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

For token position m and dimension pair r, RoPE uses:

$$\theta_{m,r} = \frac{m}{10000^{2r/d}}$$

So each pair rotates at a different frequency. Early dimensions rotate quickly; later dimensions rotate slowly.

The important property is what happens inside the attention dot product:

$$(R_m q) \cdot (R_n k) = q \cdot (R_{n-m} k)$$

The score depends on the relative distance $n - m$ between the query position and key position. That is how RoPE encodes relative position without adding a separate position vector to X.

The full attention formula becomes:

$$\text{Attention}_{\text{RoPE}}(Q, K, V) = \text{softmax}\left(\frac{(RQ)(RK)^\top}{\sqrt{d_k}} + M\right) V$$

where RQ means "rotate every query row by its position" and RK means the same for keys.

7.3 A Small Example

Take a two-dimensional query and key:

$$q = [1, 0], \quad k = [1, 0]$$

Before rotation, their dot product is:

$$q \cdot k = 1$$

Now place the query at position 1 and the key at position 3. In two dimensions the frequency is 1, so the rotations use angles 1 and 3 radians:

$$R_1 q = [\cos 1, \sin 1]$$

$$R_3 k = [\cos 3, \sin 3]$$

Their dot product is:

$$\cos 1 \cos 3 + \sin 1 \sin 3 = \cos(3 - 1)$$

The result depends on the distance between positions, not just on the original vector contents. For real model dimensions, many dimension pairs rotate at many frequencies, so attention gets a detailed relative-position signal.

7.4 The Code Structure

```
def rope_angle(position: int, pair_index: int, d_model: int) -> float:
    return position / (10000.0 ** (2.0 * pair_index / d_model))
```

`rope_angle` computes the rotation angle for one position and one pair of dimensions. The formula matches the sinusoidal frequency schedule: early dimension pairs rotate faster, later pairs rotate more slowly.

```
def rotate_pair(x: float, y: float, angle: float) -> tuple[float, float]:
    cos_theta = math.cos(angle)
    sin_theta = math.sin(angle)
    return x * cos_theta - y * sin_theta, x * sin_theta + y * cos_theta
```

`rotate_pair` is the 2D rotation from the math section. It takes two neighboring vector components and returns the rotated pair.

```
def apply_rope_to_vector(vector: Vector, position: int) -> Vector:
    if len(vector) % 2 != 0:
        raise ValueError("RoPE requires an even vector dimension")

    rotated: Vector = []
```

```python
    d_model = len(vector)
    for i in range(0, d_model, 2):
        angle = rope_angle(position, i // 2, d_model)
        x, y = rotate_pair(vector[i], vector[i + 1], angle)
        rotated.extend([x, y])
    return rotated
```

`apply_rope_to_vector` walks through a query or key vector two dimensions at a time. Each pair gets its own angle, determined by the token position and the pair index. RoPE requires an even vector dimension because every dimension must belong to a pair.

```python
def apply_rope(matrix: Matrix) -> Matrix:
    return [
        apply_rope_to_vector(row, position)
        for position, row in enumerate(matrix)
    ]
```

`apply_rope` applies that vector rotation to every row in a matrix. The row index is the token position, so row `0` gets position `0`, row `1` gets position `1`, and so on.

```python
def scaled_dot_product_attention_with_rope(
    query: Matrix,
    key: Matrix,
    value: Matrix,
) -> tuple[Matrix, Matrix]:
    return scaled_dot_product_attention(
        apply_rope(query),
        apply_rope(key),
        value,
    )
```

`scaled_dot_product_attention_with_rope` keeps the attention formula from Chapter 6. The only difference is that queries and keys are rotated before the dot products are computed. Values are not rotated; they are still mixed by the final attention weights.

```python
def chapter_07(seed: int = 607) -> dict[str, object]:
    rng = random.Random(seed)
    query = random_matrix(4, 8, rng)
    key = random_matrix(4, 8, rng)
    value = random_matrix(4, 8, rng)
    output, weights = scaled_dot_product_attention_with_rope(query, key, value)
```

```python
    return {
        "rotated_first": apply_rope_to_vector([1.0, 0.0], 1),
        "rotated_shape": (len(apply_rope(query)), len(query[0])),
        "output_shape": (len(output), len(output[0])),
        "weight_rows": [sum(row) for row in weights],
    }
```

`chapter_07` is the runnable demo used by `python3 src/python/chapter_demos.py`. It checks that RoPE preserves the query/key matrix shape and that the resulting attention rows still sum to 1 after softmax.

7.5 Takeaways

- Earlier positional encodings add position to the token representation before attention.
- RoPE (Su et al., 2021) puts position inside attention by rotating Q and K.
- Values are not rotated; they are still blended by the attention weights.
- The query-key dot product depends only on the relative distance between positions, not absolute indices.
- No extra parameters — the rotation is deterministic.
- Generalises to sequences longer than those seen during training.
- Compatible with linear attention variants.
- Used in LLaMA, Mistral, Qwen, and most open-weight models.

> **Note**
>
> *What's next?* RoPE changes how one attention head sees position. The next chapter asks how a model can track several kinds of relationships at once: *multi-head attention*. See Chapter 8.

8 Multi-Head Attention — Many Conversations at Once

A single attention head is expressive, but limited. It produces one set of attention weights — one pattern of "who attends to whom." In natural language, multiple distinct relationships coexist in the same sentence. Consider:

```
"The animal didn't cross the street because it was too tired."
```

- One pattern might link "it" → "animal" (coreference)
- Another might link "cross" → "street" (verb-object)
- Another might track the causal structure "because"

Multi-head attention runs `H` independent attention heads in parallel, each free to specialize on a different relationship type. Their outputs are concatenated and projected back to the model dimension.

8.1 The Idea

A single attention operation produces one pattern of "who attends to whom." But a sentence carries many different kinds of relationships at the same time.

In "The animal didn't cross the street because it was too tired":

- *it* refers back to *animal* — that is a *coreference* relationship.
- *cross* links to *street* — that is a *verb-object* relationship.
- *because* connects a cause to an effect — that is a *logical* relationship.

One attention head can only focus on one of these at a time. Multi-head attention runs several independent attention operations *in parallel* — each one free to specialize on a different pattern. Each head sees the same input but learns to ask a different question of it.

At the end, the results from all heads are stitched back together and projected into a single vector, the same size as before. The model learns entirely from data which head should track grammar, which should track meaning, which should track proximity — no one programs this in explicitly.

The result is a richer representation than any single head could produce: each token's final vector carries signals from multiple independent attention patterns at once.

8.2 The Math

Step 1 — Compute each head independently.

Each head $h \in \{1, \ldots, H\}$ projects the input into its own query, key, and value spaces, then runs a standard attention:

$$
\begin{aligned}
Q^h &= XW_q^h \in \mathbb{R}^{T \times d_k} \\
K^h &= XW_k^h \in \mathbb{R}^{T \times d_k} \\
V^h &= XW_v^h \in \mathbb{R}^{T \times d_v} \\
\text{head}^h &= \text{softmax}\left(\frac{Q^h {K^h}^\top}{\sqrt{d_k}} + M\right) V^h
\end{aligned}
$$

Step 2 — Concatenate.

The H outputs are placed side by side:

$$
\text{MultiHead} = \text{concat}(\text{head}^1, \text{head}^2, \ldots, \text{head}^H) \in \mathbb{R}^{T \times (H \cdot d_v)}
$$

Because $d_v = d/H$, the result has shape $[T \times d]$ — the same as the input.

Step 3 — Output projection.

A learned matrix $W_o \in \mathbb{R}^{d \times d}$ mixes information across heads:

$$
\text{Output} = \text{MultiHead} \cdot W_o \in \mathbb{R}^{T \times d}
$$

Putting it all together:

$$
\text{MHA}(X) = [\text{head}^1 \| \text{head}^2 \| \ldots \| \text{head}^H] \, W_o
$$

where $\text{head}^h = \text{Attention}(XW_q^h, XW_k^h, XW_v^h)$.

8.3 The Matrix: Worked Example

Let `T = 3`, `d = 4`, `H = 2` heads, so $d_k = d_v = 2$.

Input:

```
X = [[1, 0, 1, 0],
     [0, 1, 0, 1],
     [1, 1, 0, 0]]   (3×4)
```

Head 1 uses the first 2 dimensions primarily.

```
Wq¹ = Wk¹ = Wv¹ = [[1,0],[0,1],[0,0],[0,0]]   (4×2)

Q¹ = X Wq¹ = [[1,0],[0,1],[1,1]]   (3×2)
```

Scores: $S^1 = Q^1 {K^1}^\top / \sqrt{2}$:

```
Q¹K¹  = [[1,0,1],[0,1,1],[1,1,2]]
S¹     = [[0.71, 0.00, 0.71],
          [0.00, 0.71, 0.71],
          [0.71, 0.71, 1.41]]
```

After causal mask and softmax:

```
A¹ = [[1.000, 0.000, 0.000],
      [0.414, 0.586, 0.000],
      [0.221, 0.221, 0.558]]

head¹ = A¹ V¹ = [[1.000, 0.000],
                 [0.586, 0.414],
                 [0.779, 0.779]]    (3×2)
```

Head 2 focuses on last 2 dimensions, producing similarly shaped output. After concatenating both heads and applying the output projection $W_o \in \mathbb{R}^{4\times 4}$:

```
MultiHead = [head¹ ‖ head²] =
  [[1.000, 0.000, 1.000, 0.000],
   [0.586, 0.414, 0.500, 0.500],
   [0.779, 0.779, 0.421, 0.211]]   (3×4)
```

Figure Figure 8.1 shows parallel attention heads whose outputs are concatenated and projected back to model width.

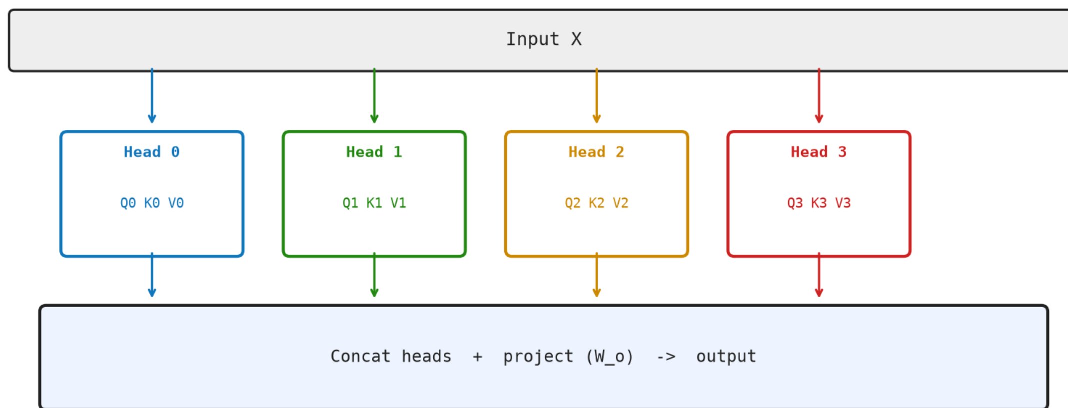

Figure 8.1: Multi-head attention

8.4 The Code: Multi-Head Attention in Python

```
@dataclass
class AttentionHead:
    wq: Matrix
    wk: Matrix
    wv: Matrix


@dataclass
class MultiHeadAttention:
    heads: list[AttentionHead]
    wo: Matrix


def make_multi_head_attention(d_model: int, num_heads: int, rng: random.Random) -> MultiHead
    if d_model % num_heads != 0:
        raise ValueError("d_model must be divisible by num_heads")
    d_key = d_model // num_heads
    heads = [
        AttentionHead(
            random_matrix(d_model, d_key, rng),
            random_matrix(d_model, d_key, rng),
```

```
            random_matrix(d_model, d_key, rng),
        )
        for _ in range(num_heads)
    ]
    return MultiHeadAttention(heads=heads, wo=random_matrix(d_model, d_model, rng))
```

Each attention head is a triple of weight matrices (W_q, W_k, W_v), each of shape $[d \times d_k]$. `MultiHeadAttention` groups the heads with the output projection W_o. `make_multi_head_attention` checks that the model width can be split evenly, allocates one parameter set per head, and creates the final output projection.

```
def multi_head_attention(x: Matrix, params: MultiHeadAttention) -> tuple[Matrix, list[Matrix
    results = [
        self_attention(x, head.wq, head.wk, head.wv)
        for head in params.heads
    ]
    concatenated = hstack([output for output, _weights in results])
    return matrix_multiply(concatenated, params.wo), [weights for _output, weights in result
```

`multi_head_attention` runs each head's SDPA independently, concatenates the per-head outputs column-wise, and applies the output projection.

```
def chapter_08(seed: int = 7) -> dict[str, object]:
    rng = random.Random(seed)
    x = random_matrix(4, 8, rng)
    params = make_multi_head_attention(8, 2, rng)
    output, weights = multi_head_attention(x, params)
    return {
        "output_shape": (len(output), len(output[0])),
        "num_heads": len(weights),
    }
```

The demo creates a two-head attention module, runs it over a four-token sequence, and reports the output shape plus the number of attention-weight matrices. Run it with `python3 src/python/chapter_demos.py`.

8.5 Why Multi-Head Attention Works

Each head learns to specialize. Research has identified heads that:

- Track *syntactic structure* (subject-verb agreement)
- Resolve *coreference* (“it” → “the cat”)
- Handle *positional offsets* (“look 2 tokens back”)
- Track *rare-word semantics*

These specializations emerge from training, not from explicit design.

> **Note**
>
> *Math Minute — Expressivity*
> H heads of dimension d/H can represent attention patterns that a single head of dimension d cannot easily learn. This is analogous to having H different “lenses” looking at the same sequence; each lens focuses on different features. The output projection `Wo` then combines the views.

8.6 Key Takeaways

- Multi-head attention runs `H` attention heads *in parallel*, each in a lower-dimensional subspace $d_k = d/H$.
- Each head has independent W_q, W_k, W_v matrices — each learns a different “question to ask.”
- Outputs are *concatenated* (not averaged), then projected back to `d` with W_o.
- The total parameter count is $3Hd \cdot d_k + d^2 = 3d^2 + d^2$ — same as one large head.
- Different heads specialize in different linguistic relationships.

> **Note**
>
> *What’s next?* After attention mixes information across tokens, each token’s vector goes through a small *feed-forward network* — a two-layer MLP applied identically to every position. This is where most of the model’s stored “knowledge” lives. See Chapter 9.

9 Feed-Forward Network — The Model's Memory

After multi-head attention has blended information across positions, each token's vector passes through a *feed-forward network (FFN)*. This is a simple two-layer MLP — the same network, applied independently to every position.

While attention does the *routing* (mixing information across tokens), the FFN does the *processing* (transforming each token's representation). Experiments by Geva et al. (2021) showed that the FFN layers act as "key-value memories" — the first layer's neurons trigger on specific input patterns, and the second layer retrieves associated information.

9.1 The Idea

Attention is about *communication* — each token collects information from other tokens. The feed-forward network is about *computation* — each token processes what it just collected, entirely on its own.

After attention, every token's vector has been updated with context from the surrounding words. The FFN now takes each updated vector and runs it through a private transformation: expand it into a much larger space, filter it through a non-linearity that decides which features are "active," then compress it back to the original size. No information crosses between tokens here — every position is handled independently with the same set of weights.

Why expand and then compress? The expansion gives the model room to express a large number of potential features at once. The non-linearity then selects which combinations matter and switches the rest off. The compression packages the result back into the model's standard vector size.

This expand-filter-compress pipeline is where most of the model's factual knowledge lives. Geva et al. (2021) found that individual neurons in the FFN's expanded space correlate with specific concepts ("Paris," "past tense," "chemical element").

The same FFN weights are applied to every token position in the sequence: position 1 and position 100 go through identical transformations. Only attention sees position.

9.2 The Math

> **Note**
>
> *Math Minute — ReLU and GELU*
> A *non-linearity* is a function that makes the network capable of learning complex patterns beyond linear mappings.
> `ReLU(x) = max(0, x)` — simple: negative values → 0, positive values unchanged.
> $\text{GELU}(x) = x \cdot \Phi(x)$ where Φ is the standard normal CDF. GELU is smooth and probabilistic. GPT uses GELU. LLaMA and most modern models use `SwiGLU`, a gated variant.
> `Sigmoid(x) = 1/(1+e )` squashes any real number to (0,1). Used in gates.

For a sequence input $X \in \mathbb{R}^{T \times d}$, the FFN applies identically to each row:

$$FFN(X) = \text{GELU}(XW_1 + \mathbf{1} \cdot b_1^\top)W_2 + \mathbf{1} \cdot b_2^\top$$

Breaking it down:

$$\begin{aligned} H &= XW_1 + \mathbf{1}b_1^\top \in \mathbb{R}^{T \times d_{\text{ff}}} && \text{(expand)} \\ H' &= \text{GELU}(H) \in \mathbb{R}^{T \times d_{\text{ff}}} && \text{(activate)} \\ Y &= H'W_2 + \mathbf{1}b_2^\top \in \mathbb{R}^{T \times d} && \text{(contract)} \end{aligned}$$

Parameter count:

- W_1: $d \times d_{\text{ff}} = d \times 4d = 4d^2$
- W_2: $d_{\text{ff}} \times d = 4d^2$
- Total: $\approx 8d^2$ — *twice* the parameter count of all attention weight matrices combined!

9.3 GELU Deep Dive

The exact GELU formula is:

$$\text{GELU}(x) = x \cdot \frac{1}{2} \cdot \left[1 + \text{erf}\left(\frac{x}{\sqrt{2}}\right)\right]$$

A fast approximation (used in practice):

$$\text{GELU}(x) \approx 0.5 \cdot x \cdot (1 + \tanh(\sqrt{2/\pi} \cdot (x + 0.044715 \cdot x^3)))$$

Key behaviors:

- `GELU(0) = 0`
- For large positive `x`: $\text{GELU}(x) \approx x$ (passes through)
- For large negative `x`: $\text{GELU}(x) \approx 0$ (suppressed)
- Smooth everywhere — gradient flows through during training

9.4 The Matrix: Worked Example

Let `T = 2`, `d = 4`, $d_{\text{ff}} = 8$.

Input (after attention):

```
X = [[ 0.5,  1.2, -0.3,  0.8],
     [-0.1,  0.7,  0.9, -0.4]]    (2×4)
```

Tracing row 0 of X: `x = [0.5, 1.2, -0.3, 0.8]`

Step 1 — Expand: $h = xW_1$

```
h = [1.1, -0.4, 0.8, 2.1, -0.3, 0.6, 1.5, -0.8]
```

Step 2 — GELU:

```
GELU(1.1)    0.95
GELU(-0.4)   -0.12
GELU(0.8)    0.64
GELU(2.1)    2.06
GELU(-0.3)   -0.11
GELU(0.6)    0.44
GELU(1.5)    1.40
GELU(-0.8)   -0.17

h' = [0.95, -0.12, 0.64, 2.06, -0.11, 0.44, 1.40, -0.17]
```

Step 3 — Contract: $y = h'W_2$

```
y   [0.7, 1.1, -0.2, 0.9]
```

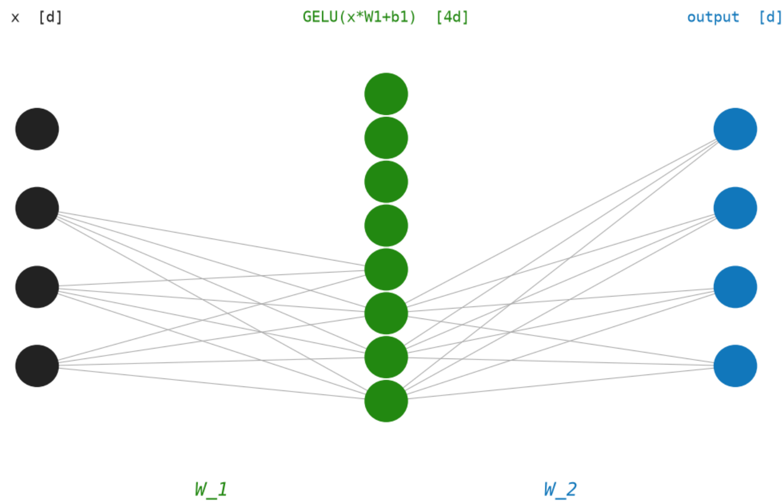

Figure 9.1: Feed-forward network

The output **y** is a transformed version of the input vector — same dimensionality, different values. The transformation was learned to improve next-token prediction.

Figure Figure 9.1 shows the feed-forward network expanding the hidden dimension, applying GELU, and contracting back to model width.

Figure Figure 9.2 shows the FFN interpretation as associative memory, with keys and values stored in its weight matrices.

9.5 The Code: FFN in Python

```
def sigmoid(x: float) -> float:
    return 1.0 / (1.0 + math.exp(-x))


def relu(x: float) -> float:
    return max(0.0, x)


def gelu(x: float) -> float:
    scale = math.sqrt(2.0 / math.pi)
    return 0.5 * x * (1.0 + math.tanh(scale * (x + 0.044715 * x**3)))
```

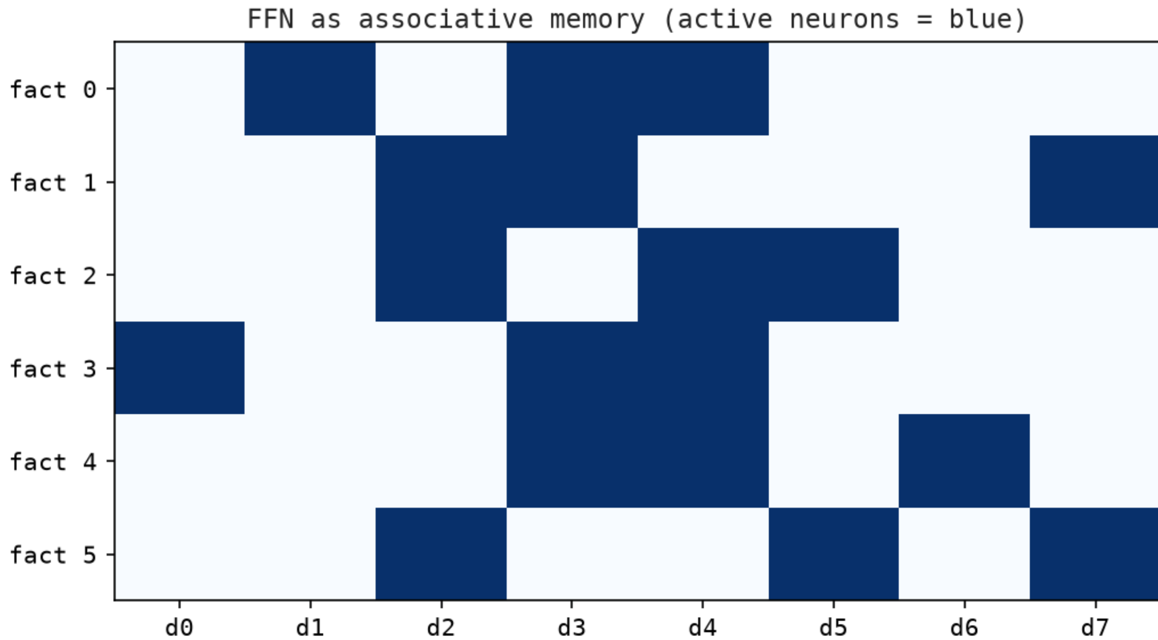

Figure 9.2: Feed-forward network as associative memory

```
def gelu_matrix(matrix: Matrix) -> Matrix:
    return [[gelu(value) for value in row] for row in matrix]
```

GELU is the activation function used in GPT-2. For large positive x, GELU approaches x; for large negative x, it approaches 0. `gelu_matrix` applies the scalar GELU to every element of a matrix.

```
def add_bias(matrix: Matrix, bias: Vector) -> Matrix:
    return [[value + bias[j] for j, value in enumerate(row)] for row in matrix]
```

`add_bias` broadcasts a bias vector b across all rows of a matrix M.

```
@dataclass
class FeedForward:
    w1: Matrix
    b1: Vector
    w2: Matrix
    b2: Vector
```

```python
def make_feed_forward(d_model: int, rng: random.Random) -> FeedForward:
    d_hidden = 4 * d_model
    return FeedForward(
        w1=random_matrix(d_model, d_hidden, rng),
        b1=[0.0] * d_hidden,
        w2=random_matrix(d_hidden, d_model, rng),
        b2=[0.0] * d_model,
    )
```

`FeedForward` holds the four parameters. `make_feed_forward` allocates them: W_1 expands the model dimension d to $4d$, and W_2 contracts it back.

```python
def feed_forward(x: Matrix, params: FeedForward) -> Matrix:
    hidden = gelu_matrix(add_bias(matrix_multiply(x, params.w1), params.b1))
    return add_bias(matrix_multiply(hidden, params.w2), params.b2)
```

`feed_forward` composes the two linear layers with GELU: expand, activate, contract.

```python
def chapter_09(seed: int = 8) -> dict[str, object]:
    rng = random.Random(seed)
    x = random_matrix(3, 8, rng)
    params = make_feed_forward(8, rng)
    output = feed_forward(x, params)
    return {"output_shape": (len(output), len(output[0]))}
```

The demo applies one FFN to a three-token sequence and returns the unchanged outer shape. Run it with `python3 src/python/chapter_demos.py`.

9.6 The FFN as a Key-Value Memory

A striking insight from Geva et al. (2021): the FFN layers act like *associative memories*.

The first layer W_1 stores "keys" — each column of W_1 is a pattern to match against the input. GELU fires when the input matches. The second layer W_2 stores "values" — the corresponding information to retrieve.

Concretely:

$$h = \text{GELU}(xW_1)$$

where each component $h_k \approx x \cdot W_1[:, k]$ — a dot-product "match score" for pattern k. High h_k means "input matches pattern k."

$$y = hW_2$$

Pull out value $W_2[k, :]$ weighted by h_k.

This means the model can learn: "if the input contains patterns related to France, activate neuron 47, which retrieves 'Paris' associations."

9.7 Key Takeaways

- The FFN is a two-layer MLP applied *identically to every position*: $FFN(x) = \text{GELU}(xW_1 + b_1)W_2 + b_2$.
- The inner dimension $d_{\text{ff}} = 4d$ means the FFN has *more parameters than the attention layers*.
- GELU is a smooth activation function that "softly gates" negative values.
- The FFN acts as an *associative memory*: first layer matches patterns, second layer retrieves associated information.
- No cross-position communication happens here — that's attention's job.

> **ℹ Note**
>
> *What's next?* We now have all the pieces: attention that mixes information, and an FFN that processes each token's vector. How are they wired together? With *residual connections* and *layer normalization* to form the *transformer block* — Chapter 10.

10 The Transformer Block — Putting It Together

A transformer block combines multi-head attention and the feed-forward network with two crucial wiring techniques: *residual connections* and *layer normalization*. Without these, deep stacks of transformer blocks would fail to train — gradients would vanish or explode.

A GPT model is simply a stack of `N` identical transformer blocks. GPT-2 small: 12 blocks. GPT-3 175B: 96 blocks.

10.1 The Idea

A GPT model is a stack of identical blocks — GPT-2 small has 12, GPT-3 has 96. Stacking many layers should make the model more powerful, but in practice it causes two problems: information gets distorted as it passes through many transformations, and very deep networks are notoriously hard to train.

A transformer block solves both problems with two simple ideas.

Residual connections (also called skip connections): instead of replacing a vector entirely, each sub-layer only adds its output to the original. The input is preserved and combined with the new information. Think of it as writing notes in the margin rather than rewriting the whole page. Each layer only needs to learn *what to change*, not what the whole answer should be. This keeps information flowing cleanly all the way through, even in very deep stacks.

Layer normalization: the numbers inside a vector can grow or shrink unpredictably as the model trains. If they drift too far, the training process becomes unstable. Layer normalization resets them to a consistent scale before each sub-layer processes them. It is a housekeeping step: it does not change which direction the vector points, just how large the numbers are.

Together, these two techniques make it possible to stack dozens of transformer blocks without losing control of training.

10.2 The Math

A single transformer block (Pre-LN variant, used by most modern GPTs):

$$x_1 = x + \text{MHA}(\text{LayerNorm}(x)) \quad \text{(Multi-Head Attention sub-layer)}$$
$$x_2 = x_1 + \text{FFN}(\text{LayerNorm}(x_1)) \quad \text{(Feed-Forward Network sub-layer)}$$

This is the *Pre-Layer-Norm* architecture. The original "Attention Is All You Need" paper used Post-LN, but Pre-LN is more stable to train and is used in GPT-2 onwards.

10.2.1 Layer Normalization

For a vector $x \in \mathbb{R}^d$:

$$\text{LayerNorm}(x) = \gamma \odot \frac{x - \mu}{\sigma} + \beta$$

where $\mu = \frac{1}{d}\sum_i x_i$, $\sigma = \sqrt{\frac{1}{d}\sum_i (x_i - \mu)^2}$, $\gamma, \beta \in \mathbb{R}^d$ are learned scale and shift, and $\odot$ is element-wise multiplication.

> **Note**
>
> *Math Minute — Variance and Standard Deviation*
> The variance of a set of numbers $\{x_1, \ldots, x_n\}$ measures how spread out they are: $Var = (1/n)\sum_i (x_i - \mu)^2$ where μ is the mean. The standard deviation $\sigma = \sqrt{Var}$ is in the same units as the original values. Dividing by σ makes the spread equal to 1 — "standardizing."

10.3 The Residual Stream

A powerful way to understand the GPT architecture is through the lens of the *residual stream* (Elhage et al., Anthropic 2021):

The input embedding is injected into a "stream" — a vector of dimension `d` per token. Each transformer block *reads from* this stream (via attention and FFN) and *adds back to* it (via residual connections). The stream carries information across all blocks.

```
stream  = token_embeddings + positional_encodings    [T × d]
stream¹ = stream  + MHA(LN(stream ))
stream¹ = stream¹ + FFN(LN(stream¹))
stream² = stream¹ + MHA(LN(stream¹))
stream² = stream² + FFN(LN(stream²))

stream  = final output
```

This view clarifies that attention heads and FFN neurons are all writing to the same shared workspace.

10.4 The Matrix: Worked Example

Let `T = 2`, `d = 4`.

Input embedding (position-encoded):

```
x = stream [0] = [1.0, -0.5, 0.8, -0.2]
```

LayerNorm step:

```
  = (1.0 - 0.5 + 0.8 - 0.2) / 4 = 0.275
 ² = [(1.0-0.275)² + (-0.5-0.275)² + (0.8-0.275)² + (-0.2-0.275)²] / 4
   = 0.407
  = √0.407   0.638

x_norm = [1.136, -1.215, 0.823, -0.745]
(With  =1,  =0 for simplicity)
```

MHA produces (suppose): `mha_out = [0.3, 0.7, -0.1, 0.5]`

Residual connection:

```
x  = x + mha_out = [1.0+0.3, -0.5+0.7, 0.8-0.1, -0.2+0.5]
   = [1.3, 0.2, 0.7, 0.3]
```

Second LayerNorm + FFN + residual (analogously) $\rightarrow x_2$.

x_1 contains *both* the original information (from `x`) and the new information (from `mha_out`). Nothing is overwritten — the residual stream accumulates.

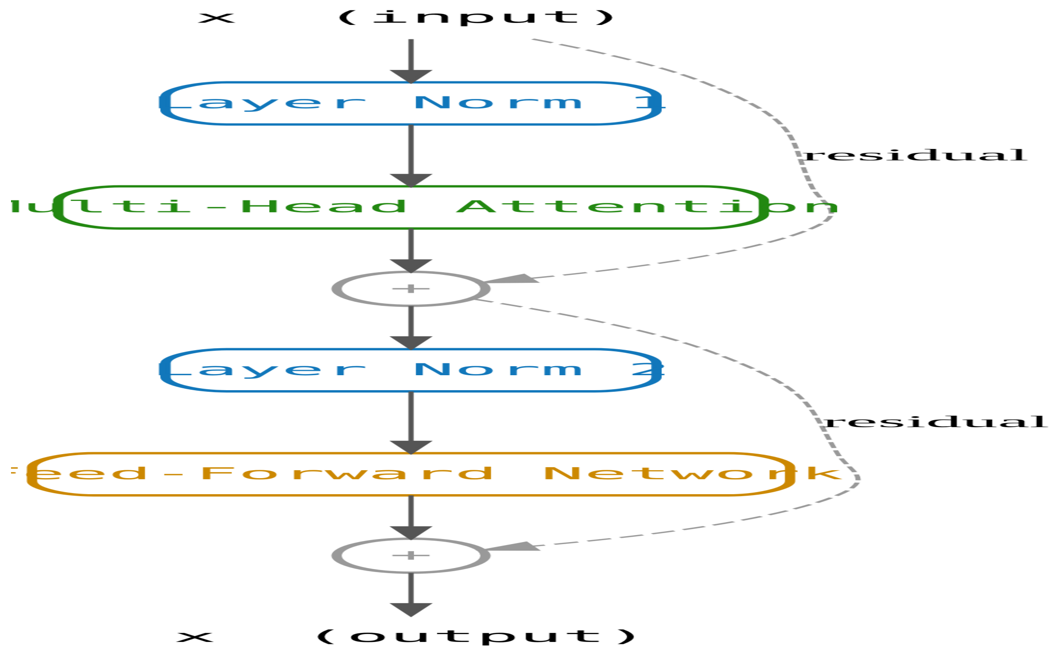

Figure 10.1: Full transformer block (Pre-LN) — LayerNorm before each sub-layer, residual connections around both.

Figure 10.1 shows the full Pre-LN transformer block with LayerNorm before each sub-layer and residual connections around both.

Figure 10.2 shows the residual stream carrying information forward while gradients flow backward through additions.

10.5 The Code: Transformer Block in Python

```
@dataclass
class LayerNorm:
    gamma: Vector
    beta: Vector
    epsilon: float = 1.0e-5


def make_layer_norm(d_model: int) -> LayerNorm:
    return LayerNorm(gamma=[1.0] * d_model, beta=[0.0] * d_model)
```

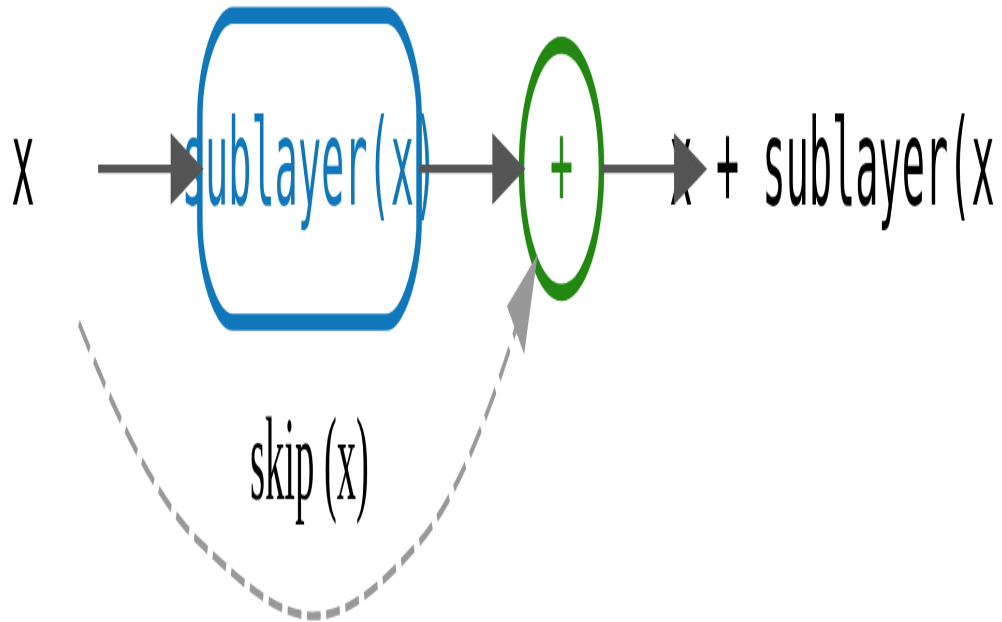

Figure 10.2: The residual stream — information flows unimpeded through addition, gradients flow backward.

```
def layer_norm_vector(vector: Vector, params: LayerNorm) -> Vector:
    avg = mean(vector)
    std = math.sqrt(variance(vector) + params.epsilon)
    return [
        params.gamma[i] * ((value - avg) / std) + params.beta[i]
        for i, value in enumerate(vector)
    ]
```

Layer normalization shifts a single vector to mean 0, variance 1, then applies learned scale and shift. `LayerNorm` holds the learned parameters γ and β. `make_layer_norm` initializes them to identity.

```
def layer_norm_matrix(matrix: Matrix, params: LayerNorm) -> Matrix:
    return [layer_norm_vector(row, params) for row in matrix]
```

`layer_norm_matrix` applies `layer_norm_vector` independently to each row of a $[T \times d]$ matrix.

```
@dataclass
class TransformerBlock:
    attention: MultiHeadAttention
    feed_forward: FeedForward
    ln1: LayerNorm
    ln2: LayerNorm


def make_transformer_block(d_model: int, num_heads: int, rng: random.Random) -> TransformerBl
    return TransformerBlock(
        attention=make_multi_head_attention(d_model, num_heads, rng),
        feed_forward=make_feed_forward(d_model, rng),
        ln1=make_layer_norm(d_model),
        ln2=make_layer_norm(d_model),
    )
```

`TransformerBlock` groups the attention module, feed-forward module, and two layer norms. `make_transformer_block` allocates their parameters.

```
def transformer_block(x: Matrix, params: TransformerBlock) -> Matrix:
    attn_out, _weights = multi_head_attention(layer_norm_matrix(x, params.ln1), params.atten
    x1 = matrix_add(x, attn_out)
    ffn_out = feed_forward(layer_norm_matrix(x1, params.ln2), params.feed_forward)
    return matrix_add(x1, ffn_out)


def forward_stack(x: Matrix, blocks: Iterable[TransformerBlock]) -> Matrix:
    for block in blocks:
        x = transformer_block(x, block)
    return x
```

`transformer_block` is the Pre-LN residual block: normalize, attend, add the residual, normalize, apply FFN, add the residual again. `forward_stack` threads the sequence through N blocks.

```
def chapter_10(seed: int = 9) -> dict[str, object]:
    rng = random.Random(seed)
    x = random_matrix(3, 8, rng)
    block = make_transformer_block(8, 2, rng)
    normed = layer_norm_matrix(x, make_layer_norm(8))
    output = transformer_block(x, block)
    return {
```

```
        "normed_shape": (len(normed), len(normed[0])),
        "output_shape": (len(output), len(output[0])),
    }
```

The demo runs one transformer block over a three-token sequence and checks that layer norm and the block both preserve the outer shape. Run it with `python3 src/python/chapter_demos.py`.

10.6 Why N Blocks?

A single transformer block has limited capacity. Each block:

- Runs one round of attention (all heads) — *mixes information globally*
- Runs one FFN per position — *processes each token's updated representation*

But complex language tasks require many rounds of computation. By stacking `N` blocks, the model builds increasingly abstract representations:

- Early blocks: local patterns, syntax, common phrases
- Middle blocks: semantic roles, entity tracking
- Late blocks: task-specific reasoning, pragmatics

The residual stream carries information across all blocks. Later blocks can read from and write to everything earlier blocks computed.

10.7 Key Takeaways

- A transformer block = `MHA → residual → LN → FFN → residual → LN` (Pre-LN ordering).
- *Residual connections* let gradients flow directly and prevent vanishing gradients.
- *Layer normalization* keeps activations at mean 0, std 1 within each sub-layer.
- The *residual stream* is the shared workspace that all blocks read from and write to.
- GPT stacks `N` identical blocks; capacity scales with `N` and `d`.

> **ℹ Note**
>
> *What's next?* After `N` blocks, we have a final matrix $X_{\text{final}} \in \mathbb{R}^{T \times d}$. Each row is a rich representation of the corresponding token in context. The last step: turn the final vector for position `T` into a probability distribution over the vocabulary — that is *vocabulary projection* in Chapter 11.

Part IV

Prediction and Learning

This part follows the model from final hidden states to training updates. It explains how GPT chooses the next token, measures mistakes, and changes its weights.

- In Chapter 11, the final vector becomes logits and next-token probabilities.
- In Chapter 12, predictions become a scalar training signal through cross-entropy and perplexity.
- In Chapter 13, gradients and weight updates connect the loss back to the model parameters.

11 Vocabulary Projection — From Vectors to Words

After `N` transformer blocks, we have a matrix $X_{\text{final}} \in \mathbb{R}^{T \times d}$. The final row, `X_final[T-1]`, is a rich contextualized representation of the last token. The last step is to project this vector into a probability distribution over the vocabulary: which token is most likely to come next?

This projection is called the *language model head*, or *unembedding layer*.

11.1 The Idea

After all the attention and feed-forward layers, we have a vector representing the last token — a list of hundreds of numbers that encodes everything the model knows about what has been said so far. But the user needs a *word*, not a list of numbers.

The final step converts that vector into a probability over the entire vocabulary: for each of the 50,000-odd words the model knows, what is the chance it is the right next word? The word with the highest probability (or a word sampled from the distribution) becomes the model's output.

How does a vector become a probability distribution? The model compares the vector against every word in the vocabulary and assigns a score to each one. High score means "this word fits the context well." Low score means it does not. Then those scores are squeezed into the range 0–1 and made to sum to 1, giving a proper probability.

This step is the mirror image of Chapter 4: embedding turned an integer (a word ID) into a vector; unembedding turns a vector back into a distribution over integers. Many implementations reuse the same table for both directions: the same weights that encode "cat → vector" are transposed to decode "vector → how much does this look like cat?"

11.2 The Math

Given the final hidden state $h_t = X_{\text{final}}[T-1] \in \mathbb{R}^d$:

Step 1 — Final Layer Norm:

$$\tilde{h}_t = \text{LayerNorm}(h_t) \in \mathbb{R}^d$$

Step 2 — Vocabulary Projection:

$$\text{logits} = \tilde{h}_t W_u \in \mathbb{R}^{|V|}$$

With weight tying: $W_u = E^\top$, so:

$$\text{logits}[i] = \tilde{h}_t \cdot E_i$$

The logit for token `i` is the dot product of the model's "prediction vector" with token `i`'s embedding. Tokens whose embeddings align with the prediction vector get high logits.

Step 3 — Softmax → Probabilities:

$$P(\text{next} = i \mid \text{context}) = \text{softmax}(\text{logits})[i] = \frac{\exp(\text{logits}[i])}{\sum_j \exp(\text{logits}[j])}$$

> **Note**
>
> *Math Minute — Temperature*
> We can control the "sharpness" of the distribution with a *temperature parameter* T_{temp}:
>
> $$P(i) = \text{softmax}(\text{logits}/T_{\text{temp}})[i]$$
>
> - $T_{\text{temp}} \to 0$: argmax — always pick the highest-probability token (greedy)
> - $T_{\text{temp}} = 1$: standard softmax
> - $T_{\text{temp}} > 1$: flattens the distribution — more randomness/creativity
>
> Temperature is not a model parameter — it's a sampling hyperparameter set at inference time.

11.3 The Generation Loop

Training and inference use the same forward pass differently.

Training: Given a sequence $[t_1, t_2, \dots, t_n]$, the model predicts all next tokens simultaneously (thanks to causal masking): $P(t_2|t_1), P(t_3|t_1, t_2), \dots$. The loss is cross-entropy averaged over all positions.

Inference (generation):

1. Start with prompt tokens $[t_1, \ldots, t_n]$
2. Forward pass → logits for position n
3. Sample or argmax → t_{n+1}
4. Append t_{n+1} to the sequence
5. Forward pass → logits for position $n + 1$
6. Repeat until stop token or max length

This is called *autoregressive generation*: each new token is fed back in as input to generate the next.

11.4 The Matrix: Worked Example

Let `|V| = 5`, `d = 4`. Final hidden state:

```
h = [0.3, -0.1, 0.8, 0.2]
```

Unembedding matrix $W_u = E^\top$ (columns = token embeddings):

```
logits[0] = h · E[0] = (0.3)(0.10) + (-0.1)(-0.20) + (0.8)(0.30) + (0.2)(-0.40)
          = 0.030 + 0.020 + 0.240 - 0.080 = 0.210

logits[1] = h · E[1] = 0.150 - 0.060 - 0.560 + 0.160 = -0.310
logits[2] = h · E[2] = -0.270 - 0.010 + 0.160 - 0.060 = -0.180
logits[3] = h · E[3] = 0.120 + 0.050 + 0.480 - 0.140 = 0.510
logits[4] = h · E[4] = -0.030 - 0.080 - 0.320 + 0.100 = -0.330

logits = [0.210, -0.310, -0.180, 0.510, -0.330]
```

Softmax:

```
exp(logits) = [1.233, 0.733, 0.835, 1.665, 0.719]
sum         = 5.185

P = [0.238, 0.141, 0.161, 0.321, 0.139]
```

Token 3 has the highest probability (32.1%). That is the top-ranked token for this context.

Figure Figure 11.1 shows the vocabulary projection converting a hidden state into logits over the vocabulary.

Figure Figure 11.2 shows autoregressive generation, where each predicted token is appended and fed back into the model.

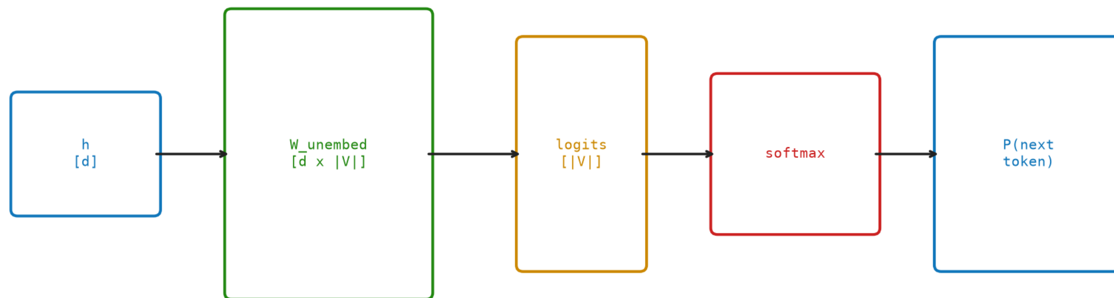

Figure 11.1: Vocabulary projection

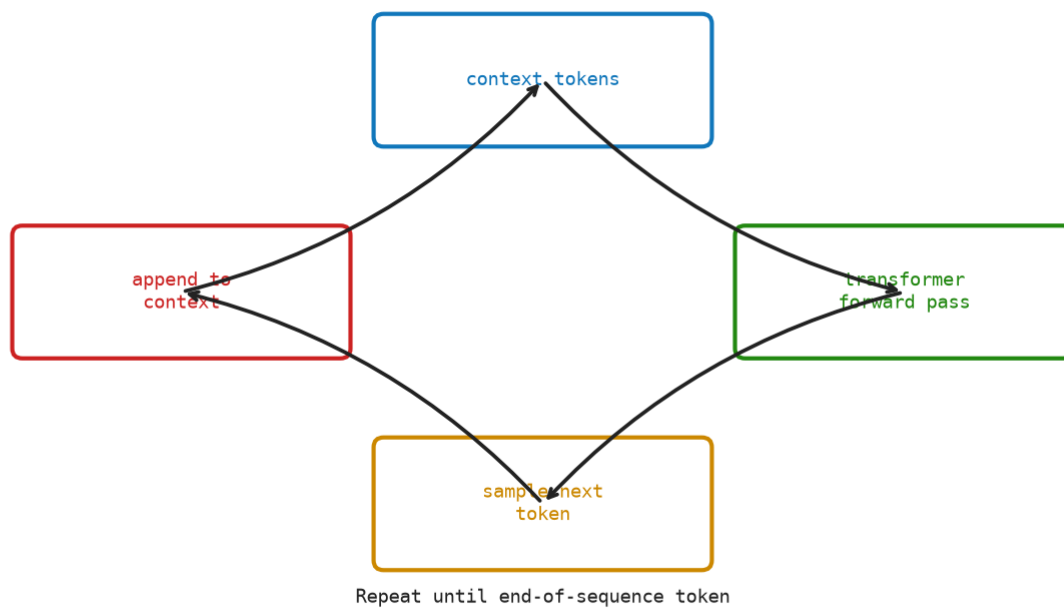

Figure 11.2: Autoregressive generation loop

11.5 The Code: Full microGPT Forward Pass in Python

```
@dataclass(frozen=True)
class GPTConfig:
    vocab_size: int
    d_model: int
    num_heads: int
    num_layers: int
    max_seq_len: int
```

Configuration is data. `GPTConfig` stores the model's five hyperparameters in one immutable object.

```
@dataclass
class GPTParams:
    wte: Matrix
    wpe: Matrix
    blocks: list[TransformerBlock]
    ln_f: LayerNorm


def make_gpt_params(config: GPTConfig, rng: random.Random) -> GPTParams:
    return GPTParams(
        wte=random_matrix(config.vocab_size, config.d_model, rng),
        wpe=random_matrix(config.max_seq_len, config.d_model, rng),
        blocks=[
            make_transformer_block(config.d_model, config.num_heads, rng)
            for _ in range(config.num_layers)
        ],
        ln_f=make_layer_norm(config.d_model),
    )
```

`make_gpt_params` allocates all learnable weights. `wte` is the token embedding matrix $[|V| \times d]$; `wpe` is the position embedding matrix. Weight tying means the unembedding step will reuse `wte` transposed, so no separate parameter is stored.

```
def gpt_forward(token_ids: Sequence[int], params: GPTParams, config: GPTConfig) -> Vector:
    if len(token_ids) > config.max_seq_len:
        raise ValueError("token_ids exceeds max_seq_len")
```

```
    token_embeddings = embed_sequence(params.wte, token_ids)
    position_embeddings = embed_sequence(params.wpe, range(len(token_ids)))
    x = matrix_add(token_embeddings, position_embeddings)
    x = forward_stack(x, params.blocks)
    h = layer_norm_vector(x[-1], params.ln_f)
    return [dot(h, token_vector) for token_vector in params.wte]
```

`gpt_forward` is the full pipeline: embed token IDs, add position embeddings, pass through N transformer blocks, apply the final layer norm to the last token's vector, then compute dot products with every row of `wte`.

```
def temperature_scale(logits: Sequence[float], temperature: float) -> list[float]:
    return [value / temperature for value in logits]


def top_k_filter(logits: Sequence[float], k: int) -> list[float]:
    keep = {index for index, _value in sorted(enumerate(logits), key=lambda item: item[1], r
    return [value if index in keep else -1.0e9 for index, value in enumerate(logits)]


def sample_token(probabilities: Sequence[float], rng: random.Random) -> int:
    threshold = rng.random()
    cumulative = 0.0
    for index, probability in enumerate(probabilities):
        cumulative += probability
        if threshold <= cumulative:
            return index
    return len(probabilities) - 1


def generate(
    params: GPTParams,
    config: GPTConfig,
    start_ids: Sequence[int],
    max_new_tokens: int,
    temperature: float,
    top_k: int,
    rng: random.Random,
) -> list[int]:
    tokens = list(start_ids)
    generated = []
    for _ in range(max_new_tokens):
```

```
        logits = gpt_forward(tokens, params, config)
        probabilities = softmax(top_k_filter(temperature_scale(logits, temperature), top_k))
        next_token = sample_token(probabilities, rng)
        tokens.append(next_token)
        generated.append(next_token)
    return generated
```

`generate` is the autoregressive loop. It repeatedly runs the model, scales and filters the logits, samples one next token, appends it to the context, and continues.

Run with `python3 src/python/chapter_demos.py`.

11.6 Weight Tying in Detail

Why does weight tying work so well?

- The embedding `E[i]` is learned to represent token `i` such that tokens that appear in similar contexts have similar embeddings.
- The unembedding $\text{logits}[i] = h \cdot E[i]$ measures the alignment between the model's prediction vector `h` and token `i`'s embedding.
- If `h` is pointing in the direction of "tokens that follow this context," and `E[i]` represents token `i`'s meaning, then tokens that are semantically appropriate will naturally score higher.

Weight tying forces the model to use a single geometric space for both "what a token means" and "how to predict a token," a constraint that also regularizes learning.

11.7 Key Takeaways

- The *unembedding* projects the final hidden state to logits: $\text{logits} = hW_u \in \mathbb{R}^{|V|}$.
- *Weight tying* ($W_u = E^\top$) reuses the embedding matrix and reduces parameters.
- *Softmax* converts logits to a probability distribution.
- *Temperature* controls sampling sharpness; *top-K* restricts the candidate set.
- *Autoregressive generation*: sample one token at a time, feeding each back as input.

11.8 The Complete Picture

You have now seen every piece:

```
Input text
   ↓ tokenizer
Token IDs: [3, 7, 12, 5, …]
   ↓ embedding matrix E
Token embeddings: [T × d]
   ↓ + positional encoding
X̃ = X + PE: [T × d]
   ↓ transformer block × N
      LayerNorm
      Multi-Head Attention (Q, K, V projections + causal mask + softmax + weighted sum)
      Residual
      LayerNorm
      Feed-Forward Network (expand → GELU → contract)
      Residual
   ↓
X_final: [T × d]
   ↓ final LayerNorm + unembedding
Logits: [|V|]
   ↓ softmax (+ temperature + top-K)
P(next token): [|V|]  →  sample  →  next token ID
```

Every box in that diagram corresponds to one chapter of this book.

12 Loss — How the Model Learns

So far we have traced the forward pass from end to end. But none of that knowledge was conjured from thin air. The model *learned* by making predictions, measuring how wrong they were, and nudging every weight in the right direction — billions of times.

This chapter is about that measurement: the *loss function*.

12.1 The Idea

The model reads a sentence and, at each position, guesses what word comes next. Those guesses are never perfect — but how do we measure *how wrong* a guess is?

We need a single number that captures "wrongness" in a way that is small when the model is doing well and large when it is doing badly. That number is the *loss*. And crucially, it needs to be computable from the model's output so we can use it to improve the model.

Here is the key insight: the model already outputs a probability for every word in the vocabulary. If the true next word is "cat" and the model assigns a 90% probability to "cat," that is a good prediction — loss should be low. If the model assigns only 1% probability to "cat" (spreading the rest across unrelated words), that is a bad prediction — loss should be high.

The *cross-entropy loss* formalizes this: it measures how much probability the model assigned to the *correct* answer. A perfect model assigns 100% to the right word — loss is zero. A confused model spreads probability thinly — loss is high.

12.2 The Problem: Evaluating a Probability Distribution

After the forward pass, the model produces a probability distribution over the vocabulary:

$$P(\cdot \mid t_1, \dots, t_t) \in \mathbb{R}^{|V|}$$

We know the true next token t_{t+1} (because it comes from the training text). The question is: *how do we turn "how wrong is this distribution" into a single differentiable number?*

The answer is *cross-entropy loss*.

12.3 Cross-Entropy Loss

12.3.1 The Formula

For a single prediction at position t:

$$\mathcal{L}(t) = -\log P(t_{t+1} \mid t_1, \dots, t_t)$$

That's it. Negative log of the probability assigned to the *true* next token.

12.3.2 Why Negative Log?

The model outputs a probability $p \in (0, 1]$. We want a loss that is zero when the prediction is perfect ($p = 1$), large when the model is wrong ($p \to 0$), and differentiable everywhere.

$-\log(p)$ delivers all three:

Predicted probability p	$-\log(p)$	Interpretation
1.00	0.00	Perfect
0.80	0.22	Pretty good
0.50	0.69	Random guess
0.10	2.30	Mostly wrong
0.01	4.61	Very wrong
0.001	6.91	Catastrophically wrong

Suppose the vocabulary has four tokens and the model assigns probabilities $[0.1, 0.6, 0.2, 0.1]$. If the correct token index is 1, then the probability of the correct answer is 0.6. The loss is:

$$\mathcal{L} = -\log(0.6) \approx 0.511$$

If the model assigned probability 1.0 to the correct token, the loss would be 0.0. If it assigned 0.01, the loss would jump to 4.605. Low confidence in the correct answer is expensive, as Figure 12.1 shows.

> **Note**
>
> *Math Minute — Logarithms*
> $\log(p)$ for $p \in (0, 1]$ is always ≤ 0. It passes through $(1, 0)$ and tends to $-\infty$ as $p \to 0$. Because we negate it, $-\log(p) \geq 0$: zero means perfect, large means wrong. All modern ML uses natural log (base e), so loss is measured in *nats*. Using base 2 gives *bits*.

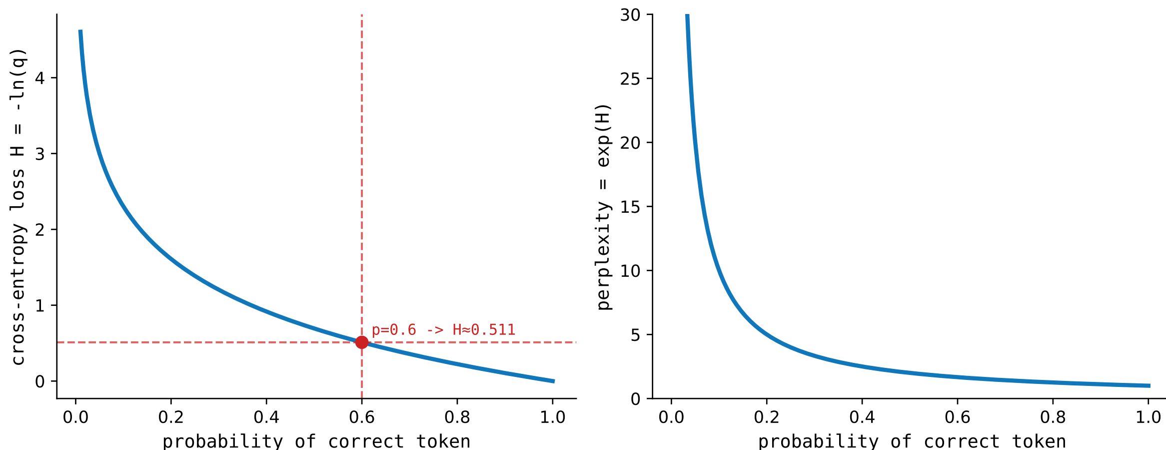

Figure 12.1: Cross-entropy loss grows steeply as the model assigns less probability to the correct token

12.3.3 The Full Training Loss

A single document of length T contributes:

$$\mathcal{L}_{\text{doc}} = \frac{1}{T} \sum_{t=1}^{T} -\log P(t_{t+1} \mid t_1, \ldots, t_t)$$

12.3.4 Perplexity

Loss as nats is hard to interpret intuitively. *Perplexity* converts it to something more concrete:

$$\text{PPL} = e^{\mathcal{L}}$$

Perplexity is the *effective branching factor*: on average, how many equally likely choices does the model think there are at each step?

Loss $\mathcal{L}$	Perplexity	Interpretation
0.00	1.0	Perfect — only one plausible token
0.69	2.0	Two equally likely tokens
2.30	10.0	Ten equally likely tokens
4.61	100	Random guess over 100 tokens
6.91	1000	Very confused

GPT-2 (1.5B) reached ~18 perplexity on WikiText-103. GPT-4 is estimated well below 5 on standard benchmarks. Early training starts around 100–1000; loss curves falling to perplexity ~10–20 signals the model has learned real language structure.

> **Note**
>
> *Math Minute — Why Exponential?*
> If a model assigns uniform probability $1/k$ to each of k options, the cross-entropy is $-\log(1/k) = \log k$. Exponentiating: $e^{\log k} = k$. So perplexity of k means the model behaves like a uniform distribution over k choices. Perplexity 1 = model is certain. Perplexity $|V|$ 50,000 = model knows nothing.

12.3.5 Teacher Forcing

Notice the formula conditions on *true* past tokens, not the model's *own predictions*. This is called *teacher forcing*: during training, we always feed the ground-truth context.

This makes training stable, parallelizable, and simple. The causal mask from Chapter 6 is what enables this — position t can only attend to positions $1 \ldots t$, so the model's prediction at position t is causally correct even when processing all positions in parallel.

12.3.6 Cross-Entropy as Information Theory

Cross-entropy comes from information theory. The *cross-entropy* between true q and predicted p is:

$$H(q, p) = -\sum_j q_j \log p_j$$

When q is one-hot, only the true token's term survives: $H(q, p) = -\log p_{t_{t+1}}$. Exactly our loss.

Minimizing cross-entropy is equivalent to minimizing the KL divergence between the model's predictions and the true data distribution. This is why cross-entropy is the natural loss for language models.

12.4 The Matrix: Worked Example

Trace the loss for a 4-token sequence `["The", "cat", "sat", "on"]` with $|V| = 5$ (toy vocabulary).

12.4.1 Forward Pass Outputs

The model produces logits at each position. After softmax:

```
Position 0 (predicting token after "The"):
  P = [0.05, 0.10, 0.60, 0.20, 0.05]   ← true next = token 3
  P(true) = 0.20,  loss = -log(0.20) = 1.609

Position 1 (predicting after "The cat"):
  P = [0.10, 0.05, 0.15, 0.10, 0.60]   ← true next = token 4
  P(true) = 0.60,  loss = -log(0.60) = 0.511

Position 2 (predicting after "The cat sat"):
  P = [0.40, 0.20, 0.15, 0.15, 0.10]   ← true next = token 0
  P(true) = 0.40,  loss = -log(0.40) = 0.916

Position 3 (predicting after "The cat sat on"):
  P = [0.05, 0.05, 0.80, 0.05, 0.05]   ← true next = token 2
  P(true) = 0.80,  loss = -log(0.80) = 0.223
```

12.4.2 Total Loss

$$\mathcal{L} = \frac{1}{4}(1.609 + 0.511 + 0.916 + 0.223) = 0.815$$

$$\text{PPL} = e^{0.815} \approx 2.26$$

The model is effectively choosing between about 2.3 equally likely tokens on average.

Figure Figure 12.2 shows cross-entropy loss as softmax probability followed by the negative log of the true-token probability.

12.5 Python Implementation

```
def cross_entropy_loss(logits: Sequence[float], true_id: int) -> float:
    probabilities = softmax(logits)
    return -math.log(max(probabilities[true_id], 1.0e-12))
```

`cross_entropy_loss` computes $-\log P(\text{true token})$ for a single position.

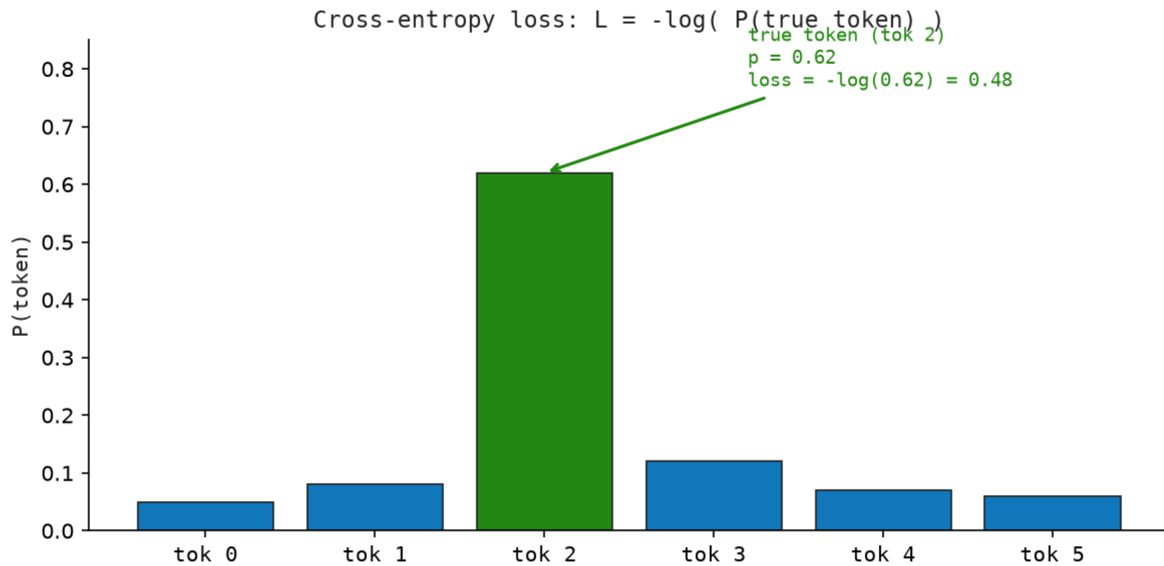

Figure 12.2: Cross-entropy loss

```
def sequence_loss(logits_list: Sequence[Sequence[float]], true_ids: Sequence[int]) -> float:
    losses = [cross_entropy_loss(logits, true_id) for logits, true_id in zip(logits_list, tru
    return sum(losses) / len(losses)
```

`sequence_loss` averages the per-position losses into one scalar: the training signal for a complete sequence.

```
def perplexity(loss: float) -> float:
    return math.exp(loss)
```

`perplexity` converts the mean loss back to an interpretable scale.

```
def softmax_cross_entropy_grad(logits: Sequence[float], true_id: int) -> Vector:
    grad = softmax(logits)
    grad[true_id] -= 1.0
    return grad
```

`softmax_cross_entropy_grad` implements the closed-form gradient: $\partial\mathcal{L}/\partial z_j = P(j) - \mathbf{1}[j = \text{true}]$.

Run with `python3 src/python/chapter_demos.py`. Expected output:

```
Per-position losses:
  pos 0 (true=3): loss=1.609  P(true)=0.200
  pos 1 (true=4): loss=0.511  P(true)=0.600
  pos 2 (true=0): loss=0.916  P(true)=0.400
  pos 3 (true=2): loss=0.223  P(true)=0.800

Mean loss: 0.815
Perplexity: 2.26
```

12.6 Key Takeaways

- *Cross-entropy loss* is $-\log P(\text{true next token})$. Zero when perfect, unbounded when wrong.
- *Teacher forcing* feeds ground-truth context at training time, enabling full parallelism via causal masking.
- *Perplexity* $= e^{\mathcal{L}}$ is the effective branching factor — more intuitive than raw nats.
- *The softmax-CE gradient* is $P(j) - \mathbf{1}[j = t_{t+1}]$: a closed-form, numerically stable backprop step.
- *Cross-entropy* = *KL divergence* (since labels are one-hot), so training minimizes the KL from model predictions to the true data distribution.
- The loss is the *single* signal that shapes every weight: embeddings, attention projections, FFN weights, layer norms — all updated from this one number per token.

Note

What's next? Chapter 12 gave us a loss: a single number measuring how wrong the model is. But knowing *how wrong* we are is not enough. We also need to know *which weights* are responsible for the error, and *by how much to change each one*. That is the job of *training* — Chapter 13.

13 Training — Teaching the Model

Chapter 12 gave us a loss: a single number measuring how wrong the model is. But knowing *how wrong* we are is only half the battle. We also need to know *which weights* are responsible for the error, and *by how much to change each one*.

That is the job of *training*.

13.1 The Idea

The loss tells us *how wrong* the model was at its last prediction. Training uses that wrongness to *improve* the model, so it does better next time.

Here is the core loop, in plain terms:

1. *Forward pass*: feed a sentence into the model, let it predict the next word at every position, measure how wrong those predictions were (the loss).
2. *Backward pass*: trace back through every calculation the model just made and figure out, for each weight, whether increasing it would have made the loss higher or lower, and by how much. This produces a *gradient* for every weight.
3. *Update*: nudge each weight slightly in the direction that reduces the loss. Weights that caused big errors get bigger nudges; weights that barely mattered get smaller nudges.
4. *Repeat*: billions of times, across trillions of words.

That is the entire training algorithm.

13.2 The Goal: Move the Loss Down

The model has millions of parameters. Call them collectively θ. The loss $\mathcal{L}(\theta)$ is a function of all of them. We want to find θ that minimizes the loss.

The strategy is *gradient descent*: repeatedly take a small step downhill on the loss surface.

$$\theta \leftarrow \theta - \eta \cdot \nabla_\theta \mathcal{L}$$

$\eta > 0$ is the *learning rate*: how big a step to take. $\nabla_\theta \mathcal{L}$ is the *gradient*: a collection of partial derivatives, one per parameter.

> **Note**
>
> *Math Minute — Partial Derivatives*
> A partial derivative $\frac{\partial \mathcal{L}}{\partial w}$ answers: *if we increase w by a tiny ϵ while holding every other weight fixed, by how much does $\mathcal{L}$ change?*
> A large positive value means w is pushing loss up; decrease it. A negative value means the opposite. Zero means w is momentarily irrelevant to the loss.
> Notation: ∂ (curly d) instead of d signals "partial"; other variables are held constant.

13.3 The Chain Rule

Backpropagation is the chain rule from calculus, applied to a computation graph.

Suppose the loss depends on weight w through a chain of intermediate values:

$$w \xrightarrow{f} z \xrightarrow{g} \mathcal{L}$$

The chain rule says:

$$\frac{d\mathcal{L}}{dw} = \frac{d\mathcal{L}}{dz} \cdot \frac{dz}{dw}$$

If we know how $\mathcal{L}$ varies with z, and how z varies with w, we multiply to get how $\mathcal{L}$ varies with w.

In a transformer, the chain has hundreds of layers. Backprop starts at the loss and works backward, multiplying local derivatives as it goes. Each layer only needs two things: the gradient arriving from the layer above (called δ, "delta"), and its own local derivative.

13.3.1 Step 1: A gradient update for one weight

```python
def sgd_update_scalar(weight: float, gradient: float, learning_rate: float) -> float:
    return weight - learning_rate * gradient
```

If the gradient is positive (loss rises when w increases), we subtract — decreasing w. If negative, we add. The learning rate η scales how large the step is.

This is the entire idea. Everything that follows is this rule, applied to millions of weights simultaneously.

13.4 Backprop Through Softmax and Cross-Entropy

Chapter 12 stated the softmax-CE gradient without proof:

$$\frac{\partial \mathcal{L}}{\partial z_j} = P(j) - \mathbb{1}[j = t]$$

where z_j are the logits and t is the true token.

For the true token $j = t$: gradient is $P_t - 1$ (negative, so we push z_t up). For every other token: gradient is P_j (positive, so we push those logits down). The model is nudged to be more confident about the correct answer.

```
def softmax_cross_entropy_grad(logits: Sequence[float], true_id: int) -> Vector:
    grad = softmax(logits)
    grad[true_id] -= 1.0
    return grad
```

`softmax_cross_entropy_grad` starts with the predicted probabilities, then subtracts 1 from the correct token. That gives the gradient signal that pushes the correct logit up and the incorrect logits down.

13.5 Backprop Through a Linear Layer

The most common operation in a transformer is a *linear layer*: $y = Wx$.

Given the upstream gradient $\delta = \frac{\partial \mathcal{L}}{\partial y}$ (arriving from the layer above), the chain rule gives:

$$\frac{\partial \mathcal{L}}{\partial W} = \delta \cdot x^\top \qquad \frac{\partial \mathcal{L}}{\partial x} = W^\top \cdot \delta$$

The first equation tells us how to update W. The second passes the gradient backward to whatever fed x into this layer.

13.5.1 Step 2: Linear layer backward pass

```
def linear_backward(delta: Vector, weights: Matrix, x: Vector) -> tuple[Matrix, Vector]:
    grad_w = [[delta_i * x_j for x_j in x] for delta_i in delta]
    grad_x = [sum(weights[i][j] * delta[i] for i in range(len(delta))) for j in range(len(x)
    return grad_w, grad_x
```

`linear_backward` returns both gradients needed by backpropagation. `grad_w` updates the linear layer's weights, while `grad_x` passes the gradient back to the layer that produced `x`.

13.6 Accumulating Gradients Across the Sequence

We compute the softmax-CE gradient independently at each position t. But all positions share the same unembedding matrix W_u (and the same transformer weights). Their gradient contributions must be *summed* before updating any weight.

13.6.1 Step 4: Summing gradient contributions

```
def accumulate_gradients(gradients: Sequence[Matrix]) -> Matrix:
    rows, cols = shape(gradients[0])
    total = make_matrix(rows, cols)
    for gradient in gradients:
        total = matrix_add(total, gradient)
    return total
```

The more a weight influenced the output at many positions, the larger (and more reliable) its accumulated gradient.

13.7 The Gradient Descent Step

With gradients accumulated for every parameter, the update rule is:

$$W \leftarrow W - \eta \cdot \frac{\partial \mathcal{L}}{\partial W}$$

13.7.1 Step 5: Updating a weight matrix in place

```
def sgd_update_matrix(weights: Matrix, gradient: Matrix, learning_rate: float) -> None:
    for i, row in enumerate(weights):
        for j, value in enumerate(row):
            row[j] = value - learning_rate * gradient[i][j]
```

Every entry W_{ij} moves a small step in the direction that reduces the loss.

13.8 One Training Step

Now we can assemble the full cycle: forward pass to get predictions and loss, backward pass to get gradients, update pass to improve every weight.

13.9 Watching the Loss Fall

Running thousands of training steps, the loss curve looks roughly like this:

Step	Loss	Perplexity	Interpretation
0	10.82	50,000	Random — model knows nothing
100	6.91	1,000	Ruling out most tokens
1,000	4.61	100	Learned basic frequency
10,000	2.30	10	Has rough grammar
100,000	0.92	2.5	Strong language model

The curve drops steeply at first (obvious mistakes are easy to fix) then more slowly and noisily (subtle patterns are harder and data points disagree).

> **Note**
>
> *Why does loss get noisy later?*
> Early on, large gradients correct glaring errors. Later, gradients are smaller and point in slightly different directions for different training examples. The noise is not a bug: random variation in gradient direction helps the model escape poor local minima. This is the *stochastic* in *stochastic gradient descent* (SGD).

Figure 13.1 shows gradients flowing backward from the loss through each layer while weights are updated.

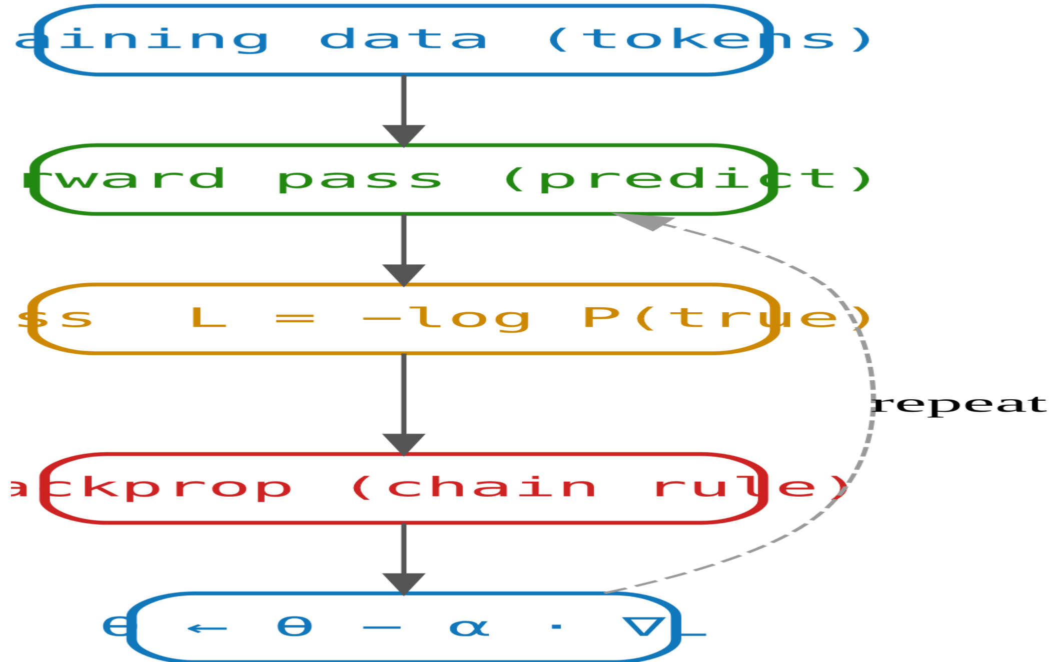

Figure 13.1: Backpropagation: gradient flowing from loss backward through each layer, updating weights.

13.10 Key Takeaways

- *Gradient descent* moves every weight downhill: $\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L}$.
- *Partial derivatives* measure the sensitivity of the loss to each individual weight, holding all others fixed.
- *Backpropagation* is the chain rule applied layer by layer from loss to inputs. Each layer only needs the upstream gradient δ and its own local derivative; no global information required.
- *Linear layer backward:* $\frac{\partial \mathcal{L}}{\partial W} = \delta x^\top$, $\frac{\partial \mathcal{L}}{\partial x} = W^\top \delta$. The same two matrices as the forward pass, just transposed and multiplied in a different order.
- *Softmax-CE gradient* is $P(j) - \mathbb{1}[j = t]$: the model's predicted probability minus the true label. Nearly zero when correct and confident; large when wrong.
- *Shared weights accumulate gradients* from every position in the sequence before any update is applied.
- The six-step cycle (forward, loss, backward, accumulate, update, repeat) is all of training. Every weight in the model, over billions of tokens, updated by exactly this loop.

Part V

Modern Extensions

This part connects the baseline transformer to modern GPT implementations. It keeps the simple model from earlier chapters as the reference point, then shows which production refinements change speed, memory use, context length, or architecture.

- In Chapter 14, you will see rotary position embeddings, grouped-query attention, FlashAttention, and major architectural variants.

14 Modern GPT

The previous chapters built a complete GPT from scratch. Real production models add several refinements that improve efficiency, context length, and capability. This chapter surveys the most important innovations beyond the baseline transformer.

14.1 KV Cache

During inference, GPT generates one token at a time. At each step, the model runs a full forward pass — including attention over every previous token.

Without any optimization, this is wasteful: the key and value matrices for tokens $1, \dots, t-1$ are identical to what was computed in the previous step. The *KV cache* (Key-Value cache) stores those matrices and reuses them.

At step t, only the new token's query, key, and value are computed fresh:

$$q_t = x_t W_q, \quad k_t = x_t W_k, \quad v_t = x_t W_v$$

Then k_t and v_t are appended to the cache, and attention is computed using the full cached K and V:

$$\text{head}_t = \text{softmax}\left(\frac{q_t\, K_{\text{cache}}^\top}{\sqrt{d_k}} + M_t\right) V_{\text{cache}}$$

The result is that each new token costs $O(N)$ compute rather than $O(N^2)$ — a large saving for long sequences.

The trade-off is memory. For a model with L layers, H heads, head dimension d_k, and sequence length N, the cache holds:

$$2 \times L \times H \times N \times d_k \text{ values}$$

(factor of 2 for K and V; values are stored per layer, per head).

The KV cache is present in every production transformer inference engine. Grouped-Query Attention (next section) directly addresses the memory cost of the cache.

14.2 Multi-Query and Grouped-Query Attention

Standard multi-head attention (Chapter 8) creates separate Q, K, V projections for every head. The key and value matrices dominate GPU memory during inference.

Multi-Query Attention (MQA) (Shazeer, 2019) uses a single K and V shared across all heads, reducing the KV cache by a factor of H (number of heads).

Grouped-Query Attention (GQA) (Ainslie et al., 2023) is a middle ground: G groups of heads share a single K, V, where $1 \leq G \leq H$. Setting $G = 1$ recovers MQA; setting $G = H$ recovers standard MHA.

Method	KV heads	Memory	Quality
MHA	H	1x	Baseline
GQA	H/G	1/G x	Near-MHA
MQA	1	1/H x	Slight drop

GQA is used in LLaMA 3, Mistral, and Gemma. It allows inference to fit in less memory without significant quality degradation.

14.3 Flash Attention

Standard self-attention computes the full $n \times n$ attention matrix explicitly:

$$\text{Attn} = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V$$

For a sequence of length n, this requires $O(n^2)$ memory — prohibitive for long contexts.

Flash Attention (Dao et al., 2022) reorders the computation using *tiling*: it processes blocks of the sequence in GPU SRAM (fast on-chip memory) and never materialises the full $n \times n$ matrix in HBM (slow off-chip memory).

The result is mathematically identical to standard attention, but:

- Memory: $O(n)$ instead of $O(n^2)$.
- Speed: 2–4× faster than PyTorch's built-in attention in practice.
- Enables context windows of 32k–128k tokens at practical batch sizes.

Flash Attention 2 and 3 added further improvements (better parallelism, support for GQA). It is now the default in all major frameworks.

> **Note**
>
> Flash Attention is a hardware-aware algorithm, not a new mathematical operation. The inputs and outputs are identical to standard attention.

14.4 Alternative Architectures

The transformer is not the only architecture for sequence modelling. Several alternatives challenge or complement it.

14.4.1 Mamba (State Space Models)

Mamba (Gu & Dao, 2023) is based on *selective state space models* (SSMs). Instead of attending to all past tokens, it maintains a compressed hidden state that is updated recurrently (like an RNN), but with a selective mechanism that decides what to remember.

Key properties:

- Linear time in sequence length ($O(n)$ vs $O(n^2)$ for attention).
- No attention matrix — the context is compressed into a fixed-size state.
- Competitive with transformers on language modelling benchmarks.
- Less effective at in-context retrieval tasks (where attention excels).

14.4.2 Diffusion Language Models (DiffusionGemma)

Autoregressive models like GPT generate text *left-to-right*, one token at a time. *Diffusion language models* (DLMs) generate all tokens in parallel by iterative denoising.

MDLM and *Gemma-based diffusion models* start from a fully masked sequence and refine it over T denoising steps. Each step predicts a cleaner version of the entire sequence.

Advantages: - Can revise earlier tokens after seeing later context. - Faster generation at inference time (fewer sequential steps than token length). - Quality still trails autoregressive models as of 2025.

14.4.3 Mixture of Experts (MoE)

Mixture of Experts replaces the single feed-forward network in each transformer block with E expert networks and a *router* that activates only K of them per token.

- Mistral 8x7B and GPT-4 are believed to use MoE.
- Total parameters can be large while compute per token stays small.
- Trade-off: routing instability, expert load imbalance.

14.5 Key Takeaways

Innovation	Benefit
RoPE	Relative position encoding, better length generalisation
GQA / MQA	Smaller KV cache, faster and cheaper inference
Flash Attention	Linear memory, 2-4x faster attention, long context
Mamba (SSM)	Linear-time sequence modelling, no attention matrix
Diffusion LMs	Parallel generation, bidirectional revision
MoE	Larger model capacity without proportional compute cost

The field moves fast. Each of these innovations addresses a concrete bottleneck — memory, speed, context length, or capability — that became critical as models scaled. Understanding the baseline GPT from Chapter 3 through Chapter 13 makes every one of these extensions legible.

A microGPT in Python — Complete Runnable Code

This appendix assembles every Python snippet from the book into three files under `src/python/`, with all dependencies resolved. The implementation uses only the Python standard library.

A.1 Files

- `common.py` — forward-pass building blocks (embeddings, attention, FFN, transformer, GPT forward)
- `train.py` — loss functions and gradient descent (cross-entropy, backprop, SGD)
- `inference.py` — generation and end-to-end demo

A.2 Installation & Running

```
# Run the full demo
python3 src/python/inference.py

# Run every chapter's end-to-end code check
python3 src/python/run_book_code.py
```

A.3 End-to-End Demo

```python
def demo(seed: int = 7) -> dict[str, object]:
    config = GPTConfig(vocab_size=100, d_model=32, num_heads=4, num_layers=2, max_seq_len=64
    model_rng = random.Random(seed)
    sample_rng = random.Random(seed + 1)
    params = make_gpt_params(config, model_rng)
    logits = gpt_forward([1, 2, 3, 4, 5], params, config)
```

```
    probabilities = softmax(logits)
    top_token = max(range(len(probabilities)), key=probabilities.__getitem__)
    generated = generate(params, config, [1, 2, 3], 10, temperature=0.8, top_k=10, rng=sampl
    return {
        "config": config,
        "parameters": count_parameters(config),
        "logits_first_10": logits[:10],
        "top_token": top_token,
        "top_probability": probabilities[top_token],
        "generated": generated,
        "full_sequence": [1, 2, 3] + generated,
    }
```

A.4 microGPT Architecture Summary (Reference)

Component	Input	Output	Parameters
Token Embedding	`[T]` ints	$[T \times d]$	$\|V\| \times d$
Positional Embedding	`[T]` positions	$[T \times d]$	$T_{max} \times d$
$\times$ N Transformer Blocks			
LayerNorm 1	$[T \times d]$	$[T \times d]$	`2d`
Multi-Head Attn	$[T \times d]$	$[T \times d]$	$4d^2$
Residual	$[T \times d]$	$[T \times d]$	0
LayerNorm 2	$[T \times d]$	$[T \times d]$	`2d`
FFN	$[T \times d]$	$[T \times d]$	$8d^2$
Residual	$[T \times d]$	$[T \times d]$	0
Final LayerNorm	$[T \times d]$	$[T \times d]$	`2d`
Unembedding	`[d]`	$[\|V\|]$	$d \times \|V\|$ (tied)

Total parameters: $2|V|d + T_{max} \cdot d + N(12d^2 + 4d) + 2d$

For GPT-2 small: `d=768, N=12, |V|=50257, T_max=1024` → ~117M params.