Vectors of meaning

A mathematical journey into transformers

Valentin Radu

2026-02-05

Table of contents

Preface

This book is a mathematical journey into the architecture that powers modern language models. We start with the basics: what is a vector? what does a derivative measure? These simple questions become the foundation for increasingly sophisticated machinery. By the final chapters, we derive complete transformer equations with explicit forward and backward propagation, every gradient computed, every dimension specified. The progression is exponential: early chapters move slowly through fundamentals, later chapters synthesize everything into a unified mathematical framework.

What this book is

Vectors of meaning takes a math-first approach. Every concept is derived rigorously, with explicit dimensions, indices, and gradient computations. We show every step: no operation is left as an exercise, no mechanism assumed familiar. By the final chapter, a reader will have the complete mathematical specification to implement a transformer from scratch.

The book progresses in three parts:

Part I: Foundations establishes the mathematical prerequisites. We cover linear algebra (vectors, matrices, eigenvalues, and the geometric intuition behind them), calculus (derivatives, the chain rule, and backpropagation), probability (distributions, expectations, and information theory), notation conventions, neural network basics (neurons, layers, loss functions, gradient descent), and sequence modeling challenges (why recurrent networks struggle and what transformers solve).

Part II: Building blocks develops the components that make transformers work. We explore embeddings (how discrete tokens become continuous vectors), the attention mechanism (weighted combinations based on relevance), self-attention (tokens attending to each other), multi-head attention (parallel attention with different perspectives), and positional encoding (injecting sequence order into permutation-invariant attention).

Part III: The transformer architecture brings everything together. We derive the complete encoder-decoder transformer with full forward and backward propagation, examine training objectives (language modeling, masked language modeling, RLHF), and explore scaling laws (the mathematical relationships between compute, data, parameters, and performance).

Who this book is for

This book is for readers who want to understand transformers through their actual mathematics. It's for students and researchers building rigorous foundations, engineers moving beyond API calls to genuine understanding, and anyone curious about how language models work at a fundamental level.

If you've read papers and found yourself lost in notation, or watched explanations that glossed over the details, this book provides the complete formal treatment.

Prerequisites

We assume familiarity with:

- **Linear algebra**: vectors, matrices, matrix multiplication
- **Calculus**: derivatives, partial derivatives, the chain rule
- **Probability**: random variables, distributions, expectations

We review these topics in the foundations section, but they shouldn't be entirely new. Some comfort with mathematical notation helps, though we define everything we use.

How to read this book

Each chapter builds on previous ones. The foundations establish tools we use throughout; the building blocks are assembled into the complete architecture. Reading linearly works best, though readers comfortable with the prerequisites might skim Part I.

Mathematical derivations include explicit dimensions and worked examples. We prioritize clarity over brevity: if a step isn't obvious, we show it. Code appears sparingly and only where it genuinely clarifies a concept that mathematics alone cannot.

The book uses a consistent notation system detailed in Chapter 4. Bold uppercase ($\mathbf{W}$) denotes matrices, bold lowercase ($\mathbf{x}$) denotes vectors, and regular font denotes scalars. Derivatives use Leibniz notation ($\frac{df}{dx}$) throughout.

We begin with the mathematics that underlies everything: linear algebra.

Citation

If you use this book in your work, please cite it as:

> Radu, V. (2024). *Vectors of meaning: A mathematical journey into transformers*. https://vom.radval.me. DOI: 10.5281/zenodo.18490032

```bibtex
@book{radu2024vectors,
  title     = {Vectors of meaning: A mathematical journey into transformers},
  author    = {Radu, Valentin},
  year      = {2024},
  url       = {https://vom.radval.me},
  doi       = {10.5281/zenodo.18490032}
}
```

Part I

Foundations

Chapter 1

Linear algebra essentials

Learning objectives

After completing this chapter, you will be able to:

- Define vectors, vector spaces, and linear independence
- Compute dot products and interpret them geometrically as similarity measures
- Perform matrix multiplication and understand it as a linear transformation
- Calculate eigenvalues and eigenvectors, and understand their geometric meaning
- Apply these concepts to understand how neural networks transform data

Before we dive into transformers, we need to establish a solid mathematical foundation. Linear algebra is the language of neural networks—every operation in a transformer can be understood as a matrix operation. This chapter reviews the essential concepts we'll use throughout the book.

1.1 Vectors and vector spaces

A vector is an ordered list of numbers. We write vectors as column matrices and denote them with lowercase bold letters:

$$\mathbf{v} = [v_1 \; v_2 \; \vdots \; v_n]$$

The set of all n-dimensional real vectors forms a vector space $\mathbb{R}^n$. A vector space must satisfy certain properties. If $\mathbf{u}, \mathbf{v}, \mathbf{w}$ are vectors and a, b are scalars, then:

1. **Closure under addition**: $\mathbf{u} + \mathbf{v}$ is also in the space
2. **Closure under scalar multiplication**: $a\mathbf{v}$ is also in the space
3. **Associativity**: $(\mathbf{u} + \mathbf{v}) + \mathbf{w} = \mathbf{u} + (\mathbf{v} + \mathbf{w})$
4. **Commutativity**: $\mathbf{u} + \mathbf{v} = \mathbf{v} + \mathbf{u}$
5. **Identity**: There exists a zero vector $\mathbf{0}$ such that $\mathbf{v} + \mathbf{0} = \mathbf{v}$
6. **Inverse**: For every $\mathbf{v}$ there exists $-\mathbf{v}$ such that $\mathbf{v} + (-\mathbf{v}) = \mathbf{0}$
7. **Distributivity**: $a(\mathbf{u} + \mathbf{v}) = a\mathbf{u} + a\mathbf{v}$ and $(a + b)\mathbf{v} = a\mathbf{v} + b\mathbf{v}$
8. **Scalar multiplication associativity**: $a(b\mathbf{v}) = (ab)\mathbf{v}$
9. **Scalar identity**: $1\mathbf{v} = \mathbf{v}$

Why do we care about these properties? Because they guarantee we can perform algebraic manipulations safely. In transformers, we'll constantly be adding vectors (combining information) and scaling them (adjusting magnitudes), so we need these operations to behave predictably.

Before we continue, let's clarify what "linear" means. We'll encounter this word everywhere: linear combinations, linear independence, linear transformations. The term "linear" captures a fundamental idea: operations that respect scaling and addition. Specifically, a function or operation f is linear if it satisfies two properties:

1. **Scaling**: $f(a\mathbf{v}) = af(\mathbf{v})$ for any scalar a
2. **Addition**: $f(\mathbf{u} + \mathbf{v}) = f(\mathbf{u}) + f(\mathbf{v})$

These can be combined into one property: $f(a\mathbf{u} + b\mathbf{v}) = af(\mathbf{u}) + bf(\mathbf{v})$. Linear operations are simple in a precise sense. They don't have interactions or nonlinear terms. If you double the input, you double the output. If you add two inputs, you can process them separately and add the results. This makes linear operations tractable to analyze mathematically, which is why we study them first. Of course, transformers are not purely linear (otherwise they'd be very limited), but understanding the linear parts is essential before we add nonlinearity.

What violates linearity? Consider $f(x) = x^2$. This fails scaling: $f(2x) = (2x)^2 = 4x^2$, but $2f(x) = 2x^2$. The squaring creates an extra factor. Or consider $f(x) = x + 1$. This fails the zero test: a linear function must map zero to zero (since $f(0) = f(0 \cdot \mathbf{v}) = 0 \cdot f(\mathbf{v}) = 0$), but $f(0) = 1 \neq 0$. Any constant shift breaks linearity.

A **linear combination** of vectors $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_k$ is any expression of the form $c_1\mathbf{v}_1 + c_2\mathbf{v}_2 + \cdots + c_k\mathbf{v}_k$ where $c_1, c_2, \ldots, c_k$ are scalars. We're mixing the vectors together with different weights. This is called "linear" because the relationship between the coefficients c_i and the result is linear: if you double all coefficients, you double the result. If you add two linear combinations, you get another linear combination.

A set of vectors $\{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_k\}$ is **linearly independent** if none of them is redundant. More precisely, no vector in the set can be obtained by mixing together the others. For example, in $\mathbb{R}^2$, the vectors $\mathbf{v}_1 = \begin{bmatrix}1 & 0\end{bmatrix}$ and $\mathbf{v}_2 = \begin{bmatrix}0 & 1\end{bmatrix}$ are linearly independent because you can't get $\mathbf{v}_1$ by scaling $\mathbf{v}_2$, and vice versa. But $\mathbf{v}_1 = \begin{bmatrix}1 & 0\end{bmatrix}$, $\mathbf{v}_2 = \begin{bmatrix}0 & 1\end{bmatrix}$, and $\mathbf{v}_3 = \begin{bmatrix}2 & 3\end{bmatrix}$ are linearly dependent because $\mathbf{v}_3 = 2\mathbf{v}_1 + 3\mathbf{v}_2$.

The formal definition captures this idea precisely. The vectors are linearly independent if the only way to make the zero vector from a linear combination is to use all zero coefficients:

$$c_1\mathbf{v}_1 + c_2\mathbf{v}_2 + \cdots + c_k\mathbf{v}_k = \mathbf{0}$$

has $c_1 = c_2 = \cdots = c_k = 0$ as its only solution. Why does this work? Suppose we could write one vector, say $\mathbf{v}_1$, as a combination of the others: $\mathbf{v}_1 = a_2\mathbf{v}_2 + \cdots + a_k\mathbf{v}_k$. Then we could rearrange to get $\mathbf{v}_1 - a_2\mathbf{v}_2 - \cdots - a_k\mathbf{v}_k = \mathbf{0}$, which is a linear combination equaling zero with nonzero coefficients. So if any vector is redundant (expressible via the others), we can find a nonzero combination that equals zero. The formal definition says this can't happen. A **basis** for a vector space is a linearly independent set of vectors that spans the entire space. The number of vectors in a basis is the **dimension** of the space. For $\mathbb{R}^n$, there are infinitely many possible bases. The most common one is the **standard basis**:

$$\mathbf{e}_1 = \begin{bmatrix}1 & 0 & \vdots & 0\end{bmatrix}, \quad \mathbf{e}_2 = \begin{bmatrix}0 & 1 & \vdots & 0\end{bmatrix}, \quad \ldots, \quad \mathbf{e}_n = \begin{bmatrix}0 & 0 & \vdots & 1\end{bmatrix}$$

But we could equally well use any other set of n linearly independent vectors. For example, in $\mathbb{R}^2$, the vectors $\begin{bmatrix}1 & 1\end{bmatrix}$ and $\begin{bmatrix}1 & -1\end{bmatrix}$ form a perfectly valid basis, just a different one from the standard basis.

Every vector $\mathbf{v} \in \mathbb{R}^n$ can be uniquely written as $\mathbf{v} = v_1\mathbf{e}_1 + v_2\mathbf{e}_2 + \cdots + v_n\mathbf{e}_n$.

Here's a crucial insight for transformers: the same vector can be expressed in different bases, giving different coordinate representations of the same underlying information. Let's work through a concrete example to see exactly what this means. Consider the vector $\mathbf{v} = \begin{bmatrix}3 & 4\end{bmatrix}$ in the standard basis. This notation means:

$$\mathbf{v} = 3\begin{bmatrix}1 & 0\end{bmatrix} + 4\begin{bmatrix}0 & 1\end{bmatrix}$$

Now let's use a different basis: $\mathbf{b}_1 = \begin{bmatrix}1 & 1\end{bmatrix}$ and $\mathbf{b}_2 = \begin{bmatrix}1 & -1\end{bmatrix}$. These vectors are linearly independent (you can't get one by scaling the other), so they form a valid basis for $\mathbb{R}^2$. What are the coordinates of $\mathbf{v}$ in this new basis? We need to find c_1 and c_2 such that:

$$c_1 [1\ 1] + c_2 [1\ -1] = [3\ 4]$$

This gives us two equations:

$$c_1 + c_2 = 3$$
$$c_1 - c_2 = 4$$

Adding these equations: $2c_1 = 7$, so $c_1 = 7/2$. Subtracting: $2c_2 = -1$, so $c_2 = -1/2$. Let's verify:

$$\frac{7}{2}[1\ 1] + \left(-\frac{1}{2}\right)[1\ -1] = [7/2\ 7/2] + [-1/2\ 1/2] = [3\ 4] \quad \checkmark$$

The same geometric vector is represented as $[3\ 4]$ in the standard basis but as $[7/2\ -1/2]$ in the new basis. The vector itself hasn't changed (it still points to the same location in space), just the numbers we use to describe it.

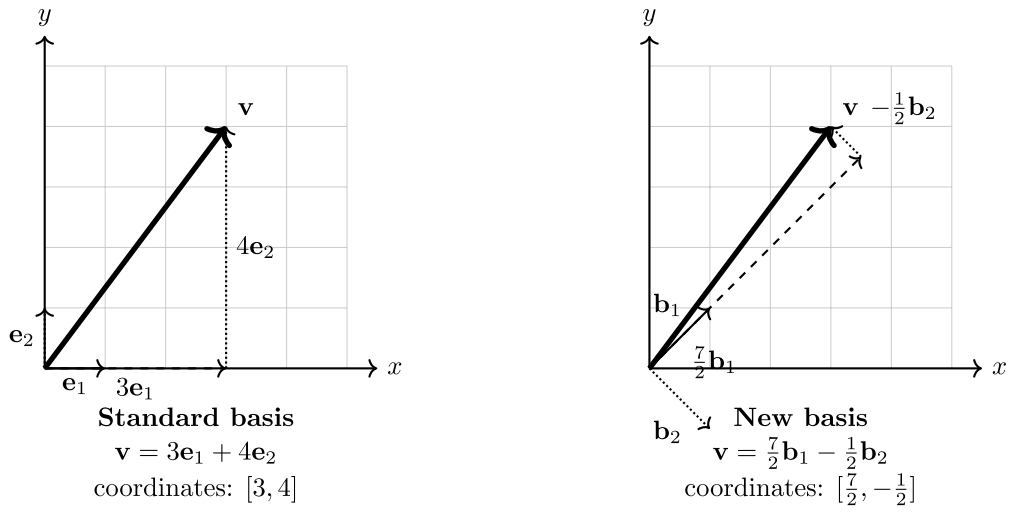

Figure 1.1: The same vector $\mathbf{v}$ in two different bases. Left: Standard basis where $\mathbf{v} = 3\mathbf{e}_1 + 4\mathbf{e}_2$, coordinates [3, 4]. Right: New basis where $\mathbf{v} = \frac{7}{2}\mathbf{b}_1 - \frac{1}{2}\mathbf{b}_2$, coordinates [7/2, -1/2]. The thick solid arrow ($\mathbf{v}$) points to the same location in both diagrams. Only the coordinate system has changed.

Notice that the thick solid arrow (our vector $\mathbf{v}$) points to exactly the same location in both diagrams. In the left diagram, we get there by going 3 steps along $\mathbf{e}_1$ (horizontal) then 4 steps along $\mathbf{e}_2$ (vertical). In the right diagram, we get to the same place by going 7/2 steps along $\mathbf{b}_1$ (diagonal up-right) then -1/2 steps along $\mathbf{b}_2$ (diagonal down-right). Different paths using different basis vectors, same destination. The coordinates [7/2, -1/2] don't mean "the point at x=3.5, y=-0.5 in standard coordinates." They mean "7/2 units along $\mathbf{b}_1$ and -1/2 units along $\mathbf{b}_2$", which lands at the same spot as [3, 4] in standard coordinates.

Why does this matter? Some bases make certain patterns obvious while others obscure them. Consider a dataset of points arranged in an ellipse. In the standard x-y basis, the pattern looks complicated. But if we rotate to a basis aligned with the ellipse's major and minor axes, the pattern becomes simple: points lie within a certain distance along each axis. We've revealed structure by choosing the right basis.

In transformers, different basis representations correspond to different "views" of the same information. When we embed a word as a vector, that vector contains information about the word's meaning, syntax, context, etc. But this information might not be easily accessible in the original basis. The attention mechanism, as we'll see, is fundamentally about learning useful basis transformations. It projects vectors into new bases

(via query, key, and value matrices) where relationships between words become clear. If we want to know "which words should attend to which," we need a basis where similar words align, dissimilar words separate. Attention learns these transformations automatically from data. The matrix operations we'll study aren't just computational mechanics, they're geometric transformations that reorganize information to make patterns visible.

1.2 The dot product and geometric intuition

The dot product (or inner product) of two vectors $\mathbf{u}, \mathbf{v} \in \mathbb{R}^n$ is:

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

The dot product has a beautiful geometric interpretation. If θ is the angle between $\mathbf{u}$ and $\mathbf{v}$, then:

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

where $\mathbf{v}\| = \sqrt{v_1^2 + v_2^2 + \cdots + v_n^2}$ is the **Euclidean norm** (length) of $\mathbf{v}$. This tells us the dot product measures how much two vectors point in the same direction. When $\theta = 0$ (parallel vectors), $\cos\theta = 1$ and the dot product is maximized. When $\theta = 90°$ (orthogonal vectors), $\cos\theta = 0$ and the dot product is zero. When $\theta = 180°$ (opposite directions), $\cos\theta = -1$ and the dot product is minimized.

Let's prove this geometric interpretation. Consider the law of cosines applied to the triangle formed by vectors $\mathbf{u}$, $\mathbf{v}$, and $\mathbf{u} - \mathbf{v}$:

$$\mathbf{u} - \mathbf{v}\|^2 = \mathbf{u}\|^2 + \mathbf{v}\|^2 - 2\mathbf{u}\|\|\mathbf{v}\| \cos\theta$$

Expanding the left side:

$$\mathbf{u} - \mathbf{v}\|^2 = (\mathbf{u} - \mathbf{v}) \cdot (\mathbf{u} - \mathbf{v}) = \mathbf{u} \cdot \mathbf{u} - 2\mathbf{u} \cdot \mathbf{v} + \mathbf{v} \cdot \mathbf{v} = \mathbf{u}\|^2 - 2\mathbf{u} \cdot \mathbf{v} + \mathbf{v}\|^2$$

Equating the two expressions:

$$\mathbf{u}\|^2 - 2\mathbf{u} \cdot \mathbf{v} + \mathbf{v}\|^2 = \mathbf{u}\|^2 + \mathbf{v}\|^2 - 2\mathbf{u}\|\|\mathbf{v}\| \cos\theta$$

Simplifying:

$$-2\mathbf{u} \cdot \mathbf{v} = -2\mathbf{u}\|\|\mathbf{v}\| \cos\theta$$

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

This geometric interpretation is crucial for understanding transformers. We constantly compute dot products between vectors to measure similarity. A large dot product means the vectors are similar (pointing in similar directions), while a small dot product means they're dissimilar. This is how transformers decide which pieces of information are related and should interact.

1.3 Matrices and linear transformations

A matrix is a rectangular array of numbers. We write matrices as uppercase bold letters:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

This matrix $\mathbf{A}$ has m rows and n columns, so we say $\mathbf{A} \in \mathbb{R}^{m \times n}$. Every matrix represents a linear transformation. When we multiply a vector $\mathbf{x} \in \mathbb{R}^n$ by matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$, we get a vector $\mathbf{y} \in \mathbb{R}^m$:

$$\mathbf{y} = \mathbf{A}\mathbf{x}$$

The i-th component of $\mathbf{y}$ is:

$$y_i = \sum_{j=1}^{n} a_{ij} x_j$$

This is just the dot product of the i-th row of $\mathbf{A}$ with $\mathbf{x}$. We can think of matrix-vector multiplication in two equivalent ways:

1. **Row perspective**: Each element of $\mathbf{y}$ is a weighted combination of the elements of $\mathbf{x}$. Let's see a concrete example:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \begin{bmatrix} 7 \\ 8 \\ 9 \end{bmatrix} = \begin{bmatrix} 1 \cdot 7 + 2 \cdot 8 + 3 \cdot 9 \\ 4 \cdot 7 + 5 \cdot 8 + 6 \cdot 9 \end{bmatrix} = \begin{bmatrix} 7 + 16 + 27 \\ 28 + 40 + 54 \end{bmatrix} = \begin{bmatrix} 50 \\ 122 \end{bmatrix}$$

The first component (50) comes from the dot product of the first row $[1, 2, 3]$ with the vector $[7, 8, 9]$. The second component (122) comes from the dot product of the second row $[4, 5, 6]$ with the same vector.

2. **Column perspective**: $\mathbf{y}$ is a linear combination of the columns of $\mathbf{A}$, with weights given by $\mathbf{x}$:

$$\mathbf{y} = x_1 \mathbf{a}_1 + x_2 \mathbf{a}_2 + \cdots + x_n \mathbf{a}_n$$

where $\mathbf{a}_j$ is the j-th column of $\mathbf{A}$. Using the same example:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \begin{bmatrix} 7 \\ 8 \\ 9 \end{bmatrix} = 7 \begin{bmatrix} 1 \\ 4 \end{bmatrix} + 8 \begin{bmatrix} 2 \\ 5 \end{bmatrix} + 9 \begin{bmatrix} 3 \\ 6 \end{bmatrix} = \begin{bmatrix} 7 \\ 28 \end{bmatrix} + \begin{bmatrix} 16 \\ 40 \end{bmatrix} + \begin{bmatrix} 27 \\ 54 \end{bmatrix} = \begin{bmatrix} 50 \\ 122 \end{bmatrix}$$

We're taking 7 copies of the first column, 8 copies of the second column, and 9 copies of the third column, then adding them together. Same result, different interpretation.

Both perspectives are useful. In transformers, the row perspective helps us understand how each output dimension depends on the input. The column perspective helps us see matrix multiplication as mixing together column vectors.

When we multiply two matrices $\mathbf{A} \in \mathbb{R}^{m \times n}$ and $\mathbf{B} \in \mathbb{R}^{n \times p}$, we get $\mathbf{C} = \mathbf{A}\mathbf{B} \in \mathbb{R}^{m \times p}$:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

Matrix multiplication is associative ($\mathbf{A}(\mathbf{B}\mathbf{C}) = (\mathbf{A}\mathbf{B})\mathbf{C}$) and distributive ($\mathbf{A}(\mathbf{B} + \mathbf{C}) = \mathbf{A}\mathbf{B} + \mathbf{A}\mathbf{C}$), but generally not commutative ($\mathbf{A}\mathbf{B} \neq \mathbf{B}\mathbf{A}$). The order matters. For example:

$$\begin{bmatrix}1 & 2 & 0 & 1\end{bmatrix}\begin{bmatrix}0 & 1 & 1 & 0\end{bmatrix} = \begin{bmatrix}2 & 1 & 1 & 0\end{bmatrix}, \quad \text{but} \quad \begin{bmatrix}0 & 1 & 1 & 0\end{bmatrix}\begin{bmatrix}1 & 2 & 0 & 1\end{bmatrix} = \begin{bmatrix}0 & 1 & 1 & 2\end{bmatrix}$$

Different results! This is why we must be careful about order in transformer architectures.

The **transpose** of a matrix $\mathbf{A}$ is denoted $\mathbf{A}^T$ and defined by swapping rows and columns: $(A^T)_{ij} = a_{ji}$. Some useful properties:

$$(\mathbf{A}^T)^T = \mathbf{A}, \quad (\mathbf{AB})^T = \mathbf{B}^T\mathbf{A}^T, \quad (\mathbf{A}+\mathbf{B})^T = \mathbf{A}^T + \mathbf{B}^T$$

A matrix $\mathbf{A}$ is **symmetric** if $\mathbf{A}^T = \mathbf{A}$. Symmetric matrices have special properties we'll encounter when we study attention patterns.

1.4 Norms and distances

We need a way to measure the "size" or "length" of vectors. In ordinary space, we use the Pythagorean theorem: a vector $\mathbf{v} = \begin{bmatrix}3 & 4\end{bmatrix}$ has length $\sqrt{3^2 + 4^2} = \sqrt{9+16} = 5$. But there are other sensible ways to measure size, and different measures are useful in different contexts.

A **norm** is a function that assigns a non-negative size to each vector. For a function $\|\cdot\|$ to be a proper norm, it must satisfy three properties:

1. **Positive definiteness**: $\mathbf{v}\| \geq 0$ with equality if and only if $\mathbf{v} = \mathbf{0}$
2. **Homogeneity**: $\|a\mathbf{v}\| = |a|\|\mathbf{v}\|$ for any scalar a
3. **Triangle inequality**: $\|\mathbf{u} + \mathbf{v}\| \leq \|\mathbf{u}\| + \|\mathbf{v}\|$

Let's understand what these mean. Positive definiteness says only the zero vector has zero size. Homogeneity says if you scale a vector by factor a, its size scales by $|a|$. The triangle inequality says the direct path from origin to $\mathbf{u} + \mathbf{v}$ is never longer than going via $\mathbf{u}$ first. This generalizes the geometric fact that a straight line is the shortest path.

What would violate these properties? Consider $f(\mathbf{v}) = \|\mathbf{v}\|_2 + 1$. This seems like a reasonable "size" measure, but it fails homogeneity: $f(2\mathbf{v}) = 2\|\mathbf{v}\|_2 + 1$, while $2f(\mathbf{v}) = 2\|\mathbf{v}\|_2 + 2$. The constant term breaks scaling. Or consider $g(\mathbf{v}) = \sum_i v_i$ (sum without absolute values). For $\mathbf{v} = \begin{bmatrix}1 & -1\end{bmatrix}$, we get $g(\mathbf{v}) = 0$ despite $\mathbf{v} \neq \mathbf{0}$, violating positive definiteness.

1.4.1 Common norms

The most common norms form a family called p-norms:

$$\mathbf{v}\|_p = \left(\sum_{i=1}^{n} |v_i|^p\right)^{1/p}$$

Different values of p give different norms:

L^1 **norm** ($p = 1$): $\mathbf{v}\|_1 = \sum_{i=1}^{n} |v_i|$. This sums the absolute values of components. For $\mathbf{v} = \begin{bmatrix}3 & -4\end{bmatrix}$, we get $\mathbf{v}\|_1 = |3| + |-4| = 7$. This is also called the Manhattan distance or taxicab norm, because it measures distance as if you could only travel along axis-aligned streets.

L^2 **norm** ($p = 2$): $\mathbf{v}\|_2 = \sqrt{\sum_{i=1}^{n} v_i^2}$. This is the Euclidean norm, the "ordinary" geometric length. For $\mathbf{v} = \begin{bmatrix}3 & -4\end{bmatrix}$, we get $\mathbf{v}\|_2 = \sqrt{9+16} = 5$. This is what we usually mean by "length" in everyday geometry.

L^∞ **norm** ($p = \infty$): $\mathbf{v}\|_\infty = \max_i |v_i|$. This takes the largest absolute component. For $\mathbf{v} = \begin{bmatrix}3 & -4\end{bmatrix}$, we get $\mathbf{v}\|_\infty = \max(3, 4) = 4$. The name comes from taking the limit as $p \to \infty$ in the p-norm formula.

Let's verify these give different values on a concrete example. For $\mathbf{v} = \begin{bmatrix}2 & 3 & -1\end{bmatrix}$:

$$\mathbf{v}\|_1 = 2 + 3 + 1 = 6, \quad \mathbf{v}\|_2 = \sqrt{4 + 9 + 1} = \sqrt{14} \approx 3.74, \quad \mathbf{v}\|_\infty = 3$$

All three are valid norms. In transformers, we most commonly use the L^2 norm because it has nice geometric and optimization properties.

Let's look more closely at the triangle inequality: $\|\mathbf{u} + \mathbf{v}\| \leq \|\mathbf{u}\| + \|\mathbf{v}\|$. This says the direct path from origin to $\mathbf{u} + \mathbf{v}$ is never longer than going via $\mathbf{u}$ first. Geometrically, it captures the fact that a straight line is the shortest path between two points.

What if this didn't hold? Imagine a "norm" where $\|\mathbf{u} + \mathbf{v}\| > \|\mathbf{u}\| + \|\mathbf{v}\|$ for some vectors. Taking a detour would be shorter than going directly. This would break our geometric intuition about distance. For instance, if walking from A to B directly took 10 minutes, but walking from A to C then C to B took only 8 minutes, our notion of "distance" would be broken. You could keep finding shorter and shorter paths by adding more intermediate points, which makes no sense for measuring actual spatial distance. The triangle inequality ensures that distances behave sensibly: the direct route is always shortest (or at worst, tied).

1.4.2 Distance

Once we have a norm, we can define **distance** between vectors. The distance from $\mathbf{u}$ to $\mathbf{v}$ is simply the norm of their difference:

$$d(\mathbf{u}, \mathbf{v}) = \|\mathbf{u} - \mathbf{v}\|$$

Different norms give different distance measures. Using L^2, we get Euclidean distance. Using L^1, we get Manhattan distance. In transformers, distances between embedding vectors indicate semantic similarity: words with similar meanings have embeddings that are close together.

1.4.3 Unit vectors and normalization

A **unit vector** is one with norm 1. We can convert any nonzero vector $\mathbf{v}$ into a unit vector pointing in the same direction by **normalizing** it:

$$\hat{\mathbf{v}} = \frac{\mathbf{v}}{\|\mathbf{v}\|}$$

Let's verify this has norm 1: $\|\hat{\mathbf{v}}\| = \left\|\frac{\mathbf{v}}{\|\mathbf{v}\|}\right\| = \frac{1}{\|\mathbf{v}\|}\|\mathbf{v}\| = 1$ by homogeneity. For example, $\mathbf{v} = [3\ 4]$ has $\mathbf{v}\|_2 = 5$, so $\hat{\mathbf{v}} = [3/5\ 4/5]$ is the unit vector in the same direction.

Normalization is ubiquitous in transformers. Layer normalization rescales activation vectors to have controlled statistics (zero mean and unit variance). This stabilizes training by preventing activations from growing too large or shrinking too small. When we compute attention weights, we often normalize vectors before taking dot products to ensure numerical stability.

1.4.4 Matrix norms

We can also define norms for matrices. The simplest is the **Frobenius norm**, which treats the matrix as a long vector:

$$\mathbf{A}\|_F = \sqrt{\sum_{i,j} a_{ij}^2}$$

For example, $\|[1 \quad 2\ 3 \quad 4]\|_F = \sqrt{1 + 4 + 9 + 16} = \sqrt{30}$. This measures the overall "size" of all matrix entries.

1.5 Matrix properties and decompositions

1.5.1 Rank

The **rank** of a matrix $\mathbf{A}$ is the dimension of the vector space spanned by its columns (equivalently, by its rows). Intuitively, rank measures the number of independent directions in the output. Consider $\mathbf{A} \in \mathbb{R}^{m \times n}$ as a linear transformation from $\mathbb{R}^n$ to $\mathbb{R}^m$. The rank tells us the dimension of the image: how much of the output space can we actually reach?

Let's see concrete examples. The matrix $\mathbf{A} = \begin{bmatrix} 1 & 0 & 0 & 1 \end{bmatrix}$ has rank 2. Both columns are linearly independent, and we can reach any point in $\mathbb{R}^2$ by multiplying appropriate vectors. But consider $\mathbf{B} = \begin{bmatrix} 1 & 2 & 2 & 4 \end{bmatrix}$. The second column is twice the first: $\begin{bmatrix} 2 & 4 \end{bmatrix} = 2 \begin{bmatrix} 1 & 2 \end{bmatrix}$.

Why does this mean we only span one dimension? Recall that $\mathbf{Bx}$ is a linear combination of $\mathbf{B}$'s columns using weights from $\mathbf{x}$. The first column points in direction $\begin{bmatrix} 1 & 2 \end{bmatrix}$, and the second column also points in direction $\begin{bmatrix} 1 & 2 \end{bmatrix}$ (just scaled by 2). Both columns lie on the same line through the origin. Any combination of them must also lie on that same line. We're mixing together two vectors that both point in the same direction, so the result can only point in that direction too. We can never "escape" the line to reach other parts of the 2D plane. The column space of $\mathbf{B}$ is just the line along $\begin{bmatrix} 1 & 2 \end{bmatrix}$, which is one-dimensional.

It has rank 1. For any vector $\mathbf{x} = \begin{bmatrix} x_1 & x_2 \end{bmatrix}$:

$$\mathbf{Bx} = x_1 \begin{bmatrix} 1 & 2 \end{bmatrix} + x_2 \begin{bmatrix} 2 & 4 \end{bmatrix} = (x_1 + 2x_2) \begin{bmatrix} 1 & 2 \end{bmatrix}$$

All outputs lie on the same line. The transformation collapses the 2D plane onto a 1D line.

What does "information is lost" mean? Consider two different inputs:

$$\mathbf{x}_1 = \begin{bmatrix} 1 & 0 \end{bmatrix}, \quad \mathbf{x}_2 = \begin{bmatrix} -1 & 1 \end{bmatrix}$$

Let's multiply both by $\mathbf{B}$:

$$\mathbf{Bx}_1 = \begin{bmatrix} 1 & 2 & 2 & 4 \end{bmatrix} \begin{bmatrix} 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 2 \end{bmatrix}$$

$$\mathbf{Bx}_2 = \begin{bmatrix} 1 & 2 & 2 & 4 \end{bmatrix} \begin{bmatrix} -1 & 1 \end{bmatrix} = \begin{bmatrix} -1+2 & -2+4 \end{bmatrix} = \begin{bmatrix} 1 & 2 \end{bmatrix}$$

The same output! Two completely different inputs map to the same result. If someone gives us the output $\begin{bmatrix} 1 & 2 \end{bmatrix}$, we can't tell whether it came from $\mathbf{x}_1$, $\mathbf{x}_2$, or infinitely many other vectors. The original information about which input we started with is irrecoverably lost. This is why $\mathbf{B}$ has no inverse: we can't "undo" the transformation because many inputs collapse to each output.

Geometrically, imagine the 2D plane collapsing onto a line. Every point perpendicular to the line direction gets squashed to zero in that direction. Points at $(1, 0)$, $(0, 0.5)$, $(-1, 1)$, and infinitely many others all land at the same spot on the output line. The dimension perpendicular to $\begin{bmatrix} 1 & 2 \end{bmatrix}$ is completely annihilated.

A matrix is **full rank** if its rank equals the minimum of its dimensions. For a square matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$, full rank means rank equals n. This is crucial: full rank square matrices are invertible. They don't lose information, so we can "undo" the transformation. Non-full-rank matrices collapse space and are not invertible.

1.5.2 Determinant

The **determinant** tells us how a matrix transformation changes areas (in 2D) or volumes (in higher dimensions). For example, apply the matrix $\begin{bmatrix} 2 & 0 & 0 & 3 \end{bmatrix}$ to the unit square. It stretches horizontally by 2 and vertically by 3, producing a 2×3 rectangle with area 6. The determinant is $2 \cdot 3 - 0 \cdot 0 = 6$, exactly the area scaling factor. But why does this formula work? Let's build deep intuition.

The unit square has edges along the standard basis vectors: one edge goes from $(0,0)$ to $(1,0)$ (along $\mathbf{e}_1$), another from $(0,0)$ to $(0,1)$ (along $\mathbf{e}_2$). When we apply a matrix $\mathbf{A}$, where do these edges go? The columns of $\mathbf{A}$ tell us:

$$\mathbf{A} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \implies \mathbf{e}_1 \to \begin{bmatrix} a \\ c \end{bmatrix}, \quad \mathbf{e}_2 \to \begin{bmatrix} b \\ d \end{bmatrix}$$

The unit square transforms into a parallelogram whose sides are the columns of $\mathbf{A}$. So the question "what area does the unit square become?" is equivalent to "what's the area of the parallelogram spanned by the columns?"

The area of a parallelogram with sides $\mathbf{u}$ and $\mathbf{v}$ is: base times height, where height is the perpendicular distance. If $\mathbf{u} = [a, c]^T$ and $\mathbf{v} = [b, d]^T$:

- Take $\mathbf{u}$ as the base, with length $\sqrt{a^2 + c^2}$
- The height is how far $\mathbf{v}$ extends perpendicular to $\mathbf{u}$
- After working through the geometry, this equals $\frac{|ad-bc|}{\sqrt{a^2+c^2}}$
- Multiplying base $\times$ height: $\sqrt{a^2 + c^2} \cdot \frac{|ad-bc|}{\sqrt{a^2+c^2}} = |ad - bc|$

So the determinant formula:

$$\det(\mathbf{A}) = ad - bc$$

is exactly the (signed) area of the parallelogram formed by the columns. The sign captures orientation: positive if the columns maintain the same rotational order as the original basis (counterclockwise), negative if they flip it.

Here's another way to see it intuitively. The determinant measures how much "independent spreading" the columns do. If both columns point in similar directions, they don't spread much, so the parallelogram is thin (small area). If they point in perpendicular directions, they spread maximally (large area). If they point in exactly the same direction, they don't spread at all (zero area, the parallelogram collapses to a line).

Let's see this concretely. Consider $\mathbf{A} = \begin{bmatrix} 3 & 1 \\ 1 & 2 \end{bmatrix}$. The corners of the unit square transform as follows:

$$(0,0) \to A \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{1.1}$$

$$(1,0) \to A \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 3 \\ 1 \end{bmatrix} \tag{1.2}$$

$$(0,1) \to A \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \tag{1.3}$$

$$(1,1) \to A \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 4 \\ 3 \end{bmatrix} \tag{1.4}$$

The unit square becomes a parallelogram with corners at $(0,0)$, $(3,1)$, $(1,2)$, $(4,3)$. The area of a parallelogram with sides $\mathbf{u}$ and $\mathbf{v}$ is $|A \times A|$ (the magnitude of the cross product). For our parallelogram with sides $[3,1]^T$ and $[1,2]^T$:

$$\text{area} = |3 \cdot 2 - 1 \cdot 1| = |6 - 1| = 5$$

Now compute the determinant:

$$\det(\mathbf{A}) = 3 \cdot 2 - 1 \cdot 1 = 6 - 1 = 5$$

They match! The determinant directly gives us the area scaling factor. The original square had area 1, the transformed parallelogram has area 5, so $\mathbf{A}$ scales areas by a factor of 5.

This extends to any shape, not just the unit square. If we have a triangle with area 10, applying $\mathbf{A}$ transforms it to a new triangle with area $10 \times 5 = 50$. The determinant is the universal area scaling factor.

Now consider our rank-deficient matrix from earlier: $\mathbf{B} = \begin{bmatrix}1 & 2\ 2 & 4\end{bmatrix}$. Let's see what happens to the unit square:

$$(0,0) \to \begin{bmatrix}0\\0\end{bmatrix} \tag{1.5}$$

$$(1,0) \to \begin{bmatrix}1\\2\end{bmatrix} \tag{1.6}$$

$$(0,1) \to \begin{bmatrix}2\\4\end{bmatrix} = 2\begin{bmatrix}1\\2\end{bmatrix} \tag{1.7}$$

$$(1,1) \to \begin{bmatrix}3\\6\end{bmatrix} = 3\begin{bmatrix}1\\2\end{bmatrix} \tag{1.8}$$

All four corners lie on the line through $\begin{bmatrix}1 & 2\end{bmatrix}$! The square collapses to a line segment from $(0,0)$ to $(3,6)$. A line segment has zero area. The determinant:

$$\det(\mathbf{B}) = 1 \cdot 4 - 2 \cdot 2 = 4 - 4 = 0$$

Zero! This makes perfect sense. When a transformation collapses 2D shapes onto a 1D line, all areas become zero. In general, $\det(\mathbf{A}) = 0$ if and only if $\mathbf{A}$ is not full rank. The transformation loses a dimension, squashing everything flat.

When $\det(\mathbf{A}) \neq 0$, areas are scaled but not annihilated. This means the transformation doesn't collapse space, so it's invertible. We can undo it. When $\det(\mathbf{A}) = 0$, areas become zero, dimensions collapse, and we can't reverse the process.

What about the sign of the determinant? Consider $\mathbf{R} = \begin{bmatrix}1 & 0\ 0 & -1\end{bmatrix}$:

$$\det(\mathbf{R}) = 1 \cdot (-1) - 0 \cdot 0 = -1$$

This matrix flips the y-axis. The unit square still becomes a parallelogram with area $|-1| = 1$ (no scaling), but the orientation reverses. Imagine the square has corners labeled clockwise as $(0,0)$, $(1,0)$, $(1,1)$, $(0,1)$. After transformation, the corners appear in counterclockwise order. The sign tells us about orientation: positive preserves it, negative reverses it.

In higher dimensions, the pattern continues. For a 3×3 matrix:

$$\mathbf{A} = \begin{bmatrix}a & b & c\ d & e & f\ g & h & i\end{bmatrix}$$

The determinant is:

$$\det(\mathbf{A}) = a(ei - fh) - b(di - fg) + c(dh - eg)$$

This looks complex, but the principle is the same: it measures how much the transformation scales volumes. Let's compute an example:

$$\mathbf{A} = \begin{bmatrix}2 & 0 & 0\ 0 & 3 & 0\ 0 & 0 & 1\end{bmatrix}$$

This is a diagonal matrix that scales the x-axis by 2, y-axis by 3, and leaves z-axis unchanged:

$$\det(\mathbf{A}) = 2(3 \cdot 1 - 0 \cdot 0) - 0(\cdots) + 0(\cdots) = 2 \cdot 3 = 6$$

The unit cube (volume 1) gets stretched to a box with dimensions $2 \times 3 \times 1$, which has volume 6. For diagonal matrices, the determinant is simply the product of diagonal entries: $\det(\mathbf{A}) = 2 \cdot 3 \cdot 1 = 6$. This makes intuitive sense: if you scale each dimension independently, the total volume scales by the product of all scaling factors.

The general principle: $\det(\mathbf{A})$ is the factor by which volumes get multiplied. Zero determinant means collapse to lower dimension. Negative means orientation reversal. This holds in any dimension.

1.5.3 Eigenvalues and eigenvectors

An **eigenvalue** of $\mathbf{A}$ is a scalar λ such that:

$$\mathbf{A}\mathbf{v} = \lambda\mathbf{v}$$

for some nonzero vector $\mathbf{v}$, called an **eigenvector**. This is a profound concept: when we apply $\mathbf{A}$ to eigenvector $\mathbf{v}$, the result points in the same direction as $\mathbf{v}$, just scaled by λ. Most vectors get rotated and stretched in complicated ways when multiplied by a matrix. Eigenvectors are special directions that only get scaled.

Why does this matter? Eigenvectors reveal the natural "axes" of a transformation. If we express vectors in the eigenvector basis, matrix multiplication becomes trivial: each component just gets multiplied by its corresponding eigenvalue. This is why eigendecomposition is so powerful.

Let's find the eigenvalues and eigenvectors of $\mathbf{A} = \begin{bmatrix} 4 & 1 \\ 2 & 3 \end{bmatrix}$. We need $\mathbf{A}\mathbf{v} = \lambda\mathbf{v}$, which we can rewrite as:

$$\mathbf{A}\mathbf{v} - \lambda\mathbf{v} = \mathbf{0}$$

$$(\mathbf{A} - \lambda\mathbf{I})\mathbf{v} = \mathbf{0}$$

For a nonzero solution $\mathbf{v}$ to exist, the matrix $\mathbf{A} - \lambda\mathbf{I}$ must not be invertible, so:

$$\det(\mathbf{A} - \lambda\mathbf{I}) = 0$$

For our example:

$$\det\left(\begin{bmatrix} 4-\lambda & 1 \\ 2 & 3-\lambda \end{bmatrix}\right) = (4-\lambda)(3-\lambda) - 2 = 0$$

$$12 - 4\lambda - 3\lambda + \lambda^2 - 2 = 0$$

$$\lambda^2 - 7\lambda + 10 = 0$$

$$(\lambda - 5)(\lambda - 2) = 0$$

The eigenvalues are $\lambda_1 = 5$ and $\lambda_2 = 2$. Now let's find the eigenvectors.

For $\lambda_1 = 5$:

$$(\mathbf{A} - 5\mathbf{I})\mathbf{v} = \begin{bmatrix} -1 & 1 & 2 & -2 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

From the first row: $-v_1 + v_2 = 0$, so $v_2 = v_1$. One eigenvector is $\mathbf{v}_1 = [1\ 1]$.

For $\lambda_2 = 2$:

$$(\mathbf{A} - 2\mathbf{I})\mathbf{v} = \begin{bmatrix} 2 & 1 & 2 & 1 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

From the first row: $2v_1 + v_2 = 0$, so $v_2 = -2v_1$. One eigenvector is $\mathbf{v}_2 = [1\ -2]$.

Let's verify:

$$\mathbf{A}\mathbf{v}_1 = \begin{bmatrix} 4 & 1 & 2 & 3 \end{bmatrix} [1\ 1] = [5\ 5] = 5\,[1\ 1] \quad \checkmark$$

$$\mathbf{A}\mathbf{v}_2 = \begin{bmatrix} 4 & 1 & 2 & 3 \end{bmatrix} [1\ -2] = [2\ -4] = 2\,[1\ -2] \quad \checkmark$$

Perfect! When we apply $\mathbf{A}$ to $\mathbf{v}_1$, it gets scaled by 5. When we apply $\mathbf{A}$ to $\mathbf{v}_2$, it gets scaled by 2. These are the natural directions of the transformation.

Here's where eigenvectors become powerful: they simplify matrix multiplication. Any vector $\mathbf{x}$ can be written as a combination of eigenvectors: $\mathbf{x} = c_1\mathbf{v}_1 + c_2\mathbf{v}_2$. Then:

$$\mathbf{A}\mathbf{x} = \mathbf{A}(c_1\mathbf{v}_1 + c_2\mathbf{v}_2) = c_1\mathbf{A}\mathbf{v}_1 + c_2\mathbf{A}\mathbf{v}_2 = c_1\lambda_1\mathbf{v}_1 + c_2\lambda_2\mathbf{v}_2$$

Instead of doing full matrix multiplication, we just multiply each coefficient by its eigenvalue! Let's try this with $\mathbf{x} = [3\ 1]$. First, express $\mathbf{x}$ in the eigenvector basis. We need c_1 and c_2 such that:

$$c_1\,[1\ 1] + c_2\,[1\ -2] = [3\ 1]$$

This gives $c_1 + c_2 = 3$ and $c_1 - 2c_2 = 1$. Solving: $c_1 = 7/3$ and $c_2 = 2/3$. Now applying $\mathbf{A}$ is trivial:

$$\mathbf{A}\mathbf{x} = \frac{7}{3} \cdot 5 \cdot [1\ 1] + \frac{2}{3} \cdot 2 \cdot [1\ -2] = \frac{35}{3}\,[1\ 1] + \frac{4}{3}\,[1\ -2] = [39/3\ 27/3] = [13\ 9]$$

Let's verify by direct multiplication:

$$\mathbf{A}\mathbf{x} = \begin{bmatrix} 4 & 1 & 2 & 3 \end{bmatrix} [3\ 1] = [12 + 1\ 6 + 3] = [13\ 9] \quad \checkmark$$

The eigenvector approach seems longer here, but for repeated multiplications (like $\mathbf{A}^{100}\mathbf{x}$), it's vastly simpler: just raise each eigenvalue to the power. In the eigenvector basis, $\mathbf{A}^{100}\mathbf{x} = c_1 \cdot 5^{100} \cdot \mathbf{v}_1 + c_2 \cdot 2^{100} \cdot \mathbf{v}_2$. No need to multiply the matrix by itself 100 times.

What if a matrix has no real eigenvalues? Consider a rotation matrix $\mathbf{R} = \begin{bmatrix} \cos\theta & -\sin\theta & \sin\theta & \cos\theta \end{bmatrix}$ for $\theta \neq 0, \pi$. This rotates vectors, so no direction stays on the same line. There are no real eigenvectors. The eigenvalues are complex: $\lambda = \cos\theta \pm i\sin\theta = e^{\pm i\theta}$. Complex eigenvalues indicate rotational behavior.

In transformers, eigenvalues appear in several ways. Weight matrices have eigenvalue spectra that affect gradient flow during training. Large eigenvalues can cause exploding gradients, while very small ones cause vanishing gradients. Normalization techniques (like layer norm) can be understood as controlling these eigenvalue distributions. When we analyze attention patterns, the eigenstructure of attention weight matrices reveals dominant patterns of information flow.

1.5.4 Orthogonal matrices

An **orthogonal matrix Q** satisfies $\mathbf{Q}^T\mathbf{Q} = \mathbf{I}$. This means the columns of $\mathbf{Q}$ form an orthonormal basis: each column has norm 1, and different columns are perpendicular. Orthogonal matrices represent pure rotations and reflections without any scaling or skewing.

The key property: orthogonal matrices preserve lengths and angles. For any vectors $\mathbf{u}, \mathbf{v}$:

$$\mathbf{Q}\mathbf{v}\|_2 = \|\mathbf{v}\|_2, \quad (\mathbf{Q}\mathbf{u}) \cdot (\mathbf{Q}\mathbf{v}) = \mathbf{u} \cdot \mathbf{v}$$

This makes sense geometrically: rotations and reflections don't change distances or angles, only orientation. The determinant of an orthogonal matrix is always ± 1: rotations have $\det(\mathbf{Q}) = 1$, reflections have $\det(\mathbf{Q}) = -1$. No volume scaling occurs.

1.6 Intuition: The matrix as a universal lens

We have covered eigenvalues, basis changes, and dot products. Before we move on to calculus, we must pause and internalize a crucial intuition.

In traditional mathematics, we often treat matrices as static tables of data or simple systems of equations. In deep learning we must view matrices differently.

A matrix is a machine. It is a lens.

Every time you see a matrix multiplication $\mathbf{y} = \mathbf{W}\mathbf{x}$ in a neural network, the matrix $\mathbf{W}$ is performing a specific physical action on the vector $\mathbf{x}$. It is not just multiplying numbers; it is transforming information.

Crucially, the *nature* of this transformation is dictated by the shape of the matrix. Specifically, it depends on how the number of outputs compares to the number of inputs.

1.6.1 1. The compressor (bottleneck)

The first type of lens appears when a matrix maps a large vector to a smaller one (many inputs to fewer outputs). This forces **lossy compression**. It is physically impossible to keep all the information from the larger space, so the matrix must make choices about what to keep and what to throw away.

Mathematically, if we have an input $\mathbf{x} \in \mathbb{R}^{512}$ and a matrix $\mathbf{W} \in \mathbb{R}^{64\times512}$, the output $\mathbf{y}$ has only 64 dimensions. Let's look at the operation row by row. If we call the rows of the matrix $\mathbf{r}_1, \mathbf{r}_2, \dots, \mathbf{r}_{64}$, the multiplication looks like this:

$$\mathbf{y} = \mathbf{W}\mathbf{x} = \begin{bmatrix} — \ \mathbf{r}_1 \ — \\ — \ \mathbf{r}_2 \ — \\ \vdots \\ — \ \mathbf{r}_{64} \ — \end{bmatrix} \mathbf{x} = \begin{bmatrix} \mathbf{r}_1 \cdot \mathbf{x} \\ \mathbf{r}_2 \cdot \mathbf{x} \\ \vdots \\ \mathbf{r}_{64} \cdot \mathbf{x} \end{bmatrix}$$

Each element y_i is simply the dot product $\mathbf{r}_i \cdot \mathbf{x}$. Recall that the dot product is a similarity test. If $\mathbf{x}$ is perpendicular (orthogonal) to a row $\mathbf{r}_i$, the result is 0.

Now, imagine a part of vector $\mathbf{x}$ that points in a direction completely different from *all* 64 rows. It is orthogonal to every single row vector. * $\mathbf{r}_1 \cdot \mathbf{x} = 0$ * $\mathbf{r}_2 \cdot \mathbf{x} = 0$ * … * $\mathbf{r}_{64} \cdot \mathbf{x} = 0$

The result is the zero vector $\mathbf{0}$. This is exactly what the **null space** is: the set of all inputs that get "annihilated" (mapped to zero) because they don't align with any of the matrix's feature detectors. The matrix is blind to these directions. It destroys that information completely.

Imagine describing a complex scene, like a busy city street, to a friend over a noisy phone line. You have 10 seconds. You cannot list every photon of light or every cracked pavement stone. You must prioritize: "Red car. Speeding. Police chasing."

A matrix that reduces dimension acts as this filter. By reducing the available space, we force the model to identify the *essence* of the input. It strips away noise, nuance, and irrelevant details. The matrix learns to be a "feature detector" for the most critical patterns. Only inputs matching these patterns survive the projection; everything else is ignored. In practice, we use this to distill a complex object (like a word with many definitions) into a focused representation of just one specific aspect (like its grammatical role).

1.6.2 2. The expander (unfolding)

The second type of lens appears when a matrix maps a small vector to a larger one (few inputs to many outputs). This creates **space for analysis**. It allows the model to "unpack" information that was tightly compressed.

Mathematically, consider mapping $\mathbf{x} \in \mathbb{R}^{512}$ to $\mathbf{y} \in \mathbb{R}^{2048}$. We define 2048 new direction vectors (the rows). Because we have more outputs than inputs, we are generating an **over-complete** representation.

Does this create "new" information? No. The output vector $\mathbf{y}$ lives in a high-dimensional space ($\mathbb{R}^{2048}$), but it is constrained to a lower-dimensional **subspace** (specifically, a 512-dimensional flat sheet called the *column space* or *image* of the matrix). You cannot reach *every* point in the 2048-dimensional space, only those that can be formed by combining the columns of $\mathbf{W}$.

So why bother? This is similar to adding **polynomial features** in regression. Imagine you have points on a 1D line that are red-blue-red. You cannot separate them with a single straight cut (a linear classifier). But if you map each point x to a 2D vector $[x, x^2]$, the points lift onto a parabola. Now, a simple straight line can slice through the parabola to separate red from blue.

The expander matrix performs a similar "lifting" operation. It computes 2048 distinct linear combinations of the original features. It effectively says: "Let's look at the data from 2048 different angles simultaneously." By projecting the data onto this higher-dimensional manifold, we increase the probability that complex, entangled patterns will become linearly separable, allowing the subsequent layers (like ReLU) to slice them apart cleanly.

Think of a crumpled piece of paper with writing on it. In its compressed ball state, the words touch and overlap so you cannot read them. To understand it, you must unfold it into a larger flat space. Or think of separate ingredients like flour, sugar, and eggs versus a mixed batter. To chemically analyze the batter, you might need to separate the components back out.

A matrix that increases dimension creates this "wiggle room." By increasing the space, we make it possible to separate complex patterns that were entangled in the lower dimension. We project the data into a high-dimensional space where it is easier to categorize. The matrix generates an "over-complete" representation, computing many distinct combinations of the input features to create a massive menu of potential patterns to look for. In practice, we use this to perform complex logical operations. We expand the data, inspect it in detail, and then compress it back down.

1.6.3 3. The mixer (rotation/perspective)

The third type of lens appears when the input and output sizes are the same. Since the matrix isn't compressing or expanding capacity, it is instead **translating languages**. It acts as a rotation or a change of perspective.

Mathematically, if $\mathbf{W} \in \mathbb{R}^{512 \times 512}$ is invertible (full rank), it performs a **change of basis**. The output $\mathbf{y}$ contains exactly the same amount of information as the input $\mathbf{x}$, just reorganized. We can write $\mathbf{y}$ as a linear combination of the columns of $\mathbf{W}$: $\mathbf{y} = x_1\mathbf{c}_1 + x_2\mathbf{c}_2 + \cdots + x_n\mathbf{c}_n$. The matrix effectively rotates the vector space, aligning the data's internal axes with the standard axes that the next layer expects. No information is lost (null space is zero), and no extra space is created; the data just "turns" to face a new direction.

Think of holding a map upside down. The information is all there. The distances are correct and the landmarks exist. But it is useless for navigation because the orientation doesn't match your reality. You need to rotate it.

A square matrix performs this re-orientation. It mixes the independent channels of the vector, essentially saying: "Don't look at Feature 1 and Feature 2 in isolation; look at their sum and their difference." It acts as a switchboard or a mixing desk, routing information from where it was computed to where it is needed next. In practice, we use this to integrate information, taking distinct, segregated reports (like "Subject is John" and "Verb is Run") and mixing them into a single, unified meaning.

1.6.4 Summary

As we progress to neural networks, never look at a weight matrix $\mathbf{W}$ as just a bag of numbers. Look at its shape.

If it is shrinking the vector, it is a **Summarizer** forcing the model to decide what matters. If it is growing the vector, it is an **Analyzer** trying to untangle complex relationships. If it is keeping the size, it is a **Translator** reorganizing information for the next step.

This "lens" intuition is more powerful than memorizing formulas because it tells you the *intent* of each component in the architecture.

Chapter 2

Calculus foundations

Learning objectives

After completing this chapter, you will be able to:

- Compute limits and derivatives from first principles
- Apply the chain rule to composite functions
- Calculate partial derivatives and gradients for multivariable functions
- Derive the backpropagation algorithm using the chain rule
- Perform matrix calculus operations needed for neural network training

2.1 Limits: the idea of approaching

Before we can understand derivatives, we need to understand **limits**. A limit captures the idea of "approaching" a value, even if we never actually reach it.

Consider the function $f(x) = \frac{x^2-1}{x-1}$. What happens at $x = 1$? If we try to plug in $x = 1$, we get $\frac{0}{0}$, which is undefined. But let's see what happens as x *approaches* 1:

x	$f(x) = \frac{x^2-1}{x-1}$
0.9	1.9
0.99	1.99
0.999	1.999
1.001	2.001
1.01	2.01
1.1	2.1

As x gets closer to 1, $f(x)$ gets closer to 2. We write this as:

$$\lim_{x \to 1} \frac{x^2 - 1}{x - 1} = 2$$

The function never equals 2 at $x = 1$ (it's undefined there), but it approaches 2 arbitrarily closely. We can verify this algebraically: $\frac{x^2-1}{x-1} = \frac{(x-1)(x+1)}{x-1} = x + 1$ for $x \neq 1$, and $x + 1 \to 2$ as $x \to 1$.

Why do limits matter? Because many important quantities can't be computed directly but can be defined as limits. The most important example is instantaneous rate of change.

2.2 Functions and derivatives

A function $f : \mathbb{R} \to \mathbb{R}$ maps real numbers to real numbers. Suppose we want to know how fast f is changing at a specific point x.

For average rate of change, we pick two points and compute the slope between them:

$$\text{average rate of change} = \frac{f(x+h) - f(x)}{h}$$

This is the slope of the **secant line** connecting $(x, f(x))$ and $(x+h, f(x+h))$. But what if we want the *instantaneous* rate of change at exactly x? We can't use $h = 0$ because that gives $\frac{0}{0}$. Instead, we take the limit as h approaches 0:

$$\frac{df}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

This is the **derivative** of f at x. Let's compute a concrete example. For $f(x) = x^2$, what is the derivative at $x = 3$?

$$\left.\frac{df}{dx}\right|_{x=3} = \lim_{h \to 0} \frac{(3+h)^2 - 3^2}{h} = \lim_{h \to 0} \frac{9 + 6h + h^2 - 9}{h} = \lim_{h \to 0} \frac{6h + h^2}{h} = \lim_{h \to 0} (6 + h) = 6$$

Let's verify this makes sense by computing the average rate of change for smaller and smaller h:

h	$\frac{(3+h)^2-9}{h}$
1	$\frac{16-9}{1} = 7$
0.1	$\frac{9.61-9}{0.1} = 6.1$
0.01	$\frac{9.0601-9}{0.01} = 6.01$
0.001	$\frac{9.006001-9}{0.001} = 6.001$

As h shrinks, the average rate approaches 6. The derivative captures the instantaneous rate of change: at $x = 3$, the function $f(x) = x^2$ is increasing at a rate of 6 units of output per unit of input.

Geometrically, the derivative is the slope of the **tangent line** to the curve at that point. As $h \to 0$, the secant line rotates and becomes the tangent line.

For general x, we can derive that $\frac{d}{dx}x^2 = 2x$. At $x = 3$, this gives $2 \cdot 3 = 6$, confirming our calculation.

Some essential derivatives to know:

$$\frac{d}{dx}x^n = nx^{n-1} \tag{2.1}$$

$$\frac{d}{dx}e^x = e^x \tag{2.2}$$

$$\frac{d}{dx}\ln x = \frac{1}{x} \tag{2.3}$$

$$\frac{d}{dx}\sin x = \cos x \tag{2.4}$$

$$\frac{d}{dx}\cos x = -\sin x \tag{2.5}$$

2.3 The chain rule: derivatives of composed functions

The **chain rule** is the most important rule in calculus for machine learning. Neural networks are compositions of many functions, and the chain rule tells us how to differentiate through all of them. Let's build deep intuition for why it works.

Suppose we have two functions composed: $y = f(u)$ where $u = g(x)$. So x feeds into g to produce u, then u feeds into f to produce y:

$$x \xrightarrow{g} u \xrightarrow{f} y$$

We want $\frac{dy}{dx}$: how does y change when we change x? The chain rule says:

$$\frac{dy}{dx} = \frac{dy}{du} \cdot \frac{du}{dx}$$

Why does multiplication make sense? Think about it in terms of small changes. Suppose:

- A small change Δx in x causes a change Δu in u
- That change Δu in u causes a change Δy in y

The ratio $\frac{\Delta u}{\Delta x}$ tells us how much u changes per unit change in x. The ratio $\frac{\Delta y}{\Delta u}$ tells us how much y changes per unit change in u. To find how much y changes per unit change in x, we multiply:

$$\frac{\Delta y}{\Delta x} = \frac{\Delta y}{\Delta u} \cdot \frac{\Delta u}{\Delta x}$$

The Δu terms "cancel" (though this is intuition, not rigorous proof). Taking the limit as all changes become infinitesimally small gives the chain rule.

Concrete example. Let $y = (3x + 1)^2$. We can write this as $y = u^2$ where $u = 3x + 1$. Then:

- $\frac{du}{dx} = 3$ (the inner function's derivative)
- $\frac{dy}{du} = 2u = 2(3x + 1)$ (the outer function's derivative)

By the chain rule:

$$\frac{dy}{dx} = \frac{dy}{du} \cdot \frac{du}{dx} = 2(3x + 1) \cdot 3 = 6(3x + 1)$$

Let's verify with numbers. At $x = 2$: $u = 3(2) + 1 = 7$, so $y = 49$.

- If x increases by a tiny amount $\Delta x = 0.001$, then u becomes $3(2.001) + 1 = 7.003$
- The change in u is $\Delta u = 0.003$, so $\frac{\Delta u}{\Delta x} = \frac{0.003}{0.001} = 3$
- With $u = 7.003$, y becomes $(7.003)^2 = 49.042009$
- The change in y is $\Delta y = 0.042009$, so $\frac{\Delta y}{\Delta u} = \frac{0.042009}{0.003} \approx 14.003 \approx 2u$
- The total rate: $\frac{\Delta y}{\Delta x} = \frac{0.042009}{0.001} = 42.009$

Our formula predicts $\frac{dy}{dx} = 6(3 \cdot 2 + 1) = 6 \cdot 7 = 42$. The numerical calculation gives 42.009, approaching 42 as $\Delta x \to 0$.

The amplification interpretation. Here's another way to think about the chain rule. Each function in the chain acts as an "amplifier" for small changes:

- The function g amplifies changes in x by factor $\frac{du}{dx}$
- The function f amplifies changes in u by factor $\frac{dy}{du}$
- The total amplification is the product: $\frac{dy}{du} \cdot \frac{du}{dx}$

If g doubles small changes (derivative = 2) and f triples small changes (derivative = 3), then the composition multiplies small changes by $2 \times 3 = 6$.

Longer chains. The chain rule extends naturally. If $y = f(u)$, $u = g(v)$, $v = h(x)$, then:

$$\frac{dy}{dx} = \frac{dy}{du} \cdot \frac{du}{dv} \cdot \frac{dv}{dx}$$

Each link in the chain contributes a multiplicative factor. This is exactly how backpropagation works in neural networks: we multiply the local derivatives at each layer to get the total derivative from output to input.

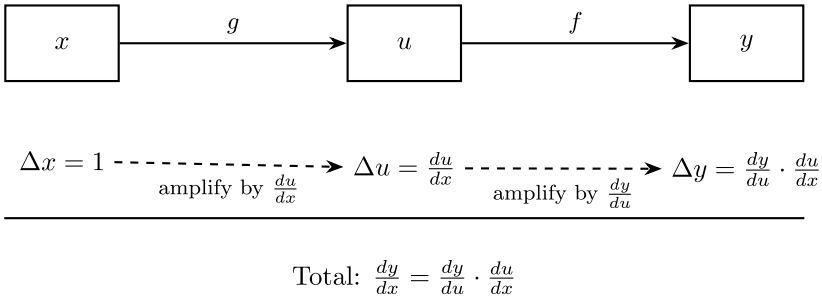

Figure 2.1: The chain rule as amplification. A small change in x gets amplified by $\frac{du}{dx}$ to become a change in u, then amplified again by $\frac{dy}{du}$ to become a change in y. The total amplification is the product of individual amplifications.

Why the chain rule matters for neural networks. Consider a simple neural network: input x, hidden layer $h = \sigma(wx + b)$, output $y = vh$. To train this network, we need $\frac{dy}{dw}$: how does the output change when we adjust the weight w?

Using the chain rule:

$$\frac{dy}{dw} = \frac{dy}{dh} \cdot \frac{dh}{dw} = v \cdot \frac{d\sigma}{dw}$$

And $\frac{d\sigma}{dw}$ requires another application of the chain rule (since σ depends on w through its argument $wx + b$). This cascading application of the chain rule through all layers is backpropagation.

2.4 Other differentiation rules

The **product rule** says: if $y = f(x) \cdot g(x)$, then:

$$\frac{dy}{dx} = \frac{df}{dx} \cdot g(x) + f(x) \cdot \frac{dg}{dx}$$

The intuition: when both f and g depend on x, changing x affects y through both paths. The first term captures the effect of f changing while g stays fixed; the second captures g changing while f stays fixed.

The **quotient rule** says: if $y = \frac{f(x)}{g(x)}$, then:

$$\frac{dy}{dx} = \frac{\frac{df}{dx} \cdot g(x) - f(x) \cdot \frac{dg}{dx}}{g(x)^2}$$

2.5 Partial derivatives and gradients

2.5.1 The problem: optimizing functions of many variables

Here's the situation that motivates everything in this section. You have a neural network with millions of parameters (weights). You have a loss function that measures how wrong the network's predictions are. The loss depends on all those parameters:

$$L = L(w_1, w_2, \dots, w_{1000000})$$

You want to find parameter values that make the loss small. How do you do it?

You can't just try all possible combinations. With a million parameters, even if each could take only 10 values, you'd have $10^{1000000}$ combinations to check. The universe doesn't have enough atoms for that.

Instead, you need a smarter strategy: **start somewhere, figure out which direction is downhill, take a step that way, repeat**. This is gradient descent, and it requires answering a key question: given where you are, which direction reduces the loss most quickly?

That question is what partial derivatives and gradients answer.

2.5.2 Thinking geometrically: functions as landscapes

To build intuition, start with a function of just two variables: $f(x, y)$. We can visualize this as a surface in 3D space, where the height at point (x, y) is $f(x, y)$.

Imagine standing on a hillside. The ground beneath you is the surface $z = f(x, y)$. You're at position (x_0, y_0) at height $f(x_0, y_0)$. You want to walk downhill as quickly as possible.

Which way should you go?

You could walk due east (increasing x, keeping y fixed). How steep is that? The slope in the east direction is $\frac{\partial f}{\partial x}$, the **partial derivative with respect to** x.

You could walk due north (increasing y, keeping x fixed). How steep is that? The slope in the north direction is $\frac{\partial f}{\partial y}$, the **partial derivative with respect to** y.

But you're not limited to walking along coordinate axes. You could walk northeast, or any direction. The **gradient** ∇f tells you the direction of steepest ascent. To go downhill fastest, walk in the direction $-\nabla f$.

2.5.3 Partial derivatives: slopes along coordinate axes

The partial derivative $\frac{\partial f}{\partial x}$ answers: "If I move only in the x direction, how fast does f change?"

Mechanically, you compute it by treating all other variables as constants and differentiating with respect to x. But the meaning is geometric: it's the slope of the surface in the x direction.

Consider $f(x, y) = x^2 + y^2$. This is a paraboloid, a bowl opening upward with its bottom at the origin.

$$\frac{\partial f}{\partial x} = 2x, \quad \frac{\partial f}{\partial y} = 2y$$

At the point $(3, 4)$:

- $\frac{\partial f}{\partial x} = 6$: walking east, you're climbing at slope 6
- $\frac{\partial f}{\partial y} = 8$: walking north, you're climbing at slope 8

At the origin $(0, 0)$:

- $\frac{\partial f}{\partial x} = 0$, $\frac{\partial f}{\partial y} = 0$: the surface is flat in both directions

This makes sense. The paraboloid has its minimum at the origin, where the surface is horizontal. As you move away from the origin, the slopes increase.

The formal definition is:

$$\frac{\partial f}{\partial x} = \lim_{h \to 0} \frac{f(x+h, y) - f(x, y)}{h}$$

Notice that y doesn't change. We're measuring the slope along a slice where y is held constant.

2.5.4 The gradient: the compass pointing uphill

The **gradient** combines all partial derivatives into a vector:

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix}$$

For $f(x, y) = x^2 + y^2$:

$$\nabla f = \begin{bmatrix} 2x \\ 2y \end{bmatrix}$$

At the point $(3, 4)$: $\nabla f = \begin{bmatrix} 6 \\ 8 \end{bmatrix}$.

This vector has a profound meaning: **it points in the direction (in the x-y plane) of steepest ascent**. The gradient lives in the input space, not in 3D. It tells you which way to walk horizontally to climb most steeply.

Why? At first this might seem wrong. Going north gives slope 8. Going east gives slope 6. Isn't north the best direction? Why would combining them help?

The key insight: when you walk diagonally, you're not diluting the good direction with the bad one. You're **harvesting height gain from both directions simultaneously**.

Let's compute the height gain for a unit step in each direction:

- **North** $[0, 1]$: Move 0 in x, 1 in y. Height gain $= 0 \times 6 + 1 \times 8 = 8$
- **East** $[1, 0]$: Move 1 in x, 0 in y. Height gain $= 1 \times 6 + 0 \times 8 = 6$
- **Gradient direction** $[0.6, 0.8]$: Move 0.6 in x, 0.8 in y. Height gain $= 0.6 \times 6 + 0.8 \times 8 = 3.6 + 6.4 = 10$

The diagonal step gets 3.6 from the x-component AND 6.4 from the y-component. These contributions add up to more than either pure direction alone.

But why is $[0.6, 0.8]$ the best proportion? Why not $[0.5, 0.5]$ or $[0.1, 0.9]$?

We want to maximize height gain $= 6u_1 + 8u_2$, subject to taking a unit step: $u_1^2 + u_2^2 = 1$.

The constraint is crucial. We have a "budget" of one unit of movement to allocate between x and y. But the budget is *circular* (Euclidean), not linear. This matters.

If the budget were linear ($u_1 + u_2 = 1$), we'd put everything into y since it pays better. But with a circular budget, we can do something clever.

Consider what happens as we tilt from north toward the gradient direction:

- $[0, 1]$: Height gain $= 0 \times 6 + 1 \times 8 = 8$
- $[0.6, 0.8]$: Height gain $= 0.6 \times 6 + 0.8 \times 8 = 10$

By giving up only 0.2 units of y-movement (from 1 to 0.8), we gain 0.6 units of x-movement. The circular constraint means small sacrifices in one direction buy disproportionately large gains in the other.

Why does $[0.6, 0.8]$ hit the sweet spot? Because it's proportional to the payoffs $[6, 8]$. The direction that maximizes $6u_1 + 8u_2$ on the unit circle points the same way as $[6, 8]$ itself. Normalizing: $[6, 8]/\sqrt{36 + 64} = [6, 8]/10 = [0.6, 0.8]$.

The gradient automatically encodes the right trade-off. Larger partial derivative means that direction is more valuable, so we tilt toward it, in exact proportion to how much more valuable it is.

What we just computed has a name: the **directional derivative**. The slope in direction $\mathbf{u}$ is:

$$D_{\mathbf{u}}f = \nabla f \cdot \mathbf{u} = \frac{\partial f}{\partial x}u_1 + \frac{\partial f}{\partial y}u_2$$

This formula captures exactly what we did: multiply each partial derivative by how much we move in that direction, then add up the contributions. The dot product is just a compact way to write "weight each slope by the corresponding component of $\mathbf{u}$, then sum."

Since the directional derivative is a dot product, we can use the geometric formula: $\nabla f \cdot \mathbf{u} = \|\nabla f\| \cos\theta$, where θ is the angle between them. This is maximized when $\theta = 0$ (vectors aligned), giving maximum slope $\|\nabla f\|$. For our example: $\sqrt{6^2 + 8^2} = 10$, exactly what we found.

One more observation: the gradient $[6, 8]$ points directly away from the origin. On our bowl-shaped paraboloid, the steepest way up is radially outward. The gradient captures this geometric fact automatically.

For gradient descent, we go the opposite way: $-\nabla f = [-6, -8]$ points toward the origin, toward the minimum. This is why gradient descent works.

2.5.5 Why this matters: navigating loss landscapes

Now extend this to a function of millions of variables. You can't visualize it, but the mathematics works the same way.

The loss function $L(w_1, w_2, \ldots, w_n)$ defines a "landscape" in n-dimensional space. You're at some point (current parameter values). The gradient ∇L tells you which direction increases the loss most rapidly.

To reduce the loss, move in direction $-\nabla L$:

$$\mathbf{w}_{\text{new}} = \mathbf{w}_{\text{old}} - \eta \nabla L$$

where η is the learning rate (step size).

This is the entire basis of training neural networks. Billions of dollars of compute, running gradient descent on loss landscapes with billions of dimensions.

2.5.6 The chain rule: tracking influence through layers

Neural networks are compositions of functions. The input passes through layer 1, then layer 2, then layer 3, and so on. The loss depends on the output, which depends on the intermediate layers, which depend on the parameters.

How does a small change in a parameter w in layer 1 affect the final loss?

The change propagates forward through all subsequent layers. We need to track this chain of influence.

Single path. If x affects y, and y affects z, and these are the only connections:

$$x \to y \to z$$

Then $\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}$. We multiply the rates of change along the path.

Multiple paths. But often x can influence z through multiple intermediate variables:

$$x \to \begin{cases} y_1 & \to z \\ y_2 & \end{cases}$$

Here x affects y_1 and y_2, and both affect z. A small change Δx causes:

- Change in y_1: $\Delta y_1 \approx \frac{dy_1}{dx}\Delta x$
- Change in y_2: $\Delta y_2 \approx \frac{dy_2}{dx}\Delta x$

These changes independently affect z:

- Effect through y_1: $\frac{\partial z}{\partial y_1}\Delta y_1$
- Effect through y_2: $\frac{\partial z}{\partial y_2}\Delta y_2$

Total effect: sum them up.

$$\frac{dz}{dx} = \frac{\partial z}{\partial y_1}\frac{dy_1}{dx} + \frac{\partial z}{\partial y_2}\frac{dy_2}{dx}$$

The general rule: multiply along each path (chain rule), then sum over all paths.

Why sum? Because the effects through different paths are independent and additive. Changing x slightly perturbs both y_1 and y_2, and z feels both perturbations. There's no interaction between the paths at the linear (first-order) level.

Why multiply along a path? Because each link in the chain amplifies or attenuates the change. If y doubles when x increases by 1, and z triples when y increases by 1, then z increases by $2 \times 3 = 6$ when x increases by 1.

2.5.7 The Jacobian: all derivatives at once

When we have vector inputs $\mathbf{x} \in \mathbb{R}^n$ and vector outputs $\mathbf{y} \in \mathbb{R}^m$, we need $m \times n$ partial derivatives: how does each output depend on each input?

The **Jacobian matrix** organizes all of these:

$$\mathbf{J} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix}$$

Row j contains the gradient of y_j. Column i tells how all outputs respond to input x_i.

Concrete example. Consider a function $\mathbf{y} = f(\mathbf{x})$ where:

$$y_1 = x_1^2 + x_2, \quad y_2 = x_1 x_2$$

The Jacobian is:

$$\mathbf{J} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} \end{bmatrix} = \begin{bmatrix} 2x_1 & 1 \\ x_2 & x_1 \end{bmatrix}$$

At the point $(x_1, x_2) = (3, 2)$, the outputs are $y_1 = 9 + 2 = 11$ and $y_2 = 6$, and the Jacobian is:

$$\mathbf{J} = \begin{bmatrix} 6 & 1 \\ 2 & 3 \end{bmatrix}$$

What does this matrix tell us? It predicts how small input changes affect outputs:

$$\Delta \mathbf{y} \approx \mathbf{J} \Delta \mathbf{x}$$

Suppose we nudge the input by $\Delta \mathbf{x} = [0.1, 0.2]^T$. The Jacobian predicts:

$$\Delta \mathbf{y} \approx \begin{bmatrix} 6 & 1 \\ 2 & 3 \end{bmatrix} \begin{bmatrix} 0.1 \\ 0.2 \end{bmatrix} = \begin{bmatrix} 6(0.1) + 1(0.2) \\ 2(0.1) + 3(0.2) \end{bmatrix} = \begin{bmatrix} 0.8 \\ 0.8 \end{bmatrix}$$

Let's verify. At $(3.1, 2.2)$:

- $y_1 = (3.1)^2 + 2.2 = 9.61 + 2.2 = 11.81$. Change: $11.81 - 11 = 0.81$
- $y_2 = (3.1)(2.2) = 6.82$. Change: $6.82 - 6 = 0.82$

The predictions (0.8, 0.8) are close to the actual changes (0.81, 0.82). The small discrepancy is because the Jacobian gives a *linear* approximation, which is only exact for infinitesimal changes.

The Jacobian is the multivariate generalization of the derivative. Just as $\frac{df}{dx}$ tells us how a scalar function responds to a scalar input, the Jacobian tells us how a vector function responds to a vector input.

2.5.8 Backpropagation: the chain rule in matrix form

Consider a neural network as a chain:

$$\mathbf{x} \xrightarrow{\text{layer 1}} \mathbf{h}_1 \xrightarrow{\text{layer 2}} \mathbf{h}_2 \xrightarrow{\cdots} \mathbf{h}_L \xrightarrow{\text{loss}} L$$

We want $\nabla_{\mathbf{x}} L$: how does the loss depend on the input (or on parameters at each layer)?

Working backward from the loss:

- $\nabla_{\mathbf{h}_L} L$ is the gradient at the last hidden layer
- $\nabla_{\mathbf{h}_{L-1}} L = \mathbf{J}_L^T \nabla_{\mathbf{h}_L} L$, where $\mathbf{J}_L$ is the Jacobian of layer L
- $\nabla_{\mathbf{h}_{L-2}} L = \mathbf{J}_{L-1}^T \nabla_{\mathbf{h}_{L-1}} L$
- … and so on back to the input

Concrete example. Let's trace through a tiny two-layer network:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \xrightarrow{\text{layer 1}} \mathbf{h} = \begin{bmatrix} h_1 \\ h_2 \end{bmatrix} \xrightarrow{\text{layer 2}} L$$

where $h_1 = x_1 + x_2$, $h_2 = x_1 - x_2$, and $L = h_1^2 + h_2^2$.

At $\mathbf{x} = [3, 1]^T$: $\mathbf{h} = [4, 2]^T$ and $L = 16 + 4 = 20$.

Step 1: Gradient at output. How does L depend on $\mathbf{h}$?

$$\nabla_{\mathbf{h}} L = \begin{bmatrix} \frac{\partial L}{\partial h_1} \\ \frac{\partial L}{\partial h_2} \end{bmatrix} = \begin{bmatrix} 2h_1 \\ 2h_2 \end{bmatrix} = \begin{bmatrix} 8 \\ 4 \end{bmatrix}$$

Step 2: Jacobian of layer 1. How does $\mathbf{h}$ depend on $\mathbf{x}$?

$$\mathbf{J} = \begin{bmatrix} \frac{\partial h_1}{\partial x_1} & \frac{\partial h_1}{\partial x_2} \\ \frac{\partial h_2}{\partial x_1} & \frac{\partial h_2}{\partial x_2} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

Step 3: Backpropagate. Multiply by $\mathbf{J}^T$:

$$\nabla_{\mathbf{x}} L = \mathbf{J}^T \nabla_{\mathbf{h}} L = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 8 \\ 4 \end{bmatrix} = \begin{bmatrix} 8+4 \\ 8-4 \end{bmatrix} = \begin{bmatrix} 12 \\ 4 \end{bmatrix}$$

Let's verify by direct calculation. Substituting: $L = (x_1 + x_2)^2 + (x_1 - x_2)^2 = 2x_1^2 + 2x_2^2$.

$$\nabla_{\mathbf{x}} L = \begin{bmatrix} 4x_1 \\ 4x_2 \end{bmatrix} = \begin{bmatrix} 12 \\ 4 \end{bmatrix} \quad \checkmark$$

Why the transpose? Look at row 1 of the calculation: $(\nabla_{\mathbf{x}} L)_1 = 1 \cdot 8 + 1 \cdot 4 = 12$. This is:

$$\frac{\partial L}{\partial x_1} = \frac{\partial L}{\partial h_1}\frac{\partial h_1}{\partial x_1} + \frac{\partial L}{\partial h_2}\frac{\partial h_2}{\partial x_1}$$

We're summing over the intermediate variables h_1, h_2. This sum is a dot product of $\nabla_{\mathbf{h}} L$ with column 1 of $\mathbf{J}$, which equals row 1 of $\mathbf{J}^T$ times $\nabla_{\mathbf{h}} L$. That's why we use the transpose.

The gradient flows backward through the network, getting transformed by each layer's Jacobian transpose. This is backpropagation.

The key insight: we multiply Jacobians across layers (because effects compound through the chain), and we sum within each Jacobian-vector product (because each input affects multiple intermediates, and we must account for all paths).

This interplay of multiplication and summation, flowing backward through the network, is how neural networks learn.

2.6 Matrix calculus

When we differentiate with respect to vectors and matrices, we need to be careful about dimensions. If $f : \mathbb{R}^n \to \mathbb{R}$ and $\mathbf{x} \in \mathbb{R}^n$, then $\nabla_{\mathbf{x}} f$ is an n-dimensional column vector. If $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$ and $\mathbf{x} \in \mathbb{R}^n$, then the Jacobian $\mathbf{J} = \nabla_{\mathbf{x}} \mathbf{f}$ is $m \times n$.

Here are some useful formulas, each with a concrete example. Let $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{a} \in \mathbb{R}^n$, $\mathbf{A} \in \mathbb{R}^{n \times n}$.

Formula 1: $\nabla_{\mathbf{x}}(\mathbf{a}^T\mathbf{x}) = \mathbf{a}$

This says: the gradient of a linear function is constant. The function $\mathbf{a}^T\mathbf{x} = a_1x_1 + a_2x_2 + \cdots$ increases at rate a_i per unit increase in x_i, regardless of where you are.

Example: Let $\mathbf{a} = [3, 2]^T$ and $f(\mathbf{x}) = \mathbf{a}^T\mathbf{x} = 3x_1 + 2x_2$.

$$\nabla f = \begin{bmatrix} \frac{\partial}{\partial x_1}(3x_1 + 2x_2) \\ \frac{\partial}{\partial x_2}(3x_1 + 2x_2) \end{bmatrix} = \begin{bmatrix} 3 \\ 2 \end{bmatrix} = \mathbf{a} \quad \checkmark$$

Formula 2: $\nabla_{\mathbf{x}}(\mathbf{x}^T\mathbf{x}) = 2\mathbf{x}$

This is the gradient of the squared norm. The function $\mathbf{x}^T\mathbf{x} = x_1^2 + x_2^2 + \cdots$ is a paraboloid centered at the origin.

Example: Let $\mathbf{x} = [3, 4]^T$. Then $f = \mathbf{x}^T\mathbf{x} = 9 + 16 = 25$.

$$\nabla f = 2\mathbf{x} = \begin{bmatrix} 6 \\ 8 \end{bmatrix}$$

This is exactly the gradient we computed earlier for $f(x, y) = x^2 + y^2$ at $(3, 4)$!

Formula 3: $\nabla_{\mathbf{x}}(\mathbf{x}^T\mathbf{A}\mathbf{x}) = (\mathbf{A} + \mathbf{A}^T)\mathbf{x}$

This is the gradient of a quadratic form. If $\mathbf{A}$ is symmetric ($\mathbf{A} = \mathbf{A}^T$), this simplifies to $2\mathbf{A}\mathbf{x}$.

Example: Let $\mathbf{A} = \begin{bmatrix} 1 & 2 \\ 0 & 3 \end{bmatrix}$ and $\mathbf{x} = [1, 1]^T$.

First, compute $f = \mathbf{x}^T\mathbf{A}\mathbf{x}$:

$$\mathbf{A}\mathbf{x} = \begin{bmatrix} 1 & 2 \\ 0 & 3 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 3 \\ 3 \end{bmatrix}, \quad f = \mathbf{x}^T(\mathbf{A}\mathbf{x}) = [1, 1] \begin{bmatrix} 3 \\ 3 \end{bmatrix} = 6$$

Now the gradient. We have $\mathbf{A} + \mathbf{A}^T = \begin{bmatrix} 1 & 2 \\ 0 & 3 \end{bmatrix} + \begin{bmatrix} 1 & 0 \\ 2 & 3 \end{bmatrix} = \begin{bmatrix} 2 & 2 \\ 2 & 6 \end{bmatrix}$.

$$\nabla f = (\mathbf{A} + \mathbf{A}^T)\mathbf{x} = \begin{bmatrix} 2 & 2 \\ 2 & 6 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 4 \\ 8 \end{bmatrix}$$

Let's verify by expanding f and differentiating directly. $f = \mathbf{x}^T\mathbf{A}\mathbf{x} = x_1(x_1+2x_2)+x_2(3x_2) = x_1^2+2x_1x_2+3x_2^2$.

$$\frac{\partial f}{\partial x_1} = 2x_1 + 2x_2 = 4, \quad \frac{\partial f}{\partial x_2} = 2x_1 + 6x_2 = 8 \quad \checkmark$$

Formula 4: $\nabla_{\mathbf{x}}\|\mathbf{x}\|_2 = \frac{\mathbf{x}}{\|\mathbf{x}\|_2}$

This is the gradient of the norm itself (not squared). It points radially outward with unit length.

Example: Let $\mathbf{x} = [3, 4]^T$. Then $\|\mathbf{x}\|_2 = 5$.

$$\nabla\|\mathbf{x}\|_2 = \frac{1}{5} \begin{bmatrix} 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 0.6 \\ 0.8 \end{bmatrix}$$

This is a unit vector pointing in the direction of $\mathbf{x}$. The norm increases at rate 1 per unit step directly away from the origin.

2.7 Important activation functions

Several nonlinear functions appear repeatedly in neural networks. The **sigmoid** function is:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

It squashes inputs to the range $(0, 1)$. The derivative is:

$$\frac{d\sigma}{dx} = \sigma(x)(1 - \sigma(x))$$

To prove this, write $\sigma(x) = (1 + e^{-x})^{-1}$ and use the chain rule:

$$\frac{d\sigma}{dx} = -(1+e^{-x})^{-2} \cdot (-e^{-x}) = \frac{e^{-x}}{(1+e^{-x})^2} = \frac{1}{1+e^{-x}} \cdot \frac{e^{-x}}{1+e^{-x}} = \sigma(x)(1-\sigma(x))$$

The **hyperbolic tangent** is:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

It squashes inputs to $(-1, 1)$. The derivative is:

$$\frac{d}{dx}\tanh(x) = 1 - \tanh^2(x)$$

The **ReLU** (rectified linear unit) is:

$$\text{ReLU}(x) = \max(0, x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$$

The derivative is:

$$\frac{d}{dx}\text{ReLU}(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x < 0 \\ \text{undefined} & \text{if } x = 0 \end{cases}$$

In practice we define it to be 0 or 1 at $x = 0$. ReLU is simple but effective, and it doesn't saturate for positive values.

The **GELU** (Gaussian error linear unit) is:

$$\text{GELU}(x) = x \cdot \Phi(x)$$

where $\Phi(x) = \frac{1}{2}[1 + \text{erf}(x/\sqrt{2})]$ is the cumulative distribution function of the standard normal. GELU is commonly used in transformers like GPT. It's smoother than ReLU and has nice probabilistic interpretations.

2.8 The softmax function

The **softmax** function is crucial for transformers. It maps a vector $\mathbf{z} \in \mathbb{R}^n$ to a probability distribution:

$$\text{softmax}(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}}$$

The outputs are positive and sum to 1, so they can be interpreted as probabilities. Softmax is a smooth, differentiable approximation to the argmax function. When one component of $\mathbf{z}$ is much larger than the others, softmax puts almost all probability mass there.

The Jacobian of softmax has a special structure. Let $\mathbf{p} = \text{softmax}(\mathbf{z})$. Then:

$$\frac{\partial p_i}{\partial z_j} = \begin{cases} p_i(1 - p_i) & \text{if } i = j \\ -p_i p_j & \text{if } i \neq j \end{cases}$$

In matrix form:

$$\mathbf{J} = \text{diag}(\mathbf{p}) - \mathbf{p}\mathbf{p}^T$$

Let's derive this. Start with:

$$p_i = \frac{e^{z_i}}{\sum_k e^{z_k}}$$

For $i = j$:

$$\frac{\partial p_i}{\partial z_i} = \frac{e^{z_i} \sum_k e^{z_k} - e^{z_i} \cdot e^{z_i}}{(\sum_k e^{z_k})^2} = \frac{e^{z_i}}{\sum_k e^{z_k}} \cdot \frac{\sum_k e^{z_k} - e^{z_i}}{\sum_k e^{z_k}} = p_i(1 - p_i)$$

For $i \neq j$:

$$\frac{\partial p_i}{\partial z_j} = \frac{0 \cdot \sum_k e^{z_k} - e^{z_i} \cdot e^{z_j}}{(\sum_k e^{z_k})^2} = -\frac{e^{z_i}}{\sum_k e^{z_k}} \cdot \frac{e^{z_j}}{\sum_k e^{z_k}} = -p_i p_j$$

In attention mechanisms, we apply softmax to similarity scores to get attention weights. Understanding how gradients flow through softmax is essential for understanding how attention is learned.

Chapter 3

Probability basics

Learning objectives

After completing this chapter, you will be able to:

- Distinguish between discrete and continuous random variables
- Calculate expectations, variances, and standard deviations
- Apply Bayes' theorem to update beliefs given evidence
- Compute KL divergence and cross-entropy between distributions
- Understand why cross-entropy is the natural loss function for classification

Probability is the language of uncertainty. In machine learning, we use it everywhere: our model's output is a probability distribution over possible next tokens, our training objective measures how well predicted probabilities match observed data, and our understanding of why networks work at all relies on probabilistic reasoning.

This chapter develops probability from the ground up, focusing on the concepts that matter most for understanding transformers.

3.1 A note on notation

In linear algebra, we used **bold uppercase** for matrices ($\mathbf{W}$, $\mathbf{X}$) and **bold lowercase** for vectors ($\mathbf{v}$, $\mathbf{x}$).

Probability introduces a different convention: **random variables** are written in *italic uppercase* without bold: X, Y, Z. This is standard notation in probability theory.

Symbol	Meaning	Example
$\mathbf{X}$	Matrix (bold)	A 3×3 array of weights
$\mathbf{x}$	Vector (bold lowercase)	A point in 3D space: $[1, 2, 3]^T$
X	Random variable (italic, not bold)	The outcome of rolling a die
x	A specific value (lowercase)	The number 4

A random variable isn't a number—it's a placeholder for an outcome that hasn't been determined yet. Think of it as the *question* rather than the *answer*.

Concrete example: You're about to roll a die.

- Before rolling: X represents the uncertain outcome. We don't know what it will be yet. X could be 1, 2, 3, 4, 5, or 6.

- After rolling: You got a 4. Now $x = 4$ is the specific value that occurred.

The notation $P(X = 4)$ asks: "Before rolling, what's the probability that X will equal 4?" For a fair die, $P(X = 4) = \frac{1}{6}$.

We write $p(x)$ as shorthand for $P(X = x)$—the probability that the random variable X takes value x. So for our die:

- $p(1) = P(X = 1) = \frac{1}{6}$
- $p(2) = P(X = 2) = \frac{1}{6}$
- …
- $p(6) = P(X = 6) = \frac{1}{6}$

Another example: Let Y be "the next word a language model predicts."

- Y is the random variable—the uncertain outcome
- $y =$ "cat" is one possible value
- $P(Y =$ "cat"$) = 0.15$ means the model assigns 15% probability to "cat"
- $p($"cat"$) = 0.15$ is the same thing, shorter notation

3.2 Why probability for neural networks?

Consider what a language model does. Given the input "The cat sat on the", what should the model output? Not a single word—that would be too confident. The model should express *uncertainty*: "mat" is likely, "floor" is possible, "elephant" is unlikely but not impossible.

The model outputs a **probability distribution** over the vocabulary:

Word	Probability
mat	0.35
floor	0.20
rug	0.15
bed	0.10
…	…
elephant	0.0001

These probabilities must be non-negative and sum to 1. They express the model's beliefs about what comes next.

Training the model means adjusting its parameters so that its predicted probabilities align with reality. If the true next word was "mat," we want the model to assign high probability to "mat." This is where the loss function comes in—but to understand loss functions, we need to understand probability properly.

3.3 Probability as a measure of belief

What does "$P($mat$) = 0.35$" mean? There are two interpretations:

Frequentist: If we saw "The cat sat on the" many times, about 35% of the time the next word would be "mat."

Bayesian: The model assigns a 35% degree of belief to "mat" being the next word.

For neural networks, the Bayesian interpretation is more useful. The model doesn't have access to infinite repetitions—it has seen some training data and formed beliefs based on it.

The key constraint on probabilities is that they must form a **valid distribution**:

1. Every probability is non-negative: $P(x) \geq 0$ for all x
2. Probabilities sum to 1: $\sum_x P(x) = 1$

These aren't arbitrary rules—they ensure probabilities behave sensibly. If we're certain one thing will happen, we assign it probability 1 and everything else probability 0. If we're completely uncertain among n options, we assign each probability $1/n$.

3.4 Discrete vs continuous

A **discrete** random variable takes countably many values (like words in a vocabulary). A **continuous** random variable takes values in a continuum (like the real numbers).

For discrete variables, we have a **probability mass function** (PMF):

$$p(x) = P(X = x)$$

Each value x has a specific probability.

Example: A fair die has $p(1) = p(2) = \cdots = p(6) = \frac{1}{6}$.

For continuous variables, we can't assign probability to individual points (there are infinitely many). Instead, we have a **probability density function** (PDF) $f(x)$, and probability comes from integration:

$$P(a \leq X \leq b) = \int_a^b f(x)\,dx$$

The density $f(x)$ tells us how "concentrated" probability is near x. High density means outcomes near x are more likely; low density means they're less likely.

In transformers, we mostly work with discrete distributions (over vocabulary tokens), but continuous distributions appear in weight initialization and analysis.

3.5 The mean (average)

You already know what an average is. Roll a die 6 times, get 2, 5, 3, 6, 1, 4, add them up, divide by 6:

$$\text{average} = \frac{2+5+3+6+1+4}{6} = \frac{21}{6} = 3.5$$

But what if we haven't rolled yet? We can still compute the *theoretical* average—what we'd expect to get if we rolled many times. For a fair die, each number 1–6 is equally likely (probability $\frac{1}{6}$), so:

$$\text{mean} = 1 \times \frac{1}{6} + 2 \times \frac{1}{6} + 3 \times \frac{1}{6} + 4 \times \frac{1}{6} + 5 \times \frac{1}{6} + 6 \times \frac{1}{6} = 3.5$$

Each outcome contributes its value times its probability. This is a **weighted average**, where the weights are probabilities.

In statistics, this theoretical average is called the **expectation** or **expected value**, written $\mathbb{E}[X]$:

Friendly term	Formal term	Notation
Mean, average	Expectation	$\mathbb{E}[X]$

The formula:

$$\text{mean} = \mathbb{E}[X] = \sum_{\text{all outcomes } x} x \times P(x)$$

Or more compactly: $\mathbb{E}[X] = \sum_x x \cdot p(x)$

Example: loaded die

Suppose a loaded die has: - 50% chance of rolling 6 - 10% chance each for 1, 2, 3, 4, 5

$$\begin{aligned}\text{mean} &= 1 \times 0.1 + 2 \times 0.1 + 3 \times 0.1 + 4 \times 0.1 + 5 \times 0.1 + 6 \times 0.5 \\ &= 0.1 + 0.2 + 0.3 + 0.4 + 0.5 + 3.0 = 4.5\end{aligned}$$

The loaded die's mean (4.5) is higher than the fair die's mean (3.5) because high outcomes are more likely.

Example: language model

A language model predicts the next word with probabilities. Suppose the possible words and their probabilities are:

Word	Probability	"Value" (arbitrary score)
cat	0.4	10
dog	0.3	8
bird	0.2	6
fish	0.1	4

Mean score: $10 \times 0.4 + 8 \times 0.3 + 6 \times 0.2 + 4 \times 0.1 = 4 + 2.4 + 1.2 + 0.4 = 8.0$

Key property: The mean is linear. If you have two uncertain quantities and add them:

$$\text{mean of } (aX + bY) = a \times \text{mean of } X + b \times \text{mean of } Y$$

In formal notation: $\mathbb{E}[aX + bY] = a\mathbb{E}[X] + b\mathbb{E}[Y]$

This works even if X and Y are related to each other. Linearity makes means easy to work with.

3.6 Spread (variance)

The mean tells us where the distribution is centered. But two distributions can have the same mean yet look very different:

- Distribution A: always returns 5
- Distribution B: returns 0 or 10, each with 50% chance

Both have mean 5. But A always gives exactly 5, while B is all over the place. We need a number that measures this **spread**—how far outcomes typically fall from the mean.

Friendly term	Formal term	Notation
Spread	Variance	$\text{Var}(X)$

How should we measure spread?

Attempt 1: average distance from mean

For distribution B, let's compute how far each outcome is from the mean (5): - Outcome 0: distance from mean $= 0 - 5 = -5$ - Outcome 10: distance from mean $= 10 - 5 = +5$

Average distance: $\frac{(-5)+(+5)}{2} = 0$

That's wrong! B is clearly spread out, but we got zero. The problem: positive and negative distances cancelled out.

Attempt 2: average of absolute distances

Let's use absolute values to prevent cancellation: - $|0 - 5| = 5$ - $|10 - 5| = 5$

Average: $\frac{5+5}{2} = 5$

This works! But absolute value has a problem for machine learning: it's not smooth. The function $|x|$ has a sharp corner at $x = 0$, which causes issues for gradient descent (we need derivatives, and corners don't have well-defined derivatives).

Attempt 3: average of squared distances (this is variance)

Instead of absolute value, let's square the distances: - $(0 - 5)^2 = 25$ - $(10 - 5)^2 = 25$

Average: $\frac{25+25}{2} = 25$

This is the **variance**. In words:

$$\text{spread} = \text{average of (distance from mean)}^2$$

In formal notation:

$$\text{Var}(X) = \mathbb{E}[(X - \mathbb{E}[X])^2]$$

Which reads as: "the expected value of the squared deviation from the expected value."

Why squared distances?

1. **No cancellation**: Squares are always positive
2. **Smooth function**: x^2 has derivatives everywhere—perfect for gradient descent
3. **Punishes big deviations**: Being 10 away contributes 100, being 2 away contributes only 4
4. **Adds nicely**: For independent random variables, spread of $(X + Y)$ = spread of X + spread of Y

Computing spread step by step

Distribution A (always 5):

Mean = 5. Only outcome is 5.

$$\text{spread} = (5 - 5)^2 \times 1 = 0$$

Zero spread—every sample equals the mean.

Distribution B (0 or 10 with equal probability):

Mean $= 0 \times 0.5 + 10 \times 0.5 = 5$

$$\text{spread} = (0 - 5)^2 \times 0.5 + (10 - 5)^2 \times 0.5 = 25 \times 0.5 + 25 \times 0.5 = 25$$

Distribution C (outcomes 4, 5, 6 with equal probability):

Mean $= \frac{4+5+6}{3} = 5$

$$\text{spread} = (4-5)^2 \times \frac{1}{3} + (5-5)^2 \times \frac{1}{3} + (6-5)^2 \times \frac{1}{3}$$
$$= 1 \times \frac{1}{3} + 0 \times \frac{1}{3} + 1 \times \frac{1}{3} = \frac{2}{3} \approx 0.67$$

Summary: Three distributions, all with mean 5, but different spreads:

Distribution	Outcomes	Spread (variance)
A	Always 5	0
C	4, 5, or 6	0.67
B	0 or 10	25

A computational shortcut

There's an equivalent formula that's often easier to compute:

$$\text{spread} = \text{Var}(X) = \mathbb{E}[X^2] - (\mathbb{E}[X])^2$$

In words: "mean of squares" minus "square of mean."

Let's verify for distribution B: - Mean: $\mathbb{E}[X] = 5$ - $\mathbb{E}[X^2] = 0^2 \cdot 0.5 + 10^2 \cdot 0.5 = 0 + 50 = 50$ - $\text{Var}(X) = 50 - 5^2 = 50 - 25 = 25$

Standard deviation: The problem with variance is that it's in squared units. If X is in meters, variance is in meters^2. To get back to the original units, take the square root:

$$\text{standard deviation} = \sigma = \sqrt{\text{variance}}$$

For distribution B: $\sigma = \sqrt{25} = 5$

Friendly term	Formal term	Notation
Spread (squared)	Variance	$\text{Var}(X)$
Spread (same units)	Standard deviation	σ or $\text{std}(X)$

Important clarification: Standard deviation is the typical *distance from the mean*, not a typical value you'd see. Distribution B has outcomes 0 and 10, mean 5, standard deviation 5. You'll never actually get a 5—but when you get 0 or 10, you're always exactly 5 away from the mean. That's what $= 5$ is telling you: "values typically land about 5 units away from the mean."

Why spread matters for neural networks: When we initialize weights, we need to control their spread. Too large → activations explode. Too small → gradients vanish. The famous Xavier and He initialization schemes are all about setting the right spread.

3.7 The bell curve (normal distribution)

The **normal distribution** (also called Gaussian, or "bell curve") is the most important continuous distribution. You've seen it: most values cluster near the middle, with fewer values as you move away.

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

Don't memorize this formula. What matters:

- μ (mu) = the mean (center of the bell)
- σ (sigma) = the standard deviation (width of the bell)
- We write $X \sim \mathcal{N}(\mu, \sigma^2)$ to mean "X follows a normal distribution with mean μ and variance σ^2"

Why is the bell curve everywhere?

The **central limit theorem** answers this: if you add up many independent random things, the sum looks like a bell curve—no matter what the individual things look like.

Roll one die: uniform distribution (each number equally likely). Roll 100 dice and sum them: bell curve centered around 350.

This matters for neural networks because each neuron computes a sum:

$$\text{output} = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n$$

Even if individual weights and inputs have weird distributions, their sum tends toward a bell curve. This is why normal distributions appear constantly in neural network analysis.

The standard normal: $\mathcal{N}(0, 1)$ has mean 0 and spread 1. Any normal can be converted to standard normal by subtracting the mean and dividing by the standard deviation:

$$Z = \frac{X - \mu}{\sigma}$$

If $X \sim \mathcal{N}(\mu, \sigma^2)$, then $Z \sim \mathcal{N}(0, 1)$.

3.8 The categorical distribution and softmax

In transformers, the output is a distribution over a vocabulary of V words. This is a **categorical distribution**: V possible outcomes with probabilities $p_1, p_2, \ldots, p_V$ where $\sum_i p_i = 1$.

How do we produce such a distribution from a neural network? The network outputs V real numbers (called **logits**), one for each word. These can be any real numbers—positive, negative, large, small. We need to convert them into valid probabilities.

The **softmax** function does this:

$$\text{softmax}(z_i) = \frac{\exp(z_i)}{\sum_{j=1}^{V} \exp(z_j)}$$

Let's trace through an example. Suppose $V = 4$ and the logits are $\mathbf{z} = [2.0, 1.0, 0.1, -1.0]$.

Step 1: Exponentiate each logit:

$$[\exp(2.0), \exp(1.0), \exp(0.1), \exp(-1.0)] = [7.39, 2.72, 1.11, 0.37]$$

Step 2: Sum them: $7.39 + 2.72 + 1.11 + 0.37 = 11.59$

Step 3: Divide each by the sum:

$$\text{softmax}(\mathbf{z}) = [0.638, 0.235, 0.096, 0.032]$$

Let's verify: $0.638 + 0.235 + 0.096 + 0.032 = 1.001 \approx 1$

Notice what softmax does: - **Larger logits → larger probabilities**: The logit 2.0 became probability 0.638 (largest) - **Preserves ordering**: The ranking of logits equals the ranking of probabilities - **Exponential amplification**: Small differences in logits become larger differences in probabilities

The softmax is *soft* because it gives non-zero probability to every option (unlike a hard max that picks one). This smoothness is crucial for gradient-based learning.

3.9 Uncertainty (entropy)

How do we measure how "uncertain" or "spread out" a probability distribution is?

Visualizing uncertainty

Imagine a language model predicting the next word. Here are three different predictions:

	Model A (certain)	Model B (uncertain)	Model C (leaning)
cat	95%	25%	70%
dog	3%	25%	10%
bird	1%	25%	10%
fish	1%	25%	10%

Model A is confident—it's pretty sure the answer is "cat." Model B has no idea—all options equally likely. Model C leans toward "cat" but isn't sure.

Entropy is a single number that captures this. Low entropy = confident. High entropy = uncertain.

Friendly term	Formal term	Notation
Uncertainty	Entropy	H

Building the formula step by step

The key insight: **rare events are surprising**.

If I say "the sun rose this morning"—not surprising (it always does). If I say "a meteor hit your car"—very surprising (that almost never happens).

We measure surprise as: surprise $= -\log(\text{probability})$

Probability	Surprise	Interpretation
1.0	0	Certain things aren't surprising
0.5	0.7	Coin flip—mildly surprising either way
0.1	2.3	Unlikely things are surprising
0.01	4.6	Rare things are very surprising

Entropy = expected surprise

For each outcome, multiply its probability by its surprise, then add them up:

$$H = \sum_{\text{outcomes}} \text{probability} \times \text{surprise} = -\sum_{x} p(x) \log p(x)$$

Computing entropy for our three models

Model A (95% cat, 3% dog, 1% bird, 1% fish):

The likely outcome (cat) has low surprise. The unlikely outcomes have high surprise but low probability, so they contribute little.

$$H = 0.95 \times 0.05 + 0.03 \times 3.5 + 0.01 \times 4.6 + 0.01 \times 4.6 \approx 0.29$$

Low entropy—the model is confident.

Model B (25% each):

Every outcome is equally surprising, and equally likely to occur.

$$H = 4 \times (0.25 \times 1.39) = 1.39$$

Maximum entropy for 4 outcomes—the model has no idea.

Model C (70% cat, 10% each for the rest):

$$H = 0.70 \times 0.36 + 3 \times (0.10 \times 2.30) = 0.25 + 0.69 = 0.94$$

Medium entropy—somewhat confident.

Summary

Model	Entropy	Confidence
A (certain)	0.29	High
C (leaning)	0.94	Medium
B (no idea)	1.39	Low (maximum for 4 outcomes)

Why this matters: During training, we want our model to become more confident about correct answers. Watching entropy decrease is one way to see the model learning.

3.10 Cross-entropy: the training loss

This is **the most important formula in machine learning**. It's how we measure "how wrong is the model?" and therefore what we minimize during training.

The setup

We're classifying an image. The true answer is "dog", but look at the model's prediction:

Class	True	Model predicts
cat	0%	70%
dog	**100%** ← correct answer	10%
bird	0%	15%
fish	0%	5%

The model is wrong—it thinks it's probably a cat! But *how* wrong? We need a number.

The idea: how surprised is the model?

The true answer is "dog." The model assigned 10% to dog.

If you only had a 10% belief in something and it turned out to be true, you'd be pretty surprised!

$$\text{loss} = -\log(\text{probability model assigned to correct answer})$$

For our example: $\text{loss} = -\log(0.10) = 2.30$

Friendly term	Formal term	Notation
Loss (how wrong)	Cross-entropy	$H(p, q)$

How loss changes with confidence

Model's confidence in correct answer	Loss	Note
1%	4.61	Very wrong—huge loss
10%	2.30	
25%	1.39	
50%	0.69	
70%	0.36	
90%	0.11	
99%	0.01	Almost certain—tiny loss

Notice two things: 1. **Wrong predictions are heavily penalized**: Going from 1% to 10% reduces loss by 2.3 2. **Diminishing returns for high confidence**: Going from 90% to 99% only reduces loss by 0.1

This is exactly what we want! The model should focus on getting wrong answers less wrong, rather than making already-good predictions slightly better.

A training story

Watch what happens as the model learns (true answer is "dog"):

Epoch	P(dog)	Loss
1	20%	1.61
2	40%	0.92
3	60%	0.51
4	80%	0.22
5	90%	0.11

Each epoch, the model assigns more probability to "dog", and the loss decreases.

The formula

When the true answer is simply "it's class j" (one-hot), the formula is just:

$$\text{loss} = -\log(q_j)$$

where q_j is the probability the model assigned to the correct class.

The general formula (when multiple answers could be correct with different weights):

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

Why cross-entropy is perfect for training

The gradient tells us how to update each probability (true answer is dog):

Class	Predicted	Gradient	Action
cat	70%	+0.70	push DOWN
dog	10%	−0.90	push UP (correct answer)
bird	15%	+0.15	push DOWN
fish	5%	+0.05	push DOWN

The gradient is simply: predicted − true

- For dog (correct): $0.10 - 1.00 = -0.90$ → negative → increase this probability
- For wrong answers: predicted $- 0$ → positive → decrease these probabilities

Training automatically moves probability mass from wrong answers to right answers.

3.11 KL divergence: the "real" distance between distributions

Cross-entropy measures how wrong our model is—but it includes some "baseline" wrongness that's unavoidable. **KL divergence** strips away that baseline to measure the *pure* difference between two distributions.

The problem with cross-entropy

Recall: cross-entropy $= -\sum p(x) \log q(x)$, where p is truth and q is our model.

But even a *perfect* model has non-zero cross-entropy! If the true distribution is uncertain (say, 50/50), even predicting exactly 50/50 gives:

$$H(p, p) = -[0.5 \log(0.5) + 0.5 \log(0.5)] = 0.69$$

This 0.69 isn't "wrongness"—it's the **inherent uncertainty** in the true distribution. You can't do better than this.

Entropy of the true distribution

The entropy $H(p)$ measures how uncertain the true distribution is:

True distribution	Entropy $H(p)$	Interpretation
[1.0, 0, 0, 0]	0	Certain—one right answer
[0.7, 0.1, 0.1, 0.1]	0.94	Mostly certain
[0.25, 0.25, 0.25, 0.25]	1.39	Maximum uncertainty

This is the *minimum possible* cross-entropy. No model can beat this—it's the irreducible uncertainty in the problem itself.

KL divergence = cross-entropy − entropy

$$D_{KL}(p \| q) = H(p, q) - H(p) = \underbrace{-\sum p(x) \log q(x)}_{\text{cross-entropy}} - \underbrace{(-\sum p(x) \log p(x))}_{\text{entropy}}$$

Simplifying:

$$D_{KL}(p \| q) = \sum p(x) \log \frac{p(x)}{q(x)}$$

Concrete example

True distribution: $p = [0.7, 0.3]$ (say, 70% chance of "cat", 30% chance of "dog")

Model prediction: $q = [0.5, 0.5]$ (model thinks it's 50/50)

Step 1: Compute entropy of truth

$$H(p) = -[0.7\log(0.7) + 0.3\log(0.3)] = -[0.7 \times (-0.36) + 0.3 \times (-1.20)]$$

$$= 0.25 + 0.36 = 0.61$$

Step 2: Compute cross-entropy

$$H(p, q) = -[0.7\log(0.5) + 0.3\log(0.5)] = -[0.7 \times (-0.69) + 0.3 \times (-0.69)]$$

$$= 0.48 + 0.21 = 0.69$$

Step 3: Compute KL divergence

$$D_{KL}(p\|q) = H(p, q) - H(p) = 0.69 - 0.61 = 0.08$$

Interpretation: - Cross-entropy (0.69) = total "surprise" when using model q - Entropy (0.61) = unavoidable surprise due to true uncertainty - KL divergence (0.08) = *extra* surprise caused by model being wrong

Why we minimize cross-entropy, not KL divergence

Since $H(p)$ is constant (the true distribution doesn't change during training), minimizing cross-entropy and minimizing KL divergence are equivalent:

$$\arg\min_q H(p, q) = \arg\min_q D_{KL}(p\|q)$$

We use cross-entropy in practice because it's simpler to compute—we don't need to know $H(p)$.

Key properties of KL divergence

Property	Meaning
$D_{KL} \geq 0$	Always non-negative
$D_{KL} = 0$ iff $p = q$	Zero only when distributions match exactly
Not symmetric	$D_{KL}(p\|q) \neq D_{KL}(q\|p)$ in general

3.12 Conditional probability: updating beliefs with new information

Conditional probability answers: "What do I believe now that I have new information?"

Example: the die behind the screen

You roll a die but can't see it. What's the probability it's even?

- All possibilities: 1, 2, 3, 4, 5, 6
- Even numbers: 2, 4, 6 → **3 out of 6 = 50%**

Now your friend peeks and says: "It's at least 4."

- Still possible: 4, 5, 6 (only 3 options remain)
- Even AND at least 4: 4, 6 → **2 out of 3 = 67%**

The new information changed your belief from 50% to 67%.

The notation

$P(A|B)$ means "probability of A, given that B is true"

Read the vertical bar as "given that" or "assuming"

- $P(\text{even}) = 50\%$ (no extra information)
- $P(\text{even}|\text{at least 4}) = 67\%$ (with the hint)

The formula

$$P(A|B) = \frac{P(\text{both A and B})}{P(B)}$$

For our example: - $P(\text{even AND at least 4}) = P(\{4,6\}) = 2/6$ - $P(\text{at least 4}) = P(\{4,5,6\}) = 3/6$ - $P(\text{even}|\text{at least 4}) = \frac{2/6}{3/6} = \frac{2}{3}$

Why this matters for language models

A language model is ALL about conditional probability. Given some words, what's the probability of the next word?

Context: "The cat sat on the"

Next word	Probability
mat	35%
floor	20%
rug	15%
table	10%
elephant	0.01%
...	...

Generating text = chaining conditionals

To compute the probability of a whole sentence, multiply the conditionals:

$$P(\text{"The cat sat"}) = P(\text{"The"}) \times P(\text{"cat"}|\text{"The"}) \times P(\text{"sat"}|\text{"The cat"})$$

$$= 0.05 \times 0.02 \times 0.15 = 0.00015$$

This is called the **chain rule of probability**. Transformers generate text exactly this way: predict one token, add it to the context, predict the next token, repeat.

3.13 Independence: when information doesn't help

Two events are **independent** if knowing one tells you nothing about the other.

Example: independent events (coin flips)

Flip two coins. All four outcomes are equally likely:

	Coin 2 = H	Coin 2 = T
Coin 1 = H	HH (25%)	HT (25%)
Coin 1 = T	TH (25%)	TT (25%)

If I tell you Coin 1 was heads, what's the probability Coin 2 is heads?

Still 50%. Coin 1 doesn't affect Coin 2. They're **independent**.

$P(\text{Coin 2 = H}|\text{Coin 1 = H}) = P(\text{Coin 2 = H}) = 50\%$

Example: dependent events (cards)

A standard deck has 52 cards: - **Red cards (26)**: 13 hearts + 13 diamonds - **Black cards (26)**: 13 spades + 13 clubs

If I tell you the card is a heart, what's the probability it's red?

100%. Hearts are always red. These events are **NOT independent**.

$P(\text{red}) = 50\%$, but $P(\text{red}|\text{heart}) = 100\%$

The test for independence

Events A and B are independent if and only if:

$$P(\text{both}) = P(A) \times P(B)$$

Why this formula?

Start from the definition: A and B are independent if knowing B doesn't change your belief about A:

$$P(A|B) = P(A)$$

Now recall the formula for conditional probability:

$$P(A|B) = \frac{P(\text{both A and B})}{P(B)}$$

If A and B are independent, we can substitute $P(A|B) = P(A)$:

$$P(A) = \frac{P(\text{both})}{P(B)}$$

Multiply both sides by $P(B)$:

$$P(A) \times P(B) = P(\text{both})$$

That's the formula! It's just a rearrangement of "knowing B doesn't change my belief about A."

Intuition with coins

Why is $P(\text{both heads}) = 0.5 \times 0.5$?

Think of it as: "To get both heads, I need the first coin to land heads (50% chance), AND THEN I need the second coin to land heads (50% chance)."

Since Coin 2 doesn't care what Coin 1 did, I just multiply the probabilities.

Intuition with cards

Why is $P(\text{red AND heart}) \neq P(\text{red}) \times P(\text{heart})$?

- $P(\text{red}) = 0.5$ (26 red cards out of 52)
- $P(\text{heart}) = 0.25$ (13 hearts out of 52)
- If independent: $P(\text{both}) = 0.5 \times 0.25 = 0.125$

But actually $P(\text{red AND heart}) = P(\text{heart}) = 0.25$, because every heart IS red.

The events aren't independent—knowing "heart" completely determines "red."

Why this matters for neural networks

When we initialize weights independently, we can predict how variance adds up:

$$\text{output} = w_1x_1 + w_2x_2 + w_3x_3 + \cdots$$

If weights are independent:

$$\text{Var}(\text{output}) = \text{Var}(w_1x_1) + \text{Var}(w_2x_2) + \text{Var}(w_3x_3) + \cdots$$

This formula is the foundation of proper weight initialization (Xavier, He, etc.). Without independence, we couldn't predict how signals scale through the network.

Chapter 4

Notation conventions

> **Learning objectives**
>
> After completing this chapter, you will be able to:
> - Read and write vectors, matrices, and scalars using standard notation
> - Interpret subscript and superscript conventions for indexing
> - Recognize common mathematical symbols used throughout the book
> - Navigate between mathematical notation and code implementations

Throughout this book, we use the following conventions:

- **Scalars**: Lowercase italic letters (x, y, n, α, β)
- **Vectors**: Lowercase bold letters ($\mathbf{x}, \mathbf{v}, \mathbf{w}$), assumed to be column vectors unless transposed
- **Matrices**: Uppercase bold letters ($\mathbf{A}, \mathbf{W}, \mathbf{X}$)
- **Sets**: Uppercase calligraphic letters ($\mathcal{D}, \mathcal{V}, \mathcal{X}$)
- **Random variables**: Uppercase italic letters (X, Y, Z)

We denote element i of vector $\mathbf{v}$ as v_i or $[\mathbf{v}]_i$. Element (i, j) of matrix $\mathbf{A}$ is a_{ij} or $[\mathbf{A}]_{ij}$. We write $\mathbf{a}_i$ for the i-th column of $\mathbf{A}$ and $\mathbf{a}_{i:}$ for the i-th row (as a row vector).

4.1 Common sets

- $\mathbb{R}$: real numbers
- $\mathbb{R}^n$: n-dimensional real vectors
- $\mathbb{R}^{m \times n}$: $m \times n$ real matrices
- $\mathbb{N}$: natural numbers $\{0, 1, 2, ...\}$
- $\mathbb{Z}$: integers

4.2 Functions and operators

- $\|\mathbf{v}\|$: Euclidean norm (default is L^2)
- $\|\mathbf{v}\|_p$: L^p norm
- $\langle \mathbf{u}, \mathbf{v} \rangle$ or $\mathbf{u} \cdot \mathbf{v}$ or $\mathbf{u}^T\mathbf{v}$: dot product
- ∇f: gradient
- $\mathbb{E}[X]$: expectation
- $\text{Var}(X)$: variance
- $\text{softmax}(\mathbf{z})$: softmax function
- $\log x$: natural logarithm (base e)

- $\log_2 x$: logarithm base 2

4.3 Asymptotic notation

- $f(n) = O(g(n))$: f grows no faster than g
- $f(n) = \Omega(g(n))$: f grows at least as fast as g
- $f(n) = \Theta(g(n))$: f and g grow at the same rate

4.4 Code conventions

In code examples, we use Python with NumPy and PyTorch. Matrix dimensions are often shown as comments, e.g., (batch_size, seq_len, d_model).

This completes our review of prerequisites. We've covered the essential linear algebra, calculus, and probability needed to understand transformers deeply. In the next chapter, we'll apply these tools to build the foundations of neural networks.

Chapter 5

Neural networks basics

Learning objectives

After completing this chapter, you will be able to:

- Describe how a single neuron computes a weighted sum with nonlinear activation
- Construct multilayer networks and trace information flow through them
- Define common loss functions for regression and classification
- Derive the backpropagation algorithm for computing gradients
- Implement gradient descent to update network parameters

Neural networks are the foundation of modern machine learning. Before we can understand transformers, we need to master how neural networks work: how they represent functions, how information flows forward through them, and how they learn through backpropagation. This chapter develops these ideas from first principles, with the mathematical rigor we established in the prerequisites.

5.1 The single neuron

The simplest neural network is a single neuron. It takes n inputs $x_1, x_2, \ldots, x_n$, multiplies each by a weight, adds them up with a bias term, and applies a nonlinear function:

$$y = \sigma\left(\sum_{i=1}^{n} w_i x_i + b\right) = \sigma(\mathbf{w}^T\mathbf{x} + b)$$

where $\mathbf{w} = [w_1, \ldots, w_n]^T$ are the weights, b is the bias, and σ is an **activation function**. Let's unpack each component.

The linear part $\mathbf{w}^T\mathbf{x} + b$ computes a weighted sum of inputs plus a constant offset. Geometrically, this defines a hyperplane in input space. In 2D, if $\mathbf{x} = [x_1, x_2]^T$, then $w_1 x_1 + w_2 x_2 + b = 0$ is a line. The weight vector $\mathbf{w}$ is perpendicular to this line, and b shifts it away from the origin.

Consider a concrete example with $\mathbf{w} = [2, 1]^T$ and $b = -3$. The linear part computes $2x_1 + x_2 - 3$. When is this positive? When $x_2 > -2x_1 + 3$, i.e., above the line $x_2 = -2x_1 + 3$. The neuron divides input space into two regions: one where the linear part is positive, one where it's negative.

But if we only had the linear part, the neuron could only represent linear functions. Stacking linear functions gives more linear functions: if $f(\mathbf{x}) = \mathbf{A}\mathbf{x}$ and $g(\mathbf{y}) = \mathbf{B}\mathbf{y}$, then $g(f(\mathbf{x})) = \mathbf{B}\mathbf{A}\mathbf{x}$, which is still linear. This is why we need the activation function σ: it introduces nonlinearity.

Common activation functions include:

Sigmoid: $\sigma(z) = \frac{1}{1+e^{-z}}$. This squashes any input to the range $(0, 1)$. For large positive z, $\sigma(z) \approx 1$. For large negative z, $\sigma(z) \approx 0$. The transition is smooth, centered at $z = 0$ where $\sigma(0) = 0.5$.

ReLU (Rectified Linear Unit): $\text{ReLU}(z) = \max(0, z)$. This is zero for negative inputs and the identity for positive inputs. It's piecewise linear but not linear overall (the "kink" at zero breaks linearity). ReLU is computationally cheap and avoids some training problems that plague sigmoid.

Tanh: $\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$. Like sigmoid but outputs in $(-1, 1)$ and is centered at zero.

What would violate our need for nonlinearity? The identity function $\sigma(z) = z$ is linear, so using it as an activation would make the entire network linear regardless of depth. Any affine function $\sigma(z) = az + b$ has the same problem.

Let's trace through a concrete neuron. Suppose we have inputs $\mathbf{x} = [0.5, 0.8]^T$, weights $\mathbf{w} = [0.4, 0.6]^T$, bias $b = -0.5$, and we use sigmoid activation. The computation proceeds:

$$z = \mathbf{w}^T\mathbf{x} + b = 0.4 \cdot 0.5 + 0.6 \cdot 0.8 - 0.5 = 0.2 + 0.48 - 0.5 = 0.18$$

$$y = \sigma(0.18) = \frac{1}{1 + e^{-0.18}} = \frac{1}{1 + 0.835} = \frac{1}{1.835} \approx 0.545$$

The neuron outputs approximately 0.545, slightly above the midpoint of 0.5 because the weighted sum 0.18 is slightly positive.

5.2 Multilayer networks

A single neuron can only represent functions that are "nearly linear" (linear followed by a squashing function). To represent complex functions, we stack neurons into layers.

A **feedforward neural network** (also called multilayer perceptron or MLP) consists of:

- An **input layer**: the raw input $\mathbf{x} \in \mathbb{R}^{n_0}$
- One or more **hidden layers**: intermediate representations
- An **output layer**: the final prediction $\mathbf{y} \in \mathbb{R}^{n_L}$

Let's define the computation precisely. For a network with L layers, let $\mathbf{h}^{(0)} = \mathbf{x}$ be the input. For each layer $\ell = 1, \ldots, L$:

$$\mathbf{z}^{(\ell)} = \mathbf{W}^{(\ell)}\mathbf{h}^{(\ell-1)} + \mathbf{b}^{(\ell)}$$

$$\mathbf{h}^{(\ell)} = \sigma(\mathbf{z}^{(\ell)})$$

where $\mathbf{W}^{(\ell)} \in \mathbb{R}^{n_\ell \times n_{\ell-1}}$ is the weight matrix for layer ℓ, $\mathbf{b}^{(\ell)} \in \mathbb{R}^{n_\ell}$ is the bias vector, and σ is applied element-wise. The final output is $\mathbf{y} = \mathbf{h}^{(L)}$.

Let's work through a concrete example. Consider a network with:

- Input dimension: $n_0 = 2$
- Hidden layer: $n_1 = 3$ neurons with ReLU activation
- Output layer: $n_2 = 1$ neuron with sigmoid activation

The weight matrices have shapes $\mathbf{W}^{(1)} \in \mathbb{R}^{3\times 2}$ and $\mathbf{W}^{(2)} \in \mathbb{R}^{1\times 3}$. Suppose:

$$\mathbf{W}^{(1)} = \begin{bmatrix} 0.2 & 0.4 \\ -0.5 & 0.3 \\ 0.1 & -0.2 \end{bmatrix}, \quad \mathbf{b}^{(1)} = \begin{bmatrix} 0.1 \\ -0.1 \\ 0.2 \end{bmatrix}$$

$$\mathbf{W}^{(2)} = \begin{bmatrix} 0.6 & -0.4 & 0.5 \end{bmatrix}, \quad \mathbf{b}^{(2)} = \begin{bmatrix} -0.2 \end{bmatrix}$$

For input $\mathbf{x} = [1.0, 0.5]^T$:

Layer 1 (hidden):

$$\mathbf{z}^{(1)} = \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)} = \begin{bmatrix} 0.2 \cdot 1.0 + 0.4 \cdot 0.5 \\ -0.5 \cdot 1.0 + 0.3 \cdot 0.5 \\ 0.1 \cdot 1.0 - 0.2 \cdot 0.5 \end{bmatrix} + \begin{bmatrix} 0.1 \\ -0.1 \\ 0.2 \end{bmatrix} = \begin{bmatrix} 0.4 \\ -0.35 \\ 0 \end{bmatrix} + \begin{bmatrix} 0.1 \\ -0.1 \\ 0.2 \end{bmatrix} = \begin{bmatrix} 0.5 \\ -0.45 \\ 0.2 \end{bmatrix}$$

$$\mathbf{h}^{(1)} = \text{ReLU}(\mathbf{z}^{(1)}) = \begin{bmatrix} \max(0, 0.5) \\ \max(0, -0.45) \\ \max(0, 0.2) \end{bmatrix} = \begin{bmatrix} 0.5 \\ 0 \\ 0.2 \end{bmatrix}$$

Notice how the second neuron outputs 0 because its pre-activation was negative. ReLU "kills" that neuron for this input.

Layer 2 (output):

$$z^{(2)} = \mathbf{W}^{(2)}\mathbf{h}^{(1)} + b^{(2)} = 0.6 \cdot 0.5 + (-0.4) \cdot 0 + 0.5 \cdot 0.2 - 0.2 = 0.3 + 0 + 0.1 - 0.2 = 0.2$$

$$y = \sigma(0.2) = \frac{1}{1 + e^{-0.2}} \approx 0.55$$

The network maps input $[1.0, 0.5]^T$ to output ≈ 0.55.

5.2.1 Why depth matters

Why use multiple layers instead of one wide layer? The key insight is that depth enables hierarchical representations.

A single hidden layer with enough neurons can approximate any continuous function (this is the **universal approximation theorem**). But "enough neurons" might be exponentially many. Deep networks can represent certain functions much more efficiently.

Consider computing the parity function: output 1 if an odd number of inputs are 1, output 0 otherwise. With one hidden layer, you need exponentially many neurons (roughly 2^{n-1}). With $\log n$ layers, you need only $O(n)$ neurons total, by computing parity hierarchically: first compute parity of pairs, then parity of those results, and so on.

More practically, deep networks learn hierarchical features. In image recognition, early layers learn edges, middle layers learn shapes, deep layers learn objects. In language, early layers learn character patterns, middle layers learn words and phrases, deep layers learn semantics. This hierarchical structure mirrors the structure of the data.

5.3 Loss functions

A neural network has parameters θ (all weights and biases). Given training data $\{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^{m}$, we want to find parameters that make the network's predictions close to the true outputs. We measure "closeness" with a **loss function** $\mathcal{L}(\theta)$.

5.3.1 Mean squared error

For regression (predicting continuous values), a natural choice is **mean squared error** (MSE):

$$\mathcal{L}_{\text{MSE}} = \frac{1}{m}\sum_{i=1}^{m}(f(\mathbf{x}^{(i)};\theta) - y^{(i)})^2$$

where $f(\mathbf{x};\theta)$ is the network's output. This penalizes predictions quadratically: being off by 2 is four times worse than being off by 1.

Let's compute MSE for a simple example. Suppose we have three data points with true values $y^{(1)} = 1, y^{(2)} = 0, y^{(3)} = 1$ and our network predicts $\hat{y}^{(1)} = 0.8, \hat{y}^{(2)} = 0.3, \hat{y}^{(3)} = 0.9$. Then:

$$\mathcal{L}_{\text{MSE}} = \frac{1}{3}[(0.8-1)^2 + (0.3-0)^2 + (0.9-1)^2] = \frac{1}{3}[0.04 + 0.09 + 0.01] = \frac{0.14}{3} \approx 0.047$$

5.3.2 Cross-entropy loss

For classification, we use **cross-entropy loss**. Suppose we're doing binary classification: the true label is $y \in \{0, 1\}$ and the network outputs a probability $\hat{y} = \sigma(z) \in (0, 1)$. The cross-entropy loss is:

$$\mathcal{L}_{\text{CE}} = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)} \log \hat{y}^{(i)} + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)})\right]$$

Why this formula? Recall from the prerequisites that cross-entropy measures how well a predicted distribution q matches a true distribution p. Here, the true distribution puts all mass on the correct class. If $y = 1$, the loss is $-\log \hat{y}$: high loss if $\hat{y}$ is small (confident wrong prediction), low loss if $\hat{y}$ is large (confident correct prediction). If $y = 0$, the loss is $-\log(1 - \hat{y})$: high loss if $\hat{y}$ is large, low loss if $\hat{y}$ is small.

Concrete example: true labels $y^{(1)} = 1, y^{(2)} = 0, y^{(3)} = 1$ and predictions $\hat{y}^{(1)} = 0.9, \hat{y}^{(2)} = 0.2, \hat{y}^{(3)} = 0.8$.

$$\mathcal{L}_{\text{CE}} = -\frac{1}{3}\left[\log(0.9) + \log(0.8) + \log(0.8)\right] = -\frac{1}{3}[-0.105 - 0.223 - 0.223] = \frac{0.551}{3} \approx 0.184$$

For multiclass classification with K classes, the network outputs a probability distribution $\hat{\mathbf{y}} = \text{softmax}(\mathbf{z})$ and the cross-entropy becomes:

$$\mathcal{L}_{\text{CE}} = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log \hat{y}_k^{(i)}$$

where $y_k^{(i)} = 1$ if example i belongs to class k, and 0 otherwise (one-hot encoding).

5.4 Backpropagation

We want to minimize the loss $\mathcal{L}(\theta)$ by adjusting parameters θ. Gradient-based optimization requires computing $\nabla_\theta \mathcal{L}$: how does the loss change when we change each parameter? For a network with millions of parameters, we need an efficient algorithm. This is **backpropagation**.

5.4.1 The computational graph perspective

A neural network computation can be viewed as a directed acyclic graph where:

- Nodes represent intermediate values (inputs, activations, outputs, loss)
- Edges represent operations (matrix multiply, add bias, apply activation)

For our two-layer network example:

$$\mathbf{x} \to \mathbf{z}^{(1)} \to \mathbf{h}^{(1)} \to z^{(2)} \to y \to \mathcal{L}$$

Each arrow is a function. To find $\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(1)}}$, we need to trace how changes in $\mathbf{W}^{(1)}$ propagate through the graph to affect $\mathcal{L}$.

5.4.2 Forward and backward passes

Backpropagation has two phases:

1. **Forward pass**: Compute all intermediate values from input to loss
2. **Backward pass**: Compute all gradients from loss back to parameters

The backward pass uses the chain rule systematically. For each node, we compute "how much does the loss change if this node's value changes?" Let's define:

$$\delta^{(\ell)} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(\ell)}}$$

This is the "error signal" at layer ℓ: how sensitive is the loss to the pre-activation values?

5.4.3 Deriving the backpropagation equations

Let's derive backpropagation for our two-layer network with sigmoid output and MSE loss. For a single training example (we'll drop the superscript (i)), the loss is:

$$\mathcal{L} = \frac{1}{2}(y - \hat{y})^2$$

where $\hat{y} = \sigma(z^{(2)})$ and we include the $\frac{1}{2}$ to simplify derivatives.

Output layer gradient:

$$\frac{\partial \mathcal{L}}{\partial z^{(2)}} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z^{(2)}} = -(y - \hat{y}) \cdot \sigma(z^{(2)})(1 - \sigma(z^{(2)})) = -(\hat{y} - y) \cdot \hat{y}(1 - \hat{y})$$

Wait, let me redo this more carefully. We have:

$$\frac{\partial \mathcal{L}}{\partial \hat{y}} = \frac{\partial}{\partial \hat{y}} \left[\frac{1}{2}(y - \hat{y})^2 \right] = -(y - \hat{y}) = \hat{y} - y$$

$$\frac{\partial \hat{y}}{\partial z^{(2)}} = \left. \frac{d\sigma}{dz} \right|_{z=z^{(2)}} = \sigma(z^{(2)})(1 - \sigma(z^{(2)})) = \hat{y}(1 - \hat{y})$$

So:

$$\delta^{(2)} = \frac{\partial \mathcal{L}}{\partial z^{(2)}} = (\hat{y} - y) \cdot \hat{y}(1 - \hat{y})$$

Now we can compute the gradients for layer 2's parameters:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(2)}} = \frac{\partial \mathcal{L}}{\partial z^{(2)}} \cdot \frac{\partial z^{(2)}}{\partial \mathbf{W}^{(2)}} = \delta^{(2)} \cdot (\mathbf{h}^{(1)})^T$$

$$\frac{\partial \mathcal{L}}{\partial b^{(2)}} = \delta^{(2)}$$

Hidden layer gradient:

To backpropagate to layer 1, we need:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{h}^{(1)}} = \frac{\partial \mathcal{L}}{\partial z^{(2)}} \cdot \frac{\partial z^{(2)}}{\partial \mathbf{h}^{(1)}} = \delta^{(2)} \cdot \mathbf{W}^{(2)}$$

This is a key insight: the error signal at a layer equals the error signal from the next layer, multiplied by the weights connecting them. The weights "distribute" the error backwards.

Then:

$$\delta^{(1)} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(1)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}^{(1)}} \odot \frac{\partial \mathbf{h}^{(1)}}{\partial \mathbf{z}^{(1)}}$$

where $\odot$ is element-wise multiplication. For ReLU:

$$\frac{\partial h_j^{(1)}}{\partial z_j^{(1)}} = \begin{cases} 1 & \text{if } z_j^{(1)} > 0 \\ 0 & \text{if } z_j^{(1)} \leq 0 \end{cases}$$

So $\delta^{(1)} = (\mathbf{W}^{(2)})^T \delta^{(2)} \odot \mathbf{1}_{z^{(1)}>0}$, where $\mathbf{1}_{z^{(1)}>0}$ is 1 where $z^{(1)}$ is positive and 0 otherwise.

Finally:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(1)}} = \delta^{(1)} \cdot \mathbf{x}^T$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(1)}} = \delta^{(1)}$$

5.4.4 Concrete backpropagation example

Let's compute gradients for our earlier forward pass example. We had:

- Input: $\mathbf{x} = [1.0, 0.5]^T$
- Hidden activations: $\mathbf{z}^{(1)} = [0.5, -0.45, 0.2]^T$, $\mathbf{h}^{(1)} = [0.5, 0, 0.2]^T$
- Output: $z^{(2)} = 0.2$, $\hat{y} = \sigma(0.2) \approx 0.55$

Suppose the true label is $y = 1$ (so we want the output to be higher).

Step 1: Output layer error

$$\delta^{(2)} = (\hat{y} - y) \cdot \hat{y}(1 - \hat{y}) = (0.55 - 1) \cdot 0.55 \cdot 0.45 = -0.45 \cdot 0.2475 \approx -0.111$$

The negative sign indicates we should adjust to increase the output.

Step 2: Output layer gradients

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(2)}} = \delta^{(2)} \cdot (\mathbf{h}^{(1)})^T = -0.111 \cdot [0.5, 0, 0.2] = [-0.056, 0, -0.022]$$

$$\frac{\partial \mathcal{L}}{\partial b^{(2)}} = -0.111$$

Step 3: Backpropagate to hidden layer

$$\frac{\partial \mathcal{L}}{\partial \mathbf{h}^{(1)}} = (\mathbf{W}^{(2)})^T \cdot \delta^{(2)} = \begin{bmatrix} 0.6 \\ -0.4 \\ 0.5 \end{bmatrix} \cdot (-0.111) = \begin{bmatrix} -0.067 \\ 0.044 \\ -0.056 \end{bmatrix}$$

Step 4: ReLU gradient

The ReLU gradient is 1 where $z^{(1)} > 0$ and 0 otherwise. Since $\mathbf{z}^{(1)} = [0.5, -0.45, 0.2]^T$, the mask is $[1, 0, 1]^T$.

$$\delta^{(1)} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}^{(1)}} \odot [1, 0, 1]^T = [-0.067, 0, -0.056]^T$$

The gradient through the second neuron is zero because ReLU killed it during the forward pass.

Step 5: Hidden layer gradients

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(1)}} = \delta^{(1)} \cdot \mathbf{x}^T = \begin{bmatrix} -0.067 \\ 0 \\ -0.056 \end{bmatrix} \cdot [1.0, 0.5] = \begin{bmatrix} -0.067 & -0.033 \\ 0 & 0 \\ -0.056 & -0.028 \end{bmatrix}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(1)}} = [-0.067, 0, -0.056]^T$$

These gradients tell us how to adjust each parameter to reduce the loss.

5.5 Gradient descent

With gradients in hand, we update parameters to reduce the loss. The simplest approach is **gradient descent**:

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

where $\eta > 0$ is the **learning rate**. We move parameters in the direction opposite to the gradient (since the gradient points toward increasing loss).

5.5.1 The learning rate

The learning rate η controls step size. Too large, and we might overshoot the minimum and diverge. Too small, and training takes forever.

Consider minimizing $f(\theta) = \theta^2$. The minimum is at $\theta = 0$ with gradient $\frac{df}{d\theta} = 2\theta$. Starting at $\theta_0 = 1$:

- With $\eta = 0.1$: $\theta_1 = 1 - 0.1 \cdot 2 = 0.8$, $\theta_2 = 0.8 - 0.1 \cdot 1.6 = 0.64$, … converges slowly
- With $\eta = 0.5$: $\theta_1 = 1 - 0.5 \cdot 2 = 0$, reaches minimum in one step!
- With $\eta = 0.9$: $\theta_1 = 1 - 0.9 \cdot 2 = -0.8$, $\theta_2 = -0.8 - 0.9 \cdot (-1.6) = 0.64$, oscillates but converges
- With $\eta = 1.1$: $\theta_1 = 1 - 1.1 \cdot 2 = -1.2$, $\theta_2 = -1.2 - 1.1 \cdot (-2.4) = 1.44$, diverges!

For this simple quadratic, any $\eta < 1$ converges, $\eta = 0.5$ is optimal, and $\eta > 1$ diverges. Real loss landscapes are more complex, but the intuition holds: there's a "Goldilocks zone" for learning rates.

5.5.2 Stochastic and mini-batch gradient descent

Computing the gradient over all m training examples is expensive. **Stochastic gradient descent** (SGD) uses a single random example:

$$\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L}^{(i)}$$

where $\mathcal{L}^{(i)}$ is the loss for example i. This is noisy but much faster per update.

Mini-batch gradient descent is a compromise: use a small batch of B examples:

$$\theta \leftarrow \theta - \eta \cdot \frac{1}{B} \sum_{i \in \text{batch}} \nabla_\theta \mathcal{L}^{(i)}$$

Typical batch sizes are 32, 64, 128, or 256. Mini-batches reduce variance compared to SGD while remaining computationally efficient.

5.5.3 Momentum and Adam

Plain SGD can be slow, especially in "ravines" where the gradient points mostly sideways. **Momentum** accumulates a velocity:

$$\mathbf{v} \leftarrow \beta \mathbf{v} - \eta \nabla_\theta \mathcal{L}$$

$$\theta \leftarrow \theta + \mathbf{v}$$

where $\beta \approx 0.9$ is the momentum coefficient. This smooths out oscillations and accelerates along consistent gradient directions.

Adam (Adaptive Moment Estimation) goes further by adapting the learning rate for each parameter based on historical gradients. It maintains running averages of the gradient and squared gradient:

$$\mathbf{m} \leftarrow \beta_1 \mathbf{m} + (1 - \beta_1) \nabla_\theta \mathcal{L}$$

$$\mathbf{v} \leftarrow \beta_2 \mathbf{v} + (1 - \beta_2) (\nabla_\theta \mathcal{L})^2$$

$$\theta \leftarrow \theta - \eta \frac{\hat{\mathbf{m}}}{\sqrt{\hat{\mathbf{v}}} + \epsilon}$$

where $\hat{\mathbf{m}}$ and $\hat{\mathbf{v}}$ are bias-corrected estimates and $\epsilon \approx 10^{-8}$ prevents division by zero. Adam is the default optimizer for training transformers.

5.6 Putting it together

Let's summarize the training loop for a neural network:

1. **Initialize** parameters randomly (careful initialization matters; we'll discuss this later)
2. **Repeat** until convergence:
   a. Sample a mini-batch of training examples
   b. **Forward pass**: Compute predictions and loss
   c. **Backward pass**: Compute gradients via backpropagation

   d. **Update**: Adjust parameters using optimizer (SGD, Adam, etc.)
3. **Evaluate** on held-out data to check generalization

This loop is the heartbeat of deep learning. Every neural network, from simple MLPs to massive transformers, trains this way. The differences lie in architecture (what functions the network computes) and scale (how many parameters, how much data, how much compute).

In the next chapter, we'll see why standard feedforward networks struggle with sequential data, motivating the architectures that eventually led to transformers.

Chapter 6

Sequence modeling

Learning objectives

After completing this chapter, you will be able to:

- Explain why feedforward networks struggle with sequential data
- Describe how recurrent neural networks process sequences step by step
- Identify the vanishing and exploding gradient problems in RNNs
- Understand how LSTM and GRU architectures address gradient flow issues
- Recognize the limitations that motivated the transformer architecture

Language, music, stock prices, weather patterns: the world is full of sequential data where order matters. The sentence "the cat sat on the mat" means something different from "the mat sat on the cat." Standard feedforward networks treat inputs as fixed-size vectors with no notion of order. This chapter explores how to model sequences, the challenges that arise, and why these challenges ultimately motivated the transformer architecture.

6.1 The problem of sequential data

A **sequence** is an ordered list of elements: $\mathbf{x} = (x_1, x_2, \ldots, x_T)$ where T is the sequence length. In natural language processing, each x_t might be a word (or subword token). In time series, each x_t might be a measurement at time t.

Sequential data has several properties that make it challenging:

Variable length. Sentences can have 5 words or 50 words. A model should handle both without being redesigned.

Order dependence. The meaning depends on the order. "Dog bites man" and "man bites dog" contain the same words but mean different things.

Long-range dependencies. Elements far apart can be related. In "The cat, which had been sleeping peacefully on the warm windowsill all afternoon, suddenly woke up," the verb "woke" must agree with "cat," not the nearby "windowsill" or "afternoon."

Context sensitivity. The same word means different things in different contexts. "Bank" means something different in "river bank" versus "bank account."

Let's try to apply a feedforward network to sequences and see what goes wrong.

6.2 Why feedforward networks fail

Suppose we want to classify sentences as positive or negative sentiment. A feedforward network needs a fixed-size input vector. How do we represent a sentence?

Approach 1: Fixed-length input. Pad short sentences with zeros and truncate long ones to some maximum length $T_{\max}$. Represent each word as a one-hot vector of vocabulary size V. The input becomes a $T_{\max} \times V$ matrix, which we flatten to a vector of dimension $T_{\max} \cdot V$.

Problems: For vocabulary $V = 10{,}000$ and maximum length $T_{\max} = 100$, the input has dimension 1 million. Most entries are zero (sparse). The network must learn separate weights for "good" in position 1, "good" in position 2, etc. It can't generalize that "good" means the same thing regardless of position.

Approach 2: Bag of words. Sum or average the word vectors, ignoring order. The input is a V-dimensional vector counting word occurrences.

Problems: "The food was good, not bad" and "The food was bad, not good" have identical bag-of-words representations but opposite meanings. Order information is completely lost.

Approach 3: n-grams. Include pairs or triples of consecutive words as features.

Problems: The number of possible n-grams explodes combinatorially. Bigrams give V^2 features, trigrams give V^3. Most are never seen in training. Long-range dependencies spanning more than n words are still missed.

The fundamental issue is that feedforward networks process inputs as unstructured vectors. They have no built-in notion of sequence, position, or the idea that the same pattern (like the word "good") might appear at different positions.

6.3 Recurrent neural networks

Recurrent neural networks (RNNs) address this by processing sequences one element at a time while maintaining a **hidden state** that summarizes what has been seen so far.

At each timestep t, the RNN:

1. Takes the current input x_t and the previous hidden state $\mathbf{h}_{t-1}$
2. Computes a new hidden state $\mathbf{h}_t$
3. Optionally produces an output $\mathbf{y}_t$

The core equation is:

$$\mathbf{h}_t = \sigma(\mathbf{W}_h \mathbf{h}_{t-1} + \mathbf{W}_x \mathbf{x}_t + \mathbf{b})$$

where $\mathbf{W}_h \in \mathbb{R}^{d \times d}$ is the hidden-to-hidden weight matrix, $\mathbf{W}_x \in \mathbb{R}^{d \times n}$ is the input-to-hidden weight matrix, $\mathbf{b} \in \mathbb{R}^d$ is the bias, and σ is typically tanh. The hidden dimension d is a hyperparameter.

The key insight: **the same weights are used at every timestep**. The network learns to process one element and update its state, then applies this same computation repeatedly. This is called **weight sharing** or **parameter tying**.

Let's trace through a concrete example. Suppose we have a tiny RNN with hidden dimension $d = 2$, processing a sequence of 3 scalar inputs: $x_1 = 1, x_2 = -1, x_3 = 2$. Let:

$$\mathbf{W}_h = \begin{bmatrix} 0.5 & 0.1 \\ 0.2 & 0.6 \end{bmatrix}, \quad \mathbf{W}_x = \begin{bmatrix} 0.3 \\ 0.4 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Initialize $\mathbf{h}_0 = [0, 0]^T$ (no prior context).

Timestep 1 ($x_1 = 1$):

$$\mathbf{h}_1 = \tanh\left(\begin{bmatrix}0.5 & 0.1\\ 0.2 & 0.6\end{bmatrix}\begin{bmatrix}0\\0\end{bmatrix} + \begin{bmatrix}0.3\\0.4\end{bmatrix}\cdot 1\right) = \tanh\begin{bmatrix}0.3\\0.4\end{bmatrix} = \begin{bmatrix}0.291\\0.380\end{bmatrix}$$

Timestep 2 ($x_2 = -1$):

$$\mathbf{h}_2 = \tanh\left(\begin{bmatrix}0.5 & 0.1\\ 0.2 & 0.6\end{bmatrix}\begin{bmatrix}0.291\\0.380\end{bmatrix} + \begin{bmatrix}0.3\\0.4\end{bmatrix}\cdot(-1)\right)$$

$$= \tanh\left(\begin{bmatrix}0.146+0.038\\0.058+0.228\end{bmatrix} + \begin{bmatrix}-0.3\\-0.4\end{bmatrix}\right) = \tanh\begin{bmatrix}-0.116\\-0.114\end{bmatrix} = \begin{bmatrix}-0.115\\-0.114\end{bmatrix}$$

Timestep 3 ($x_3 = 2$):

$$\mathbf{h}_3 = \tanh\left(\begin{bmatrix}0.5 & 0.1\\ 0.2 & 0.6\end{bmatrix}\begin{bmatrix}-0.115\\-0.114\end{bmatrix} + \begin{bmatrix}0.3\\0.4\end{bmatrix}\cdot 2\right)$$

$$= \tanh\left(\begin{bmatrix}-0.058-0.011\\-0.023-0.068\end{bmatrix} + \begin{bmatrix}0.6\\0.8\end{bmatrix}\right) = \tanh\begin{bmatrix}0.531\\0.709\end{bmatrix} = \begin{bmatrix}0.486\\0.610\end{bmatrix}$$

The final hidden state $\mathbf{h}_3 = [0.486, 0.610]^T$ encodes information about the entire sequence. Notice how each hidden state depends on all previous inputs through the chain of computations.

6.3.1 Unrolling the RNN

We can visualize an RNN by "unrolling" it through time:

$$\mathbf{h}_0 \xrightarrow{x_1} \mathbf{h}_1 \xrightarrow{x_2} \mathbf{h}_2 \xrightarrow{x_3} \cdots \xrightarrow{x_T} \mathbf{h}_T$$

Each arrow represents the same computation with the same weights. The unrolled network looks like a very deep feedforward network, but with tied weights across layers (timesteps).

6.3.2 Training RNNs: backpropagation through time

To train an RNN, we use a variant of backpropagation called **backpropagation through time** (BPTT). We unroll the network, compute the loss (e.g., at the final timestep or at each timestep), and backpropagate through the unrolled graph.

The gradient with respect to a parameter like $\mathbf{W}_h$ accumulates contributions from each timestep:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_h} = \sum_{t=1}^{T} \frac{\partial \mathcal{L}}{\partial \mathbf{h}_t}\frac{\partial \mathbf{h}_t}{\partial \mathbf{W}_h}$$

But here's where trouble begins. To compute $\frac{\partial \mathcal{L}}{\partial \mathbf{h}_1}$ when the loss depends on $\mathbf{h}_T$, we must trace the chain of dependencies:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{h}_1} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}_T}\cdot\frac{\partial \mathbf{h}_T}{\partial \mathbf{h}_{T-1}}\cdot\frac{\partial \mathbf{h}_{T-1}}{\partial \mathbf{h}_{T-2}}\cdots\frac{\partial \mathbf{h}_2}{\partial \mathbf{h}_1}$$

This is a product of $T-1$ Jacobian matrices. Let's examine what these Jacobians look like.

6.4 The vanishing and exploding gradient problem

From the RNN equation $\mathbf{h}_t = \tanh(\mathbf{W}_h \mathbf{h}_{t-1} + \mathbf{W}_x \mathbf{x}_t + \mathbf{b})$, the Jacobian of $\mathbf{h}_t$ with respect to $\mathbf{h}_{t-1}$ is:

$$\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}} = \operatorname{diag}\left(\frac{d}{dz}\tanh(\mathbf{z}_t)\right) \cdot \mathbf{W}_h$$

where $\mathbf{z}_t = \mathbf{W}_h \mathbf{h}_{t-1} + \mathbf{W}_x \mathbf{x}_t + \mathbf{b}$ and $\frac{d}{dz}\tanh(z) = 1 - \tanh^2(z)$. The diagonal matrix $\operatorname{diag}\left(\frac{d}{dz}\tanh(\mathbf{z}_t)\right)$ has entries in $(0, 1]$ since $\frac{d}{dz}\tanh \leq 1$.

The gradient from timestep T back to timestep 1 involves:

$$\frac{\partial \mathbf{h}_T}{\partial \mathbf{h}_1} = \prod_{t=2}^{T} \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}} = \prod_{t=2}^{T} \operatorname{diag}\left(\frac{d}{dz}\tanh(\mathbf{z}_t)\right) \cdot \mathbf{W}_h$$

This is a product of $T-1$ matrices. What happens to this product as T grows?

6.4.1 The problem mathematically

Consider the simplified case where all the $\operatorname{diag}\left(\frac{d}{dz}\tanh(\mathbf{z}_t)\right)$ matrices are roughly the identity (i.e., $\frac{d}{dz}\tanh$ values are close to 1). Then:

$$\frac{\partial \mathbf{h}_T}{\partial \mathbf{h}_1} \approx \mathbf{W}_h^{T-1}$$

Let $\mathbf{W}_h$ have eigenvalue decomposition $\mathbf{W}_h = \mathbf{V}\Lambda\mathbf{V}^{-1}$ where Λ is diagonal with eigenvalues $\lambda_1, \ldots, \lambda_d$. Then:

$$\mathbf{W}_h^{T-1} = \mathbf{V}\Lambda^{T-1}\mathbf{V}^{-1}$$

The eigenvalues get raised to the power $T-1$:

- If $|\lambda_i| < 1$: $\lambda_i^{T-1} \to 0$ exponentially fast. **Vanishing gradient.**
- If $|\lambda_i| > 1$: $\lambda_i^{T-1} \to \infty$ exponentially fast. **Exploding gradient.**
- If $|\lambda_i| = 1$: λ_i^{T-1} stays bounded. Stable.

For a typical random matrix, some eigenvalues will have magnitude greater than 1 and some less than 1. Over long sequences, the gradient either explodes or vanishes depending on which eigenvalues dominate.

6.4.2 Concrete example

Let's see this with numbers. Suppose $\mathbf{W}_h = \begin{bmatrix} 0.9 & 0 \\ 0 & 1.1 \end{bmatrix}$ (diagonal for simplicity, eigenvalues 0.9 and 1.1). After $T = 50$ timesteps:

$$\mathbf{W}_h^{49} = \begin{bmatrix} 0.9^{49} & 0 \\ 0 & 1.1^{49} \end{bmatrix} = \begin{bmatrix} 0.0052 & 0 \\ 0 & 97.02 \end{bmatrix}$$

The first component shrinks to nearly zero (vanishing). The second component grows to nearly 100 (exploding). After 100 timesteps, these become 2.7×10^{-5} and 9417 respectively. The imbalance is extreme.

In practice, the $\frac{d}{dz}\tanh$ factors make vanishing more common than exploding, since $\frac{d}{dz}\tanh \leq 1$ uniformly shrinks gradients. But either way, learning long-range dependencies becomes very difficult.

6.4.3 Why this matters

When gradients vanish, the network can't learn long-range dependencies. If the loss depends on the relationship between x_1 and x_T (like subject-verb agreement in a long sentence), the gradient signal from $\mathbf{h}_T$ is essentially zero by the time it reaches $\mathbf{h}_1$. The network has no way to learn that early inputs matter.

When gradients explode, training becomes unstable. Parameters get enormous updates and diverge. A common fix is **gradient clipping**: if $\|\nabla_\theta \mathcal{L}\| > \tau$ for some threshold τ, rescale the gradient to have norm τ. This prevents explosion but doesn't help with vanishing.

6.5 LSTM: a solution to vanishing gradients

The **Long Short-Term Memory** (LSTM) architecture addresses vanishing gradients by introducing a **cell state** $\mathbf{c}_t$ that acts as a "memory highway." Information can flow through the cell state with minimal transformation, allowing gradients to propagate over long distances.

An LSTM has three gates that control information flow:

Forget gate: Decides what to remove from the cell state.

$$\mathbf{f}_t = \sigma(\mathbf{W}_f[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_f)$$

Input gate: Decides what new information to add.

$$\mathbf{i}_t = \sigma(\mathbf{W}_i[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_i)$$

Output gate: Decides what to output from the cell state.

$$\mathbf{o}_t = \sigma(\mathbf{W}_o[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_o)$$

Here $[\mathbf{h}_{t-1}, \mathbf{x}_t]$ denotes concatenation and σ is the sigmoid function (outputting values in $(0, 1)$ that act as "soft switches").

The cell state update is:

$$\tilde{\mathbf{c}}_t = \tanh(\mathbf{W}_c[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_c)$$
$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t$$

The hidden state is:

$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t)$$

The key equation is $\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t$. When the forget gate $\mathbf{f}_t$ is close to 1 and the input gate $\mathbf{i}_t$ is close to 0, the cell state is simply copied: $\mathbf{c}_t \approx \mathbf{c}_{t-1}$. This additive update (rather than multiplicative) creates a gradient highway. The gradient $\frac{\partial \mathbf{c}_T}{\partial \mathbf{c}_1}$ can be close to 1 even for large T, as long as the forget gates stay open.

6.5.1 The gradient flow in LSTM

Let's trace the gradient through the cell state. We have:

$$\frac{\partial \mathbf{c}_t}{\partial \mathbf{c}_{t-1}} = \text{diag}(\mathbf{f}_t)$$

The Jacobian is just a diagonal matrix of forget gate values. If $\mathbf{f}_t \approx \mathbf{1}$ (forget gate fully open), then:

$$\frac{\partial \mathbf{c}_T}{\partial \mathbf{c}_1} = \prod_{t=2}^{T} \text{diag}(\mathbf{f}_t) \approx \mathbf{I}$$

Compare this to the vanilla RNN where we had products of $\mathbf{W}_h$ matrices. The LSTM's additive cell state update transforms the gradient from a product of matrices (which can vanish or explode) to a product of diagonal matrices with entries in $(0, 1)$ (which vanishes more slowly).

This is not a complete solution. If the forget gates are consistently below 1, gradients still eventually vanish. But LSTMs can learn to keep forget gates open for important information, allowing much longer effective memory than vanilla RNNs.

6.6 The limitations of recurrence

LSTMs and the related GRU (Gated Recurrent Unit) architecture significantly improved sequence modeling. They enabled breakthroughs in machine translation, speech recognition, and text generation. But they have fundamental limitations:

Sequential computation. An RNN must process tokens one at a time because $\mathbf{h}_t$ depends on $\mathbf{h}_{t-1}$. For a sequence of length T, we need T sequential steps. This can't be parallelized, making training slow on modern GPUs that excel at parallel computation.

Long-range dependencies remain hard. Even LSTMs struggle with very long sequences. The gradient must still flow through many timesteps, and information must be compressed into a fixed-size hidden state. Empirically, LSTMs work well for dependencies spanning tens or low hundreds of tokens, but struggle beyond that.

The bottleneck problem. All information about the past must be compressed into the hidden state vector $\mathbf{h}_t$. For tasks like machine translation, the encoder must compress an entire source sentence into a single vector before the decoder can start. Important details get lost.

Consider translating a long English sentence to French. An RNN encoder produces a single vector representing the whole sentence. The decoder must reconstruct the French sentence from this compressed representation. Early words in the source get processed first, potentially getting "overwritten" by later words in the hidden state.

These limitations motivated the search for architectures that could:

1. Process sequences in parallel rather than sequentially
2. Directly connect any position to any other position, regardless of distance
3. Avoid compressing everything into a fixed bottleneck

The solution was **attention**: a mechanism that lets the model dynamically focus on different parts of the input. Attention was first used as an add-on to RNNs, dramatically improving machine translation. The transformer architecture then showed that attention alone, without any recurrence, could achieve state-of-the-art results while being much faster to train.

In the next part of this book, we'll build up to the transformer by first understanding embeddings (how we represent discrete tokens as vectors) and then developing the attention mechanism in detail.

6.7 Summary

We've seen that:

- Sequential data has properties (variable length, order dependence, long-range dependencies) that standard feedforward networks can't handle well.
- RNNs process sequences by maintaining a hidden state that updates at each timestep, with shared weights across time.
- Training RNNs involves backpropagation through time, which leads to products of many matrices.
- These products cause gradients to vanish or explode, making long-range dependencies hard to learn.
- LSTMs mitigate vanishing gradients with additive cell state updates and gating mechanisms.

- But recurrence has fundamental limitations: sequential computation, remaining difficulty with very long range dependencies, and information bottlenecks.

The transformer architecture, which we'll build toward over the next several chapters, addresses all these limitations by replacing recurrence with attention.

Part II

Building Blocks

Chapter 7

Embeddings

Learning objectives

After completing this chapter, you will be able to:

- Explain why one-hot encoding is insufficient for representing words
- Describe how embedding matrices map discrete tokens to continuous vectors
- Compute cosine similarity between word embeddings
- Understand the Skip-gram objective for learning embeddings from context
- Apply byte-pair encoding (BPE) for subword tokenization

Before a neural network can process text, it needs numbers. Words like "cat," "dog," and "philosophy" must become vectors that capture their meanings. This chapter develops the theory of embeddings: dense, learned representations that map discrete tokens into continuous vector spaces where similar concepts are nearby.

7.1 The problem with discrete tokens

Consider a vocabulary of V words. The simplest way to convert words to numbers is **one-hot encoding**: represent word i as a vector $\mathbf{e}_i \in \mathbb{R}^V$ with a 1 in position i and 0s elsewhere.

For a vocabulary of 50,000 words:

$$\text{cat} \to [0, 0, \ldots, 1, \ldots, 0, 0]^T \quad (1 \text{ in position } 3{,}142)$$

$$\text{dog} \to [0, 0, \ldots, 1, \ldots, 0, 0]^T \quad (1 \text{ in position } 7{,}891)$$

This representation has serious problems:

High dimensionality. Each vector has dimension V, which can be tens or hundreds of thousands. This is computationally expensive.

Sparsity. Each vector has exactly one nonzero entry. Most computation involves multiplying by zeros.

No similarity structure. The dot product of any two different one-hot vectors is zero: $\mathbf{e}_i^T \mathbf{e}_j = 0$ for $i \neq j$. By this metric, "cat" is equally dissimilar to "dog" and to "philosophy." There's no notion that some words are more related than others.

No generalization. If the network learns something about "cat," that knowledge doesn't transfer to "dog" because their representations share no structure. The network must learn everything about every word from scratch.

What we want is a representation where:

- Vectors are dense and low-dimensional (say, 256 or 512 dimensions instead of 50,000)
- Similar words have similar vectors
- Relationships between words are encoded geometrically

7.2 Word embeddings

A **word embedding** maps each word to a dense vector. Instead of 50,000-dimensional one-hot vectors, we use vectors with perhaps 256 or 512 dimensions that we learn from data.

We store all word embeddings in a single matrix $\mathbf{E} \in \mathbb{R}^{V \times d}$, where V is the vocabulary size (number of words, e.g., 50,000) and d is the embedding dimension (e.g., 256). Each row of $\mathbf{E}$ contains the embedding for one word: row i holds the embedding for word i. So if "cat" has index 3142 in our vocabulary, then row 3142 of $\mathbf{E}$ is the embedding vector for "cat".

To look up an embedding, we retrieve row i of $\mathbf{E}$. Mathematically, this can be written as a matrix multiplication with a one-hot vector:

$$\mathbf{e}_i = \mathbf{E}^T \mathbf{x}$$

Here $\mathbf{x} \in \mathbb{R}^V$ is the one-hot vector for word i (all zeros except a 1 at position i), $\mathbf{E}^T \in \mathbb{R}^{d \times V}$ is the transposed embedding matrix, and $\mathbf{e}_i \in \mathbb{R}^d$ is the resulting embedding vector.

Why does this work? When you multiply $\mathbf{E}^T$ by a one-hot vector, you're computing a weighted sum of columns of $\mathbf{E}^T$, but all weights are zero except one. So you just select the i-th column of $\mathbf{E}^T$, which is the i-th row of $\mathbf{E}$. That's exactly the embedding we want.

In practice, we skip the matrix multiplication and just look up row i directly. But the matrix formulation matters when we compute gradients during training.

Let's see a concrete example. Suppose we have a tiny vocabulary of 5 words and embeddings of dimension 3:

Word	Index	Embedding
cat	0	$[0.2, 0.8, -0.1]$
dog	1	$[0.3, 0.7, -0.2]$
fish	2	$[0.1, 0.9, 0.3]$
car	3	$[-0.5, 0.1, 0.6]$
truck	4	$[-0.4, 0.2, 0.5]$

The embedding matrix is:

$$\mathbf{E} = \begin{bmatrix} 0.2 & 0.8 & -0.1 \\ 0.3 & 0.7 & -0.2 \\ 0.1 & 0.9 & 0.3 \\ -0.5 & 0.1 & 0.6 \\ -0.4 & 0.2 & 0.5 \end{bmatrix}$$

Notice that "cat" and "dog" have similar embeddings (both are animals), and "car" and "truck" have similar embeddings (both are vehicles). The embedding space captures semantic relationships.

7.2.1 Measuring similarity

In a good embedding space, similar words have similar vectors. But what does "similar vectors" mean? We measure it with **cosine similarity**:

$$\text{sim}(\mathbf{u}, \mathbf{v}) = \frac{\mathbf{u}^T\mathbf{v}}{\|\mathbf{u}\|\|\mathbf{v}\|}$$

Here $\mathbf{u}$ and $\mathbf{v}$ are two embedding vectors in $\mathbb{R}^d$. The numerator $\mathbf{u}^T\mathbf{v} = \sum_{i=1}^{d} u_i v_i$ is their dot product, and the denominator $\|\mathbf{u}\|\|\mathbf{v}\|$ multiplies their lengths (where $\|\mathbf{u}\| = \sqrt{\sum_{i=1}^{d} u_i^2}$). Dividing by both lengths normalizes for magnitude, so we only care about *direction*, not how long the vectors are.

Geometrically, cosine similarity equals $\cos\theta$ where θ is the angle between the vectors. When sim $= +1$, the vectors point in the same direction (angle $= 0°$). When sim $= 0$, they're perpendicular (angle $= 90°$). When sim $= -1$, they point in opposite directions (angle $= 180°$).

For our example embeddings:

$$\text{sim}(\text{cat}, \text{dog}) = \frac{0.2 \cdot 0.3 + 0.8 \cdot 0.7 + (-0.1)(-0.2)}{\sqrt{0.04 + 0.64 + 0.01}\sqrt{0.09 + 0.49 + 0.04}}$$

$$= \frac{0.06 + 0.56 + 0.02}{\sqrt{0.69}\sqrt{0.62}} = \frac{0.64}{0.831 \cdot 0.787} = \frac{0.64}{0.654} \approx 0.98$$

Very high similarity! Let's compare "cat" and "car":

$$\text{sim}(\text{cat}, \text{car}) = \frac{0.2 \cdot (-0.5) + 0.8 \cdot 0.1 + (-0.1) \cdot 0.6}{\sqrt{0.69}\sqrt{0.25 + 0.01 + 0.36}}$$

$$= \frac{-0.1 + 0.08 - 0.06}{\sqrt{0.69}\sqrt{0.62}} = \frac{-0.08}{0.654} \approx -0.12$$

Much lower (even slightly negative). The embedding space captures that "cat" and "dog" are more similar to each other than either is to "car."

7.2.2 The geometry of meaning

Good embeddings have remarkable geometric properties. The most famous is **analogical reasoning**:

$$\mathbf{e}_{\text{king}} - \mathbf{e}_{\text{man}} + \mathbf{e}_{\text{woman}} \approx \mathbf{e}_{\text{queen}}$$

where $\mathbf{e}_{\text{word}}$ denotes the embedding vector for that word.

What does this equation say? Rearranging: $\mathbf{e}_{\text{king}} - \mathbf{e}_{\text{man}} \approx \mathbf{e}_{\text{queen}} - \mathbf{e}_{\text{woman}}$. The vector *difference* from "man" to "king" is approximately equal to the vector difference from "woman" to "queen". Both differences point in the "royalty" direction.

Why does this work? During training, words that appear in similar contexts get similar embeddings. "King" and "queen" appear in similar contexts (ruling, crowns, thrones), so they're close. But "king" also appears in contexts similar to "man" (he, his, himself), while "queen" appears in contexts similar to "woman" (she, her, herself). The embedding space learns to separate these orthogonal dimensions: one for gender, one for royalty, and they combine additively.

This isn't magic or reasoning. It's a geometric consequence of how embeddings are trained. If the training data consistently uses masculine pronouns with "king" and feminine pronouns with "queen," the difference vectors $\mathbf{e}_{\text{king}} - \mathbf{e}_{\text{man}}$ and $\mathbf{e}_{\text{queen}} - \mathbf{e}_{\text{woman}}$ will both point in the "royalty" direction.

7.3 Learning embeddings

Where do embeddings come from? They're learned from data. This section covers the classical approach (Word2Vec) which, while no longer state-of-the-art, provides essential intuition for understanding how neural networks learn meaningful representations.

7.3.1 The distributional hypothesis

The foundation of learned embeddings is the **distributional hypothesis**: words that appear in similar contexts have similar meanings. "You shall know a word by the company it keeps."

Consider the blanks in:

- "The _____ sat on the mat."
- "The _____ chased the mouse."
- "I fed my _____ some treats."

Words like "cat," "dog," and "hamster" could fill these blanks. They appear in similar contexts, so they should have similar embeddings. Words like "democracy" or "algorithm" wouldn't fit these contexts, so they should have different embeddings.

7.3.2 Skip-gram: predicting context from words

Historical note: Skip-gram (2013) is not used in modern transformers. We cover it because it builds intuition for how embeddings are learned. Modern transformers learn embeddings end-to-end as part of the full model, not as a separate pre-training step.

The **Skip-gram** model (part of Word2Vec) learns embeddings by training a neural network to predict context words given a center word.

The setup. Given a sentence, we slide a window across it. At each position, the word in the middle is the "center word" and the surrounding words are "context words".

For example, in "the cat sat on the mat" with window size $c = 2$:

Center word	Context words (within 2 positions)
sat	the, cat, on, the

The model learns to predict context words from the center word.

Interestingly, the model uses *two* separate embedding matrices: $\mathbf{W} \in \mathbb{R}^{V \times d}$ for words when they're the center, and $\mathbf{W}' \in \mathbb{R}^{V \times d}$ for words when they're in the context. Here V is the vocabulary size and d is the embedding dimension. We write $\mathbf{w}_i$ for row i of $\mathbf{W}$ and $\mathbf{w}'_i$ for row i of $\mathbf{W}'$.

Given a center word w_c, what's the probability that w_o appears in its context? The model computes:

$$P(w_o|w_c) = \frac{\exp({\mathbf{w}'_o}^T \mathbf{w}_c)}{\sum_{w=1}^{V} \exp({\mathbf{w}'_w}^T \mathbf{w}_c)}$$

Here $\mathbf{w}_c \in \mathbb{R}^d$ is the center word's embedding (from $\mathbf{W}$) and $\mathbf{w}'_o \in \mathbb{R}^d$ is the context word's embedding (from $\mathbf{W}'$). The numerator computes the dot product ${\mathbf{w}'_o}^T \mathbf{w}_c$. When this is high, the vectors point in similar directions, indicating a likely context word. The denominator sums over all V words in the vocabulary, turning the scores into a proper probability distribution (this is the softmax).

The training objective is to maximize the probability of the context words we actually observe in the corpus:

$$\mathcal{L} = \sum_{(w_c, w_o) \in D} \log P(w_o | w_c)$$

where D is the set of all (center, context) pairs extracted from the training text. We take the log because it converts products to sums, which is easier to optimize with gradient descent.

Why does this learn good embeddings? To make $P(w_o|w_c)$ large, the model needs the dot product $\mathbf{w}'_o{}^T \mathbf{w}_c$ to be large. If "cat" often appears near "pet," "fur," and "purr," then $\mathbf{w}_{\text{cat}}$ must point in a direction that has high dot product with $\mathbf{w}'_{\text{pet}}$, $\mathbf{w}'_{\text{fur}}$, and $\mathbf{w}'_{\text{purr}}$. Similarly, "dog" also appears near "pet" and "fur" (though not "purr", maybe "bark" instead). So $\mathbf{w}_{\text{dog}}$ must also point toward $\mathbf{w}'_{\text{pet}}$ and $\mathbf{w}'_{\text{fur}}$. This forces "cat" and "dog" to have similar embeddings because they're being pulled toward the same context words.

7.3.3 Negative sampling

There's a computational problem with skip-gram. The softmax denominator sums over all V words in the vocabulary: $\sum_{w=1}^{V} \exp(\mathbf{w}'_w{}^T \mathbf{w}_c)$. With $V = 50{,}000$ words, that's 50,000 dot products per training example. Far too slow.

Negative sampling sidesteps this by changing the question. Instead of asking "which of 50,000 words is the context?" (a V-way classification), we ask "is this specific word a real context word, or a random fake?" (a binary classification).

The idea is simple. For each real (center, context) pair we observe in the corpus, we also grab k random words that *aren't* in the context. The real context word is a **positive sample**; the random words are **negative samples**. We train the model to say "yes" to the positive and "no" to the negatives.

This transforms our objective. For center word w_c, real context word w_o, and k random negative words $w_1, \ldots, w_k$, we maximize:

$$\mathcal{L} = \log \sigma(\mathbf{w}'_o{}^T \mathbf{w}_c) + \sum_{i=1}^{k} \log \sigma(-\mathbf{w}'_{w_i}{}^T \mathbf{w}_c)$$

Here $\sigma(x) = \frac{1}{1+e^{-x}}$ is the sigmoid function, which squashes any real number to a probability between 0 and 1. The first term handles the positive sample: $\mathbf{w}'_o{}^T \mathbf{w}_c$ is the dot product between center and context embeddings, and we want σ of this to be close to 1, meaning "yes, this is a real context word." The sum handles negatives: for each fake word w_i, we take the *negative* of the dot product before applying σ, because we want the model to output high confidence that these are *not* context words.

Why does this give the same learning signal as softmax? Think about what the model must do. To make $\sigma(\mathbf{w}'_o{}^T \mathbf{w}_c)$ large, the dot product between real context words and the center must be large, meaning they need to point in similar directions. To make $\sigma(-\mathbf{w}'_{w_i}{}^T \mathbf{w}_c)$ large, the dot product for random words must be small or negative. The model learns to pull real context words close and push random words away. Same outcome as softmax, but we only compute $k + 1$ dot products instead of V. Typically k is between 5 and 20.

7.3.4 A worked example

Let's trace through one training step to make this concrete. Our corpus is the sentence "the cat sat on the mat" and we're using a context window of size 1 (one word on each side of the center). For illustration, we'll use tiny 3-dimensional embeddings.

When the center word is "cat", the context words are "the" (to the left) and "sat" (to the right). This gives us two training pairs: (cat, the) and (cat, sat). Let's process the first one.

Suppose the embeddings are currently (after random initialization):

$$\mathbf{w}_{\text{cat}} = \begin{bmatrix} 0.1 \\ 0.2 \\ -0.1 \end{bmatrix}, \quad \mathbf{w}'_{\text{the}} = \begin{bmatrix} 0.3 \\ 0.1 \\ 0.2 \end{bmatrix}$$

We compute the dot product to see how aligned these vectors are:

$$\mathbf{w}'_{\text{the}}{}^T \mathbf{w}_{\text{cat}} = 0.3 \times 0.1 + 0.1 \times 0.2 + 0.2 \times (-0.1) = 0.03 + 0.02 - 0.02 = 0.03$$

The dot product is 0.03, barely positive. Passing through the sigmoid: $\sigma(0.03) = \frac{1}{1+e^{-0.03}} \approx 0.507$. The model predicts a 50.7% chance that "the" is a context word of "cat", essentially a coin flip. Since "the" actually *is* a context word, this is a poor prediction. Gradient descent will nudge both embeddings to increase their dot product, making them more aligned.

Now for a negative sample. We randomly pick a word that isn't in the context of "cat", say "algorithm". Its embedding is $\mathbf{w}'_{\text{algorithm}} = [0.5, -0.3, 0.1]^T$.

The dot product is:

$$\mathbf{w}'_{\text{algorithm}}{}^T \mathbf{w}_{\text{cat}} = 0.5 \times 0.1 + (-0.3) \times 0.2 + 0.1 \times (-0.1) = 0.05 - 0.06 - 0.01 = -0.02$$

For negative samples, we want the model to confidently say "no, this is not a context word." We achieve this by feeding the *negative* of the dot product into the sigmoid: $\sigma(-(-0.02)) = \sigma(0.02) \approx 0.505$. This is slightly above 0.5, meaning the model has a slight inclination that "algorithm" isn't a context word. Gradient descent will push the dot product more negative, increasing this confidence.

After millions of such updates across a large corpus, a pattern emerges. Words that frequently appear together develop aligned embeddings (high dot products). Words that rarely co-occur develop orthogonal or opposing embeddings (low or negative dot products). And crucially, words that share similar contexts, like "cat" and "dog" which both appear near "pet", "fur", and "fed", end up with similar embeddings because they're both being pulled toward the same set of context words.

7.4 Subword tokenization

Word-level embeddings have a problem: what about words not in the vocabulary? Misspellings, rare words, technical jargon, new slang, and morphological variants (run, running, runner) all need handling. A vocabulary of 50,000 words sounds large, but it can't cover everything.

Subword tokenization solves this by splitting words into smaller pieces that can be combined. The vocabulary contains common words whole, but rare words get broken into subword units. This way, even a word the model has never seen can be represented as a combination of familiar pieces.

7.4.1 Byte Pair Encoding (BPE)

State of the art: BPE (2015) and its variants are used in most modern large language models, including GPT-2, GPT-3, GPT-4, and LLaMA.

Byte Pair Encoding builds a vocabulary by starting small and iteratively growing it. The algorithm begins with a vocabulary of just individual characters: every letter, digit, and punctuation mark. Then it scans the training corpus, counts how often each pair of adjacent tokens appears, and merges the most frequent pair into a new token. This process repeats until the vocabulary reaches the desired size.

Let's trace through a tiny example. Suppose our corpus is just three words: "low", "lower", and "lowest". We start by representing these as character sequences:

low → `l o w` lower → `l o w e r` lowest → `l o w e s t`

The initial vocabulary is $\{\texttt{l}, \texttt{o}, \texttt{w}, \texttt{e}, \texttt{r}, \texttt{s}, \texttt{t}\}$, just the characters that appear. Now we count adjacent pairs across the corpus. The pair $(\texttt{l}, \texttt{o})$ appears 3 times (once in each word). The pair $(\texttt{o}, \texttt{w})$ also appears 3 times. Suppose we break ties alphabetically and merge $(\texttt{l}, \texttt{o})$ first.

After the merge, our vocabulary grows to $\{\texttt{l}, \texttt{o}, \texttt{w}, \texttt{e}, \texttt{r}, \texttt{s}, \texttt{t}, \texttt{lo}\}$, and the corpus becomes:

$$\text{low} \to \texttt{lo w} \qquad \text{lower} \to \texttt{lo w e r} \qquad \text{lowest} \to \texttt{lo w e s t}$$

We count pairs again. Now $(\texttt{lo}, \texttt{w})$ appears 3 times, the most frequent. We merge it:

$$\text{low} \to \texttt{low} \qquad \text{lower} \to \texttt{low e r} \qquad \text{lowest} \to \texttt{low e s t}$$

Continuing this process, we might eventually merge $(\texttt{e}, \texttt{r})$ to get `er`, then $(\texttt{e}, \texttt{s})$ to get `es`, then $(\texttt{es}, \texttt{t})$ to get `est`, and so on. The final vocabulary might include tokens like `low`, `lower`, `lowest`, `er`, and `est`.

At inference time, we tokenize new text using the learned merges. A common word like "lowest" might become a single token [`lowest`]. A rare word like "lowest-ever" might become [`lowest`, `-`, `ever`]. And a completely novel word like "transformerize" might become [`transform`, `er`, `ize`]. Each piece is familiar even though the whole word is new.

7.4.2 Properties of subword tokenization

Subword tokenization has several appealing properties. First, it handles unknown words gracefully. Even a word the model has never encountered can be broken into subword units that the model *has* seen. The word "transformerize" doesn't need to be in the vocabulary; its pieces "transform", "er", and "ize" are enough.

Second, related words share subword units, which means they share part of their representation. The words "playing", "played", and "player" might all contain the token "play". This gives the model a head start on understanding new morphological variants. If it knows what "play" means, it has a foundation for understanding "playable" even without seeing that exact word in training.

Third, the vocabulary size stays manageable. Common words get single tokens, which is efficient. Rare words get split into multiple tokens, which takes more computation but ensures everything is representable. A vocabulary of 50,000 subword tokens can effectively cover far more than 50,000 words.

Modern transformers use variants of this idea. **WordPiece** (used in BERT) is similar to BPE but uses a slightly different scoring function for merges. **SentencePiece** (used in T5 and LLaMA) operates directly on raw text without pre-tokenization, making it language-agnostic. The exact algorithms differ, but the principle is the same: learn a vocabulary of subword units that balances coverage and efficiency.

7.4.3 Token embeddings in transformers

In a transformer, each subword token has a learned embedding stored in the embedding matrix $\mathbf{E} \in \mathbb{R}^{V \times d}$, where V is the vocabulary size (number of unique tokens) and d is the model dimension (embedding size). The total number of parameters in the embedding layer is $V \times d$.

How big is this in practice? Let's compute for real models:

Model	V (vocab)	d (dimension)	Embedding parameters
GPT-2	50,257	768	38.6 million
GPT-3	50,257	12,288	617 million

GPT-3's embedding matrix alone has more parameters than entire earlier models.

How are they trained? The embedding matrix $\mathbf{E}$ is initialized randomly and trained jointly with the rest of the model via backpropagation. The training signal comes from the language modeling objective: predict the next token. Embeddings that help make good predictions survive; others get updated.

7.5 Properties of transformer embeddings

State of the art: The approach described here, learned embeddings combined with contextual attention layers, is how all modern large language models work (GPT-4, Claude, LLaMA, etc.).

Embeddings in transformers have interesting properties that emerge from training.

7.5.1 Contextual vs. static embeddings

Word2Vec embeddings are **static**: "bank" has one embedding regardless of context. But "bank" means different things in "river bank" and "bank account."

Transformer embeddings start as static (the lookup from $\mathbf{E}$), but then get transformed by the attention layers into **contextual embeddings**. After passing through transformer layers, the representation of "bank" depends on the surrounding words. In "river bank," it might be close to "shore" and "water." In "bank account," it might be close to "money" and "finance."

The initial embedding $\mathbf{E}$ provides a starting point. The attention layers modify it based on context. This is one of the key innovations of transformers over earlier approaches.

7.5.2 Embedding space geometry

Studies of transformer embedding spaces reveal structure:

Linear subspaces for concepts. Directions in embedding space often correspond to interpretable concepts. There might be a "gender direction," a "tense direction," a "formality direction."

Clustering by meaning. Words with similar meanings cluster together. Synonyms are close. Categories form regions.

Analogies still work. The word2vec-style analogies often work in transformer embeddings too, though the relationship is more complex because embeddings become contextual after attention.

Anisotropy. Transformer embeddings often occupy a narrow cone rather than filling the space uniformly. This means cosine similarities tend to be high even for unrelated words. Various normalization techniques address this.

7.6 From tokens to sequences

So far we've embedded single tokens. But transformers process *sequences*. Here's how we go from a sentence to a matrix.

First, we tokenize the text into token indices. For example, "The cat sat" might become [464, 3797, 3332], where each number is an index in the vocabulary. Next, we look up each token's embedding from the embedding matrix $\mathbf{E}$. Finally, we stack these embeddings into a matrix:

$$\mathbf{X} = \begin{bmatrix} \mathbf{e}_{464} \\ \mathbf{e}_{3797} \\ \mathbf{e}_{3332} \end{bmatrix} \in \mathbb{R}^{T \times d}$$

Here $T = 3$ is the sequence length (number of tokens), d is the embedding dimension (e.g., 768), and row t of $\mathbf{X}$ is the embedding for token t. For "The cat sat" with $d = 768$, we get a 3×768 matrix. Each row is a 768-dimensional vector representing one token.

This matrix $\mathbf{X}$ is the input to the transformer. The attention layers will transform it, letting tokens "see" each other and build context-aware representations. But that's for later chapters. The embedding layer's job is done: convert token indices to vectors.

7.7 Summary

We've seen that:

- Discrete tokens need to become vectors for neural network processing. One-hot encodings are sparse, high-dimensional, and lack similarity structure.
- Word embeddings map tokens to dense vectors where similar words are nearby. The embedding matrix $\mathbf{E}$ is learned from data.
- Embeddings are learned by predicting words from context (skip-gram) or context from words (CBOW), based on the distributional hypothesis.
- Subword tokenization (BPE, WordPiece) handles rare and unknown words by decomposing them into learned subword units.
- Transformer embeddings start static but become contextual after passing through attention layers.

The embedding layer is where text first touches the transformer. It converts a sequence of tokens into a sequence of vectors that the attention mechanism can then relate and transform. In the next chapter, we'll see how attention works.

Chapter 8

The attention mechanism

Learning objectives

After completing this chapter, you will be able to:

- Explain the information bottleneck problem in sequence-to-sequence models
- Define queries, keys, and values and their roles in attention
- Compute scaled dot-product attention scores and weights
- Derive why the scaling factor $\sqrt{d_k}$ prevents vanishing gradients
- Implement the complete attention mechanism in matrix form

Attention is the core innovation that makes transformers work. It's a mechanism for selectively combining information from different positions based on relevance. Instead of compressing a sequence into a fixed-size vector, attention lets a model look back at all positions and decide dynamically which ones matter for the current computation. This chapter develops attention from first principles.

8.1 The bottleneck problem revisited

Recall from Chapter 3 that RNN-based sequence-to-sequence models have a bottleneck: the encoder compresses the entire input sequence into a single vector, which the decoder must use to generate the output.

$$\text{input sequence} \to \text{encoder} \to \text{single vector} \to \text{decoder} \to \text{output sequence}$$

For long sequences, this single vector can't capture all the relevant information. Early parts of the input get "overwritten" by later parts.

Attention solves this by giving the decoder direct access to all encoder hidden states. At each decoding step, the decoder can "attend to" different parts of the input, focusing on what's relevant for the current output.

8.2 The basic idea: weighted combinations

Suppose we have a sequence of vectors $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n$ (the "values") and we want to produce a single output vector that combines them. In a transformer processing the sentence "The cat sat on the mat," these might be $\mathbf{v}_1$ for "The", $\mathbf{v}_2$ for "cat", $\mathbf{v}_3$ for "sat", and so on. Each vector is a learned representation of that token (we'll see exactly how these are computed later).

The simplest way to combine them is averaging:

$$\mathbf{o} = \frac{1}{n}\sum_{i=1}^{n}\mathbf{v}_i$$

But this treats all positions equally. What if some positions are more relevant than others for our current purpose? When computing the representation for "sat," we might care more about "cat" (the subject) than "the" (a function word).

Attention computes a **weighted average** where the weights depend on relevance:

$$\mathbf{o} = \sum_{i=1}^{n}\alpha_i\mathbf{v}_i$$

where $\alpha_1, \ldots, \alpha_n$ are **attention weights** that sum to 1: $\sum_i \alpha_i = 1$ and $\alpha_i \geq 0$.

If $\alpha_3 = 0.7$ and all other α_i are small, the output is dominated by $\mathbf{v}_3$. We're "paying attention" to position 3. The key question is: how do we compute the weights?

8.3 Queries, keys, and values

Attention uses three concepts: **queries**, **keys**, and **values**. Think of it like a soft database lookup. The **query** is what we're looking for, representing the current position's "question" to the rest of the sequence. The **keys** label what's available at each position, advertising what each position offers. The **values** are the actual content at each position, the information that gets retrieved.

The query compares against each key to determine relevance (the attention weights), then the weights are used to combine the values.

Concretely, suppose we have a query vector $\mathbf{q} \in \mathbb{R}^d$ representing what we're looking for, key vectors $\mathbf{k}_1, \ldots, \mathbf{k}_n \in \mathbb{R}^d$ representing what each of n positions offers, and value vectors $\mathbf{v}_1, \ldots, \mathbf{v}_n \in \mathbb{R}^{d_v}$ containing the actual content at each position. Here d is the dimension of queries and keys (they must match for the dot product), and d_v is the dimension of values (which can differ).

The attention weight for position i measures how well the query matches key i:

$$\alpha_i = \frac{\exp(s(\mathbf{q}, \mathbf{k}_i))}{\sum_{j=1}^{n}\exp(s(\mathbf{q}, \mathbf{k}_j))}$$

where $s(\mathbf{q}, \mathbf{k})$ is a **score function** measuring similarity. This is a softmax over scores, ensuring weights are positive and sum to 1.

The output is the weighted combination of values:

$$\mathbf{o} = \sum_{i=1}^{n}\alpha_i\mathbf{v}_i$$

8.4 The score function: scaled dot product

The most common score function in transformers is the **scaled dot product**:

$$s(\mathbf{q}, \mathbf{k}) = \frac{\mathbf{q}^T\mathbf{k}}{\sqrt{d}}$$

where d is the dimension of the query and key vectors.

Why the dot product? It measures alignment: $\mathbf{q}^T\mathbf{k}$ is large when $\mathbf{q}$ and $\mathbf{k}$ point in similar directions. If the query represents "what I'm looking for" and the key represents "what this position offers," a high dot product means a good match.

Why scale by $\sqrt{d}$? Consider what happens without scaling. If $\mathbf{q}$ and $\mathbf{k}$ have entries drawn from a distribution with mean 0 and variance 1, then $\mathbf{q}^T\mathbf{k} = \sum_{i=1}^{d} q_i k_i$ has variance approximately d (sum of d terms, each with variance 1). For large d, the dot products can have large magnitude.

Large dot products cause the softmax to saturate. If one score is 100 and others are around 0, then $\exp(100)$ dominates and the softmax puts essentially all weight on that position. The gradients through saturated softmax are very small, making learning difficult.

Dividing by $\sqrt{d}$ keeps the variance around 1 regardless of dimension, preventing saturation.

8.4.1 Concrete example

Let's compute attention step by step. Suppose $d = 3$ and we have:

Query: $\mathbf{q} = [1, 0, 1]^T$

Keys: $\mathbf{k}_1 = [1, 1, 0]^T$, $\mathbf{k}_2 = [0, 1, 1]^T$, $\mathbf{k}_3 = [1, 0, 1]^T$

Values: $\mathbf{v}_1 = [1, 2]^T$, $\mathbf{v}_2 = [3, 4]^T$, $\mathbf{v}_3 = [5, 6]^T$

Step 1: Compute scores

$$s_1 = \frac{\mathbf{q}^T\mathbf{k}_1}{\sqrt{3}} = \frac{1 \cdot 1 + 0 \cdot 1 + 1 \cdot 0}{\sqrt{3}} = \frac{1}{1.732} \approx 0.577$$

$$s_2 = \frac{\mathbf{q}^T\mathbf{k}_2}{\sqrt{3}} = \frac{1 \cdot 0 + 0 \cdot 1 + 1 \cdot 1}{\sqrt{3}} = \frac{1}{1.732} \approx 0.577$$

$$s_3 = \frac{\mathbf{q}^T\mathbf{k}_3}{\sqrt{3}} = \frac{1 \cdot 1 + 0 \cdot 0 + 1 \cdot 1}{\sqrt{3}} = \frac{2}{1.732} \approx 1.155$$

Step 2: Compute attention weights (softmax)

$$\alpha_1 = \frac{\exp(0.577)}{\exp(0.577) + \exp(0.577) + \exp(1.155)} = \frac{1.781}{1.781 + 1.781 + 3.174} = \frac{1.781}{6.736} \approx 0.264$$

$$\alpha_2 = \frac{1.781}{6.736} \approx 0.264$$

$$\alpha_3 = \frac{3.174}{6.736} \approx 0.471$$

Notice that α_3 is largest because $\mathbf{q}$ and $\mathbf{k}_3$ are identical (maximum alignment).

Step 3: Compute output

$$\mathbf{o} = 0.264 \begin{bmatrix} 1 \\ 2 \end{bmatrix} + 0.264 \begin{bmatrix} 3 \\ 4 \end{bmatrix} + 0.471 \begin{bmatrix} 5 \\ 6 \end{bmatrix}$$

$$= \begin{bmatrix} 0.264 + 0.792 + 2.355 \\ 0.528 + 1.056 + 2.826 \end{bmatrix} = \begin{bmatrix} 3.41 \\ 4.41 \end{bmatrix}$$

The output $[3.41, 4.41]$ is a blend of all three value vectors, weighted toward $\mathbf{v}_3 = [5, 6]$ because the query best matches $\mathbf{k}_3$.

What is this output useful for? In a transformer, this becomes the new representation for the position that issued the query. If position 1 queries positions 1, 2, and 3, the output replaces position 1's old embedding with a new one that incorporates relevant information from all three positions. The original embedding only knew about itself; the new embedding is context-aware, having gathered information from wherever the attention weights pointed.

In this example, the output $[3.41, 4.41]$ is closer to $\mathbf{v}_3 = [5, 6]$ than to $\mathbf{v}_1 = [1, 2]$ because position 3 was deemed most relevant. If these were word embeddings in a sentence, the querying word now "knows about" the other words, weighted by relevance.

8.5 Matrix formulation

When we have multiple queries, we can compute attention for all of them in parallel using matrix operations. We stack the queries into a matrix $\mathbf{Q} \in \mathbb{R}^{m \times d}$ where each of the m rows is a query vector. Similarly, we have $\mathbf{K} \in \mathbb{R}^{n \times d}$ containing n key vectors as rows, and $\mathbf{V} \in \mathbb{R}^{n \times d_v}$ containing n value vectors as rows.

The **scaled dot-product attention** computes:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d}}\right)\mathbf{V}$$

State of the art: This exact formula is used in all modern transformers, including GPT-4, Claude, LLaMA, and BERT. The scaled dot-product attention mechanism was introduced in the original "Attention Is All You Need" paper (2017).

Let's unpack this formula step by step. First, the matrix product $\mathbf{Q}\mathbf{K}^T$ gives us an $m \times n$ matrix where entry (i, j) is the dot product of query i with key j. This computes all pairwise similarity scores in one operation. Next, we divide by $\sqrt{d}$ to prevent the dot products from growing too large (as discussed earlier). Then, softmax is applied to each row independently, so row i becomes a probability distribution representing how much query i attends to each of the n keys. Finally, we multiply by $\mathbf{V}$: each row of the result is a weighted combination of value vectors, using that row's attention weights.

The output has shape $m \times d_v$: for each of the m queries, we get a d_v-dimensional output vector.

8.5.1 Matrix example

Using the same values as before, with our single query as a 1×3 matrix:

$$\mathbf{Q} = \begin{bmatrix} 1 & 0 & 1 \end{bmatrix}$$

$$\mathbf{K} = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix}$$

$$\mathbf{V} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$$

Scores: $\mathbf{Q}\mathbf{K}^T = \begin{bmatrix} 1 & 1 & 2 \end{bmatrix}$

Scaled: $\frac{1}{\sqrt{3}} \begin{bmatrix} 1 & 1 & 2 \end{bmatrix} \approx \begin{bmatrix} 0.577 & 0.577 & 1.155 \end{bmatrix}$

Softmax: $\begin{bmatrix}0.264 & 0.264 & 0.471\end{bmatrix}$

Output: $\begin{bmatrix}0.264 & 0.264 & 0.471\end{bmatrix}\begin{bmatrix}1 & 2\\ 3 & 4\\ 5 & 6\end{bmatrix} = \begin{bmatrix}3.41 & 4.41\end{bmatrix}$

Same result as before, but computed via matrix operations.

8.6 Where do Q, K, V come from?

In the examples above, we assumed queries, keys, and values were given. In a transformer, they're computed from the input using learned linear projections.

Given an input sequence $\mathbf{X} \in \mathbb{R}^{n \times d_{\text{model}}}$ where each of the n rows is a token embedding of dimension d_{model}, we compute:

$$\mathbf{Q} = \mathbf{X}\mathbf{W}^Q, \quad \mathbf{K} = \mathbf{X}\mathbf{W}^K, \quad \mathbf{V} = \mathbf{X}\mathbf{W}^V$$

Here $\mathbf{W}^Q \in \mathbb{R}^{d_{\text{model}} \times d_k}$ is the query projection matrix, $\mathbf{W}^K \in \mathbb{R}^{d_{\text{model}} \times d_k}$ is the key projection matrix, and $\mathbf{W}^V \in \mathbb{R}^{d_{\text{model}} \times d_v}$ is the value projection matrix. The dimensions d_k and d_v are hyperparameters; often $d_k = d_v = d_{\text{model}}$, but they can be smaller to reduce computation.

Each token's embedding gets projected into three different spaces. The **query projection** asks "What am I looking for?" and transforms the token into a representation optimized for searching. The **key projection** asks "What do I offer to others?" and transforms the token into a representation optimized for being found. The **value projection** asks "What information do I contribute?" and transforms the token into the content that will be retrieved.

The same token has different projections because it plays different roles depending on whether it's the one asking (query) or being asked about (key/value).

8.6.1 Why separate projections?

Why not use the same vector for query, key, and value? Consider what attention computes: "how relevant is position j to position i?" This relevance might depend on different aspects of the tokens.

For example, in "The cat sat on the mat," when generating output for "sat," we might want:

- Query based on "sat" emphasizing: "I'm a verb, I need a subject"
- Key for "cat" emphasizing: "I'm a noun, I can be a subject"
- Value for "cat" providing: its actual semantic content

Separate projections let the model learn these different roles. The query projection can emphasize syntactic features, the key projection can respond to those queries, and the value projection can provide semantic content.

8.7 Attention as information routing

Another way to understand attention: it routes information between positions. Each position can read from any other position, with the amount of information transferred controlled by the attention weights.

The attention weight α_{ij} represents "how much information flows from position j to position i." When α_{ij} is high, position i's output strongly reflects position j's value.

This creates a flexible information flow that depends on the content of the sequence. Unlike RNNs where information flows strictly sequentially, attention allows direct long-range connections. A token at position 1 can directly influence position 100 if the attention weights say it's relevant.

The computational complexity is $O(n^2 \cdot d)$ because we compute n^2 attention scores (every query with every key) and then n weighted combinations of d-dimensional values. For very long sequences, this quadratic cost becomes expensive, which is why various "efficient attention" variants exist (though we won't cover them in detail).

8.8 Attention visualized

Attention weights form an $n \times n$ matrix (for self-attention, which we'll cover next chapter). Visualizing this matrix reveals patterns that emerge from training and reflect the linguistic structure the model has learned.

Diagonal patterns appear when tokens attend to themselves or nearby tokens, capturing local context and sequential relationships. **Vertical stripes** occur when many tokens attend to a specific position, often the start of a sentence or a semantically important word like a verb. **Block patterns** emerge when groups of tokens attend to other groups, reflecting phrase structure or clause boundaries. **Sparse patterns** are common in trained models: most weights are near zero, with a few dominant connections, suggesting the model learns to focus on a small number of relevant positions rather than spreading attention uniformly.

8.9 Properties of attention

Attention has several important properties that make it well-suited for sequence modeling.

First, attention is **permutation equivariant**. This means if you shuffle the input sequence, the output gets shuffled in exactly the same way. Consider a sentence "cat sat mat" with tokens at positions 1, 2, 3. If we shuffle it to "mat cat sat" (positions 3, 1, 2), attention produces the same outputs, just reordered to match. The output for "cat" is identical whether "cat" was at position 1 or position 2.

Why? Because attention only compares content (queries against keys), not positions. The attention weight between "cat" and "sat" depends on their embeddings, not on whether "cat" is first or second. Attention treats the input as a *set* of vectors, not an ordered sequence.

This is a problem for language, where order matters ("dog bites man" vs "man bites dog"). The solution is to add position information to the embeddings before attention sees them, via positional encodings (covered later). Without positional encodings, attention would process "the cat sat" and "sat the cat" identically.

Second, attention is **differentiable**. Every operation (the dot products, the scaling, the softmax, the weighted sum) is a smooth function of its inputs. Gradients flow through attention without discontinuities, allowing end-to-end training via backpropagation.

Third, attention is **parallelizable**. Unlike RNNs where each step depends on the previous step, all positions in attention can be processed simultaneously. The matrix multiplication $\mathbf{QK}^T$ computes all pairwise scores at once, and modern GPUs are optimized for exactly this kind of parallel matrix operation.

Fourth, attention is **dynamic**: the attention pattern changes based on the input content. The same trained model will attend to different positions for different inputs, because the attention weights are computed from the queries and keys, not fixed during training. This content-based routing is what makes attention so powerful.

8.10 Attention vs. full connection

You might wonder: why not just use a fully connected layer instead of attention? A fully connected layer also lets every position influence every other position.

The difference is in how the weights are determined. In a **fully connected layer**, the weights are fixed parameters learned during training. The connection from position j to position i has the same weight regardless of what's at those positions. The weights are static, determined only by the positions.

In **attention**, the weights are computed from the content. The connection strength depends on how well the query at i matches the key at j. Different inputs produce different attention patterns, even with the same trained weights.

This makes attention a form of **content-based routing**: the same architecture can route information differently for different inputs. This is crucial for handling variable-length sequences and capturing context-dependent relationships. A fully connected layer can't generalize to sequences of different lengths, but attention can.

8.11 Summary

We've developed the attention mechanism:

- Attention computes a weighted average of values, where weights depend on query-key similarity.
- The scaled dot product $\frac{\mathbf{q}^T\mathbf{k}}{\sqrt{d}}$ measures how well a query matches a key.
- Softmax turns scores into a probability distribution (attention weights).
- In matrix form: $\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d}}\right)\mathbf{V}$
- Queries, keys, and values are computed from input via learned linear projections.
- Attention enables content-based information routing between all positions.

In the next chapter, we'll see how attention is applied when queries, keys, and values all come from the same sequence: **self-attention**.

Chapter 9

Self-attention

Learning objectives

After completing this chapter, you will be able to:

- Distinguish self-attention from cross-attention
- Compute self-attention where queries, keys, and values come from the same sequence
- Trace through a complete self-attention computation with concrete numbers
- Explain why nonlinearity is essential in neural networks
- Describe the role of the feed-forward network in transformer blocks

In the previous chapter, we developed attention as a mechanism for combining information based on relevance. The queries asked about keys, and the answers came from values. But where do these come from? In **self-attention**, the queries, keys, and values all come from the same sequence. Each position attends to every other position (including itself) in the same sequence. This is the fundamental operation at the heart of transformers.

9.1 From attention to self-attention

In general attention, queries, keys, and values can come from different sequences. Consider machine translation from English to French. The encoder processes the English sentence "The cat sat" and produces a sequence of hidden states. The decoder generates French tokens one at a time: "Le", "chat", … When generating "chat", the decoder needs to look back at the English sentence to find the relevant word ("cat"). So the decoder's current state becomes the query, while the encoder's hidden states provide the keys and values. The decoder is asking: "Which English words should I pay attention to right now?" This is called **cross-attention**: the query comes from one sequence (French, being generated) and the keys/values come from another (English, already encoded).

In **self-attention**, a single sequence provides all three. Queries, keys, and values all come from the same sequence. Each token attends to every other token in that same sequence, building a representation that incorporates information from the entire context. There's no second sequence involved.

Why is this useful? Self-attention lets each position gather information from all other positions based on relevance. A verb can find its subject, a pronoun can find its antecedent, and a word can incorporate context from anywhere in the sequence, all in a single operation.

9.2 The self-attention computation

Given an input sequence $\mathbf{X} \in \mathbb{R}^{n \times d}$ where n is the sequence length and d is the embedding dimension (each row is one token's embedding), self-attention proceeds in four steps.

Step 1: Project to queries, keys, values. We transform the input into three different representations:

$$\mathbf{Q} = \mathbf{X}\mathbf{W}^Q, \quad \mathbf{K} = \mathbf{X}\mathbf{W}^K, \quad \mathbf{V} = \mathbf{X}\mathbf{W}^V$$

Here $\mathbf{W}^Q, \mathbf{W}^K, \mathbf{W}^V \in \mathbb{R}^{d \times d_k}$ are learned weight matrices that project the d-dimensional input into d_k-dimensional query, key, and value spaces. Typically $d_k = d$ or $d_k = d/h$ where h is the number of attention heads (covered in the next chapter). The key point is that the same input $\mathbf{X}$ is used for all three projections. This is what makes it *self*-attention.

Step 2: Compute attention scores. We measure how relevant each position is to each other position:

$$\mathbf{S} = \frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}$$

The score matrix $\mathbf{S} \in \mathbb{R}^{n \times n}$ contains all pairwise scores. Entry s_{ij} is the scaled dot product between position i's query and position j's key, answering: how relevant is position j to position i?

Step 3: Apply softmax. We convert scores to probabilities:

$$\mathbf{A} = \text{softmax}(\mathbf{S})$$

Softmax is applied row-wise. Row i of the attention matrix $\mathbf{A}$ contains the attention weights, a probability distribution over all positions representing how much position i attends to each position.

Step 4: Compute output. We gather information according to the attention weights:

$$\mathbf{O} = \mathbf{A}\mathbf{V}$$

Each row of the output $\mathbf{O} \in \mathbb{R}^{n \times d_k}$ is a weighted combination of value vectors. Position i's output is the sum of all value vectors, weighted by how much position i attends to each position.

Combining all steps, the complete self-attention operation is:

$$\text{SelfAttention}(\mathbf{X}) = \text{softmax}\left(\frac{\mathbf{X}\mathbf{W}^Q(\mathbf{X}\mathbf{W}^K)^T}{\sqrt{d_k}}\right)\mathbf{X}\mathbf{W}^V$$

9.3 Concrete example

Let's work through self-attention on a tiny sequence. Suppose we have 3 tokens with 4-dimensional embeddings:

$$\mathbf{X} = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix}$$

Row 1 is token 1's embedding, row 2 is token 2's, row 3 is token 3's.

For simplicity, let's use $d_k = 2$ and assume the projection matrices are:

$$\mathbf{W}^Q = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad \mathbf{W}^K = \begin{bmatrix} 0 & 1 \\ 1 & 0 \\ 0 & 1 \\ 1 & 0 \end{bmatrix}, \quad \mathbf{W}^V = \begin{bmatrix} 1 & 1 \\ 0 & 0 \\ 0 & 1 \\ 1 & 0 \end{bmatrix}$$

Step 1: Compute Q, K, V

$$\mathbf{Q} = \mathbf{X}\mathbf{W}^Q = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ 0 & 2 \\ 1 & 1 \end{bmatrix}$$

$$\mathbf{K} = \mathbf{X}\mathbf{W}^K = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \\ 0 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 2 \\ 2 & 0 \\ 1 & 1 \end{bmatrix}$$

$$\mathbf{V} = \mathbf{X}\mathbf{W}^V = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 0 & 0 \\ 0 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 2 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}$$

Step 2: Compute scores

$$\mathbf{Q}\mathbf{K}^T = \begin{bmatrix} 2 & 0 \\ 0 & 2 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 0 & 2 & 1 \\ 2 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 4 & 2 \\ 4 & 0 & 2 \\ 2 & 2 & 2 \end{bmatrix}$$

Scaling by $\sqrt{d_k} = \sqrt{2} \approx 1.414$:

$$\mathbf{S} = \frac{1}{1.414} \begin{bmatrix} 0 & 4 & 2 \\ 4 & 0 & 2 \\ 2 & 2 & 2 \end{bmatrix} \approx \begin{bmatrix} 0 & 2.83 & 1.41 \\ 2.83 & 0 & 1.41 \\ 1.41 & 1.41 & 1.41 \end{bmatrix}$$

Step 3: Softmax

For row 1: $\text{softmax}([0, 2.83, 1.41]) = [\frac{1}{1+16.95+4.10}, \frac{16.95}{22.05}, \frac{4.10}{22.05}] \approx [0.045, 0.769, 0.186]$

For row 2: $\text{softmax}([2.83, 0, 1.41]) \approx [0.769, 0.045, 0.186]$

For row 3: $\text{softmax}([1.41, 1.41, 1.41]) = [0.333, 0.333, 0.333]$

$$\mathbf{A} \approx \begin{bmatrix} 0.045 & 0.769 & 0.186 \\ 0.769 & 0.045 & 0.186 \\ 0.333 & 0.333 & 0.333 \end{bmatrix}$$

Step 4: Compute output

$$\mathbf{O} = \mathbf{A}\mathbf{V} = \begin{bmatrix} 0.045 & 0.769 & 0.186 \\ 0.769 & 0.045 & 0.186 \\ 0.333 & 0.333 & 0.333 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}$$

For row 1: $0.045 \cdot [1, 2] + 0.769 \cdot [1, 0] + 0.186 \cdot [1, 1] = [1.0, 0.28]$

For row 2: $0.769 \cdot [1, 2] + 0.045 \cdot [1, 0] + 0.186 \cdot [1, 1] = [1.0, 1.72]$

For row 3: $0.333 \cdot [1, 2] + 0.333 \cdot [1, 0] + 0.333 \cdot [1, 1] = [1.0, 1.0]$

$$\mathbf{O} \approx \begin{bmatrix} 1.0 & 0.28 \\ 1.0 & 1.72 \\ 1.0 & 1.0 \end{bmatrix}$$

9.3.1 Interpreting the attention pattern

Look at the attention matrix $\mathbf{A}$:

- Token 1 mostly attends to token 2 (weight 0.769)
- Token 2 mostly attends to token 1 (weight 0.769)
- Token 3 attends equally to all tokens (weights 0.333 each)

These patterns emerged from the dot products between queries and keys. Tokens 1 and 2 have queries and keys that align strongly with each other but not with themselves (the diagonal scores were 0 and 0). Token 3's query aligns equally with all keys.

The output for each token is a blend of all value vectors, weighted by these attention patterns. Token 1's output is dominated by token 2's value. Token 3's output is the average of all values.

9.4 What self-attention learns

Self-attention learns to compute useful relationships between tokens. The projection matrices $\mathbf{W}^Q$, $\mathbf{W}^K$, $\mathbf{W}^V$ determine what aspects of tokens are compared and what information is gathered. Through training, these matrices evolve to support patterns the model needs for its task.

What kinds of patterns emerge? Self-attention can learn **syntactic relationships**: a verb might learn to attend to its subject and object, so in "The cat ate the fish," the query for "ate" strongly matches the keys for "cat" and "fish." It can learn **coreference**: pronouns attend to their antecedents, so in "John said he was tired," "he" attends strongly to "John."

Unlike RNNs, self-attention naturally handles **long-range dependencies**. In "The cat that I saw yesterday at the park was black," the word "was" can directly attend to "cat" without information passing through all the intervening tokens. The path length between any two positions is just one attention step, not proportional to their distance.

Self-attention also supports **copying and retrieval**: when the model needs to copy information from one position to another, it simply puts high attention weight on the source position. The output then strongly reflects the source's value. This is useful for tasks like question answering, where the answer often appears verbatim in the context.

9.5 The attention matrix

The attention matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ is central to understanding self-attention. Each entry a_{ij} represents how much position i attends to position j. Let's look at a concrete example for the sentence "The cat sat":

	The	cat	sat
The	0.70	0.15	0.15
cat	0.10	0.60	0.30
sat	0.05	0.75	0.20

Reading this table: each row shows how one token distributes its attention. The row for "sat" shows that when computing the new representation for "sat", the model pulls 5% from "The", 75% from "cat", and 20% from itself. This makes sense: "sat" is a verb looking for its subject, and "cat" is the subject.

Notice the asymmetry: "sat" attends strongly to "cat" (0.75), but "cat" attends only moderately to "sat" (0.30). The attention relationship is not mutual.

Let's examine the general properties of attention matrices.

Each row sums to 1 because softmax produces a probability distribution. Position i distributes its attention across all positions, and the total weight must be 1. All entries are non-negative (softmax outputs are always positive), so attention weights can be interpreted as probabilities.

The matrix is generally **not symmetric**: $a_{ij} \neq a_{ji}$. How much position i attends to position j can differ from how much j attends to i. In "The cat sat," the verb "sat" might strongly attend to "cat" (looking for its subject), but "cat" might not strongly attend to "sat" (nouns don't typically search for their verbs).

The **diagonal entries** are often significant. Tokens can attend to themselves, and often do. A token's own value is frequently relevant to its output. However, the diagonal is learned, not special; if self-attention isn't useful, the model can learn to ignore it.

Trained models typically develop **sparse attention patterns** where most weights are near zero and a few dominate. This suggests the model learns to focus on a small number of relevant positions rather than spreading attention uniformly. Different attention heads (covered next chapter) often specialize in different patterns.

9.6 Computational complexity

Computational complexity measures how the number of operations grows as the input size increases. We express this using big-O notation: $O(n)$ means operations grow linearly with input size, $O(n^2)$ means they grow quadratically, and so on. This matters because it determines whether an algorithm is practical for large inputs. An $O(n)$ algorithm that processes a sequence of 1,000 tokens can likely handle 10,000 tokens (10× more work). An $O(n^2)$ algorithm would require 100× more work.

Self-attention has complexity $O(n^2 \cdot d)$, where n is the sequence length and d is the dimension.

The dominant cost comes from the matrix multiplications. Computing $\mathbf{QK}^T$ multiplies an $n \times d$ matrix by a $d \times n$ matrix, producing an $n \times n$ score matrix. This takes $O(n^2 \cdot d)$ operations. The softmax takes $O(n^2)$ to process all entries. Computing $\mathbf{AV}$ multiplies the $n \times n$ attention matrix by the $n \times d$ value matrix, another $O(n^2 \cdot d)$ operations.

The n^2 term is significant. For sequence length $n = 1{,}000$, we compute and store 1 million attention scores. For $n = 10{,}000$, it's 100 million. This quadratic scaling limits how long sequences transformers can efficiently process. It's also why early GPT models had context windows of only 1,024 or 2,048 tokens.

Various "efficient attention" methods reduce this complexity through approximations (linear attention), sparsity patterns (sparse transformers), or reformulations (FlashAttention). But the basic transformer uses full quadratic attention, and understanding it is essential before exploring optimizations.

9.7 Self-attention vs. recurrence

How does self-attention compare to RNNs for sequence modeling? The table below summarizes the key differences:

Aspect	Self-Attention	RNN
Complexity per layer	$O(n^2 \cdot d)$	$O(n \cdot d^2)$
Sequential operations	$O(1)$	$O(n)$

Aspect	Self-Attention	RNN
Maximum path length	$O(1)$	$O(n)$
Parallelizable	Yes	No

The **path length** measures how many steps information must travel between two positions. In an RNN, information from position 1 must pass through all intermediate hidden states to reach position n, a path of length $n-1$. Each step applies a transformation, and information can be lost or distorted along the way. In self-attention, any position can directly attend to any other in one step, regardless of distance.

Short paths have two benefits. First, gradients flow more easily during training. In an RNN, gradients must backpropagate through n steps, risking vanishing or exploding. In self-attention, gradients flow directly between any two positions. Second, information is less likely to be lost. The "telephone game" effect, where information degrades as it passes through many transformations, is avoided.

Parallelization is the other huge advantage. RNNs must compute sequentially because $\mathbf{h}_t$ depends on $\mathbf{h}_{t-1}$. Self-attention computes all positions simultaneously. On modern GPUs that excel at parallel matrix operations, this makes self-attention dramatically faster to train. A sequence of length 1,000 requires 1,000 sequential steps in an RNN but just one parallel operation in self-attention.

9.8 Adding nonlinearity

Self-attention computes weighted averages of value vectors. This is fundamentally a linear operation: if you double all the inputs, the outputs double. While softmax adds nonlinearity in computing the attention weights, the final combination $\sum_j a_{ij}\mathbf{v}_j$ is just a weighted sum. Why is this a problem?

9.8.1 The limitation of linear functions

A function f is **linear** if it satisfies two properties: $f(a\mathbf{x}+b\mathbf{y}) = af(\mathbf{x})+bf(\mathbf{y})$ for any scalars a, b and vectors $\mathbf{x}, \mathbf{y}$. Matrix multiplication is linear: $\mathbf{W}(a\mathbf{x}+b\mathbf{y}) = a\mathbf{W}\mathbf{x}+b\mathbf{W}\mathbf{y}$.

The critical limitation: **composing linear functions gives another linear function**. If f and g are linear, then $f(g(\mathbf{x}))$ is also linear. Mathematically, if $f(\mathbf{x}) = \mathbf{W}_2\mathbf{x}$ and $g(\mathbf{x}) = \mathbf{W}_1\mathbf{x}$, then:

$$f(g(\mathbf{x})) = \mathbf{W}_2(\mathbf{W}_1\mathbf{x}) = (\mathbf{W}_2\mathbf{W}_1)\mathbf{x} = \mathbf{W}_3\mathbf{x}$$

where $\mathbf{W}_3 = \mathbf{W}_2\mathbf{W}_1$ is just another matrix. No matter how many linear layers you stack, the result is equivalent to a single linear layer. A 100-layer linear network has the same representational power as a 1-layer linear network.

Linear functions can only learn linear relationships. They can't learn "if this AND that" or "if this OR that" or "if this is greater than a threshold." Consider a classic example: a hallway light controlled by two switches (one at each end). The light turns on when exactly one switch is flipped.

The four corners represent all switch combinations. When both switches are off (0,0) or both are on (1,1), the light is off (black circles). When exactly one switch is on (0,1) or (1,0), the light is on (white circles). This creates a diagonal pattern.

On the left, we try a straight line (linear function) but it can't separate "light on" from "light off". The dashed line fails to keep the black and white points on opposite sides. On the right, a curved boundary (nonlinear function) succeeds. The curve wraps around the two white "on" points, separating them from the black "off" corners. Linear functions can't draw curves, so they can't solve this problem. To solve problems like this, we need nonlinearity.

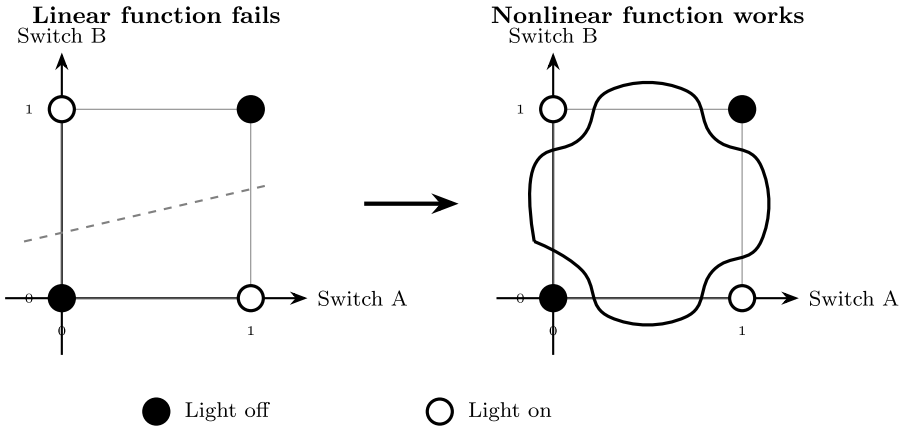

Figure 9.1: Left: Linear function fails to separate the two classes. Right: A nonlinear (curved) boundary successfully separates them

9.8.2 What is ReLU?

The **Rectified Linear Unit (ReLU)** is the most common nonlinearity in modern neural networks. It's defined as:

$$\text{ReLU}(x) = \max(0, x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$$

For a vector $\mathbf{x} = [x_1, x_2, \dots, x_d]$, we apply ReLU element-wise:

$$\text{ReLU}(\mathbf{x}) = [\max(0, x_1), \max(0, x_2), \dots, \max(0, x_d)]$$

ReLU is simple to compute: if the input is positive, pass it through unchanged. If negative, replace it with zero.

Example computation:

$$\text{ReLU}([2, -1, 0, -3, 5]) = [2, 0, 0, 0, 5]$$

9.8.3 How ReLU introduces nonlinearity

Let's understand why ReLU is nonlinear and how this simple function enables complex learning. Consider a linear function $f(x) = 2x$:

This satisfies the linearity property: $f(2x) = 2f(x)$, and $f(x + y) = f(x) + f(y)$. No matter what inputs you give it, the output is always proportional.

Now consider ReLU applied to the same input: $g(x) = \text{ReLU}(2x)$:

The key difference: for $x < 0$, the output is 0 instead of negative. This **breaks linearity**. We can verify: $g(-1) = 0$ and $g(1) = 2$, so $g(-1) + g(1) = 2$. But $g(-1 + 1) = g(0) = 0 \neq 2$. The linearity property is violated.

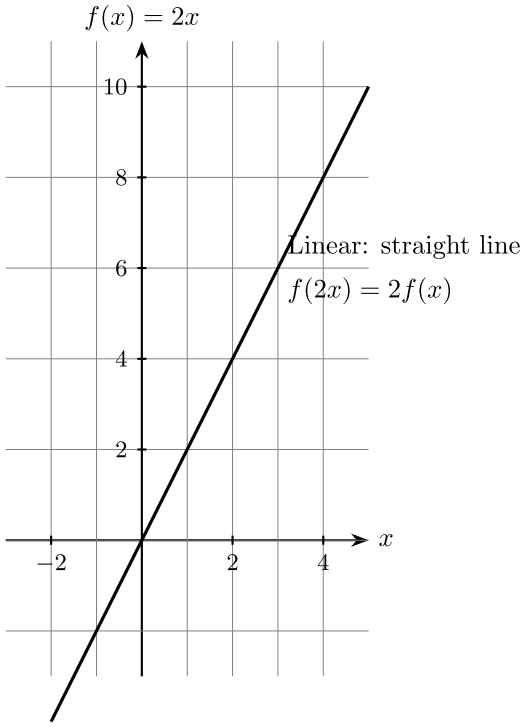

Figure 9.2: Linear function f(x) = 2x showing a straight line passing through the origin

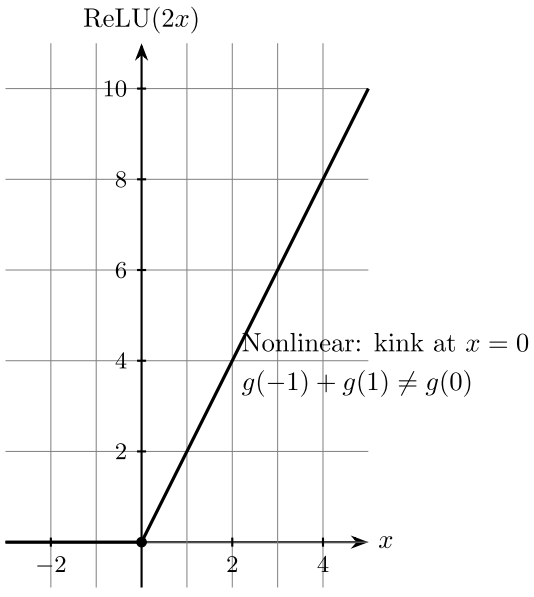

Figure 9.3: ReLU function showing nonlinearity with a bend at x=0, staying flat for negative x

9.8.4 Why the bend matters

The bend at $x = 0$ divides the input space into two regions with different behavior:

- **Region 1** ($x < 0$): Output is always 0, no matter how negative x gets. The function is "turned off."
- **Region 2** ($x > 0$): Output equals input. The function is "turned on" and passes values through.

This creates a **threshold behavior**: below zero, nothing happens; above zero, the function activates. This is fundamentally different from a line, which has the same behavior everywhere.

9.8.5 How multiple ReLUs create complex boundaries

A single ReLU creates one bend, one threshold. But what happens when you combine many ReLUs? You get many bends, many thresholds, and together they can approximate any shape.

Consider a 2D input $\mathbf{x} = [x_1, x_2]$ passing through a layer with multiple ReLU units. Each unit computes:

$$h_i = \text{ReLU}(\mathbf{w}_i^T \mathbf{x} + b_i)$$

where $\mathbf{w}_i$ is a weight vector and b_i is a bias. Geometrically, $\mathbf{w}_i^T \mathbf{x} + b_i = 0$ defines a line in 2D space. The ReLU turns this into a decision: on one side of the line, $h_i = 0$ (off); on the other side, h_i is positive (on).

With n ReLU units, you have n different lines dividing the space into regions. Each region has a different pattern of which units are on vs. off. A subsequent layer can combine these regions to create complex decision boundaries.

Example: To solve the two-switch problem (XOR), a simple network might work like this:

1. **First ReLU unit:** Detects "Switch A is on" (region where $x_1 > 0.5$)
2. **Second ReLU unit:** Detects "Switch B is on" (region where $x_2 > 0.5$)
3. **Third ReLU unit:** Detects "both switches are on" (region where $x_1 + x_2 > 1.5$)
4. **Output layer:** Combine these: "light on" = (Switch A on) + (Switch B on) - 2×(both on)

This gives: - (0,0): No units fire, output = 0 (off) - (0,1): Unit 2 fires, output = 1 (on) - (1,0): Unit 1 fires, output = 1 (on) - (1,1): Units 1, 2, 3 fire, output = 1 + 1 - 2 = 0 (off)

Each ReLU creates a region, and combining regions lets us draw the curved boundary we saw earlier.

9.8.6 Piecewise linear approximation

From another angle, functions with ReLU are **piecewise linear**: they're made of straight line segments joined at bends. Between bends, the function is linear. But the bends let the overall shape be nonlinear.

Think of approximating a smooth curve with straight line segments. With one segment, you can't do much. With two segments (one bend), you can make a corner. With ten segments, you can approximate a gentle curve. With hundreds of segments, you can approximate almost any smooth function.

A deep network with ReLU is doing exactly this in high-dimensional space: creating many regions (separated by hyperplanes at the bends) and assigning each region a different linear function. The result looks smooth and nonlinear when you zoom out, even though it's technically piecewise linear.

9.8.7 Why ReLU is simple but powerful

ReLU's power comes from being just barely nonlinear. It's the simplest possible nonlinearity: - **Computation:** Just `max(0, x)`, no expensive operations like exponentials - **Gradient:** Either 0 or 1, very simple to compute during backpropagation - **Unbounded above:** Unlike sigmoid or tanh, ReLU doesn't saturate for large positive values, avoiding vanishing gradients

Yet this simple "clip negatives to zero" operation is enough. With enough ReLU units and enough layers, the network can learn to approximate any continuous function. The key insight: you don't need fancy nonlinearities. You just need *some* nonlinearity, and ReLU's threshold behavior is sufficient.

9.8.8 Why project to higher dimensions?

The feedforward network in transformers doesn't just apply ReLU. It projects to a higher dimension first:

$$\text{FFN}(\mathbf{x}) = \mathbf{W}_2 \cdot \text{ReLU}(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2$$

where $\mathbf{W}_1 \in \mathbb{R}^{d \times d_{ff}}$ projects from dimension d to a larger dimension d_{ff} (typically $d_{ff} = 4d$), and $\mathbf{W}_2 \in \mathbb{R}^{d_{ff} \times d}$ projects back down.

Why go up and then back down? The intuition is geometric: **higher dimensions give you more room to work**.

Consider a factory quality control system inspecting manufactured parts on a conveyor belt. You measure two features for each part: size (in millimeters) and weight (in grams). Some parts are good, some are defective. The defective parts happen to fall along a diagonal pattern in 2D space, while good parts are at the corners.

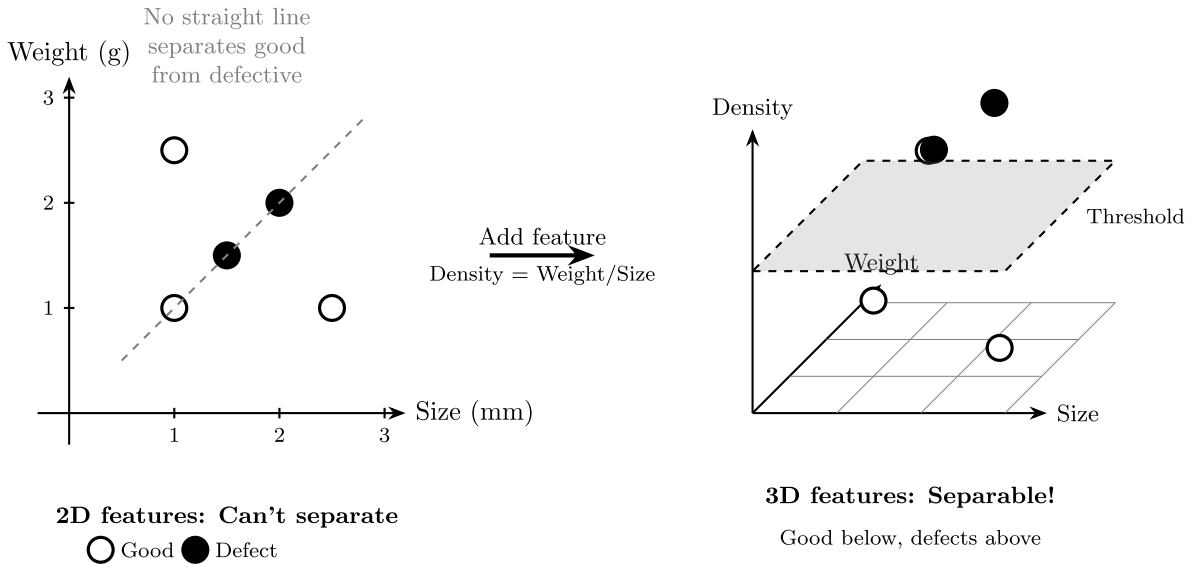

Figure 9.4: Left: With only size and weight, no straight line separates good parts from defects. Right: Adding a derived feature (density) makes separation possible with a simple threshold

In 2D (left panel), using just size and weight, the defective parts (black) are mixed with good parts (white) in a way that no straight line can separate them. The defects fall along the diagonal where size and weight are proportional, while good parts are scattered.

But what if we add a third feature: **density = weight / size**? Now we're in 3D (right panel). The good parts have low or high density (they're either light for their size or heavy for their size), while defective parts have medium density (they fall in the "wrong" proportional range). Now a simple horizontal plane separates them: good parts below the threshold, defects above.

The problem became linearly separable by adding a dimension. We didn't change the data, we just looked at it from a higher-dimensional perspective that revealed the underlying pattern.

This is what the FFN does: project to higher dimensions (via $\mathbf{W}_1$), apply nonlinearity (ReLU creates thresholds), then project back (via $\mathbf{W}_2$). The higher-dimensional space gives more "degrees of freedom" to represent complex patterns.

Think of it like this: in 2D, you can only draw straight lines. In 3D, you can draw planes. In 4D, you can draw 3D volumes. In 2048D (a typical d_{ff} for transformers), you have enormous flexibility to separate and organize data. The network learns to use these extra dimensions to compute features that aren't easily expressible in the original space, then combines them back into a useful representation.

9.8.9 Why projecting back down doesn't lose what we learned

But wait - if we can separate things in high dimensions, why doesn't projecting back down collapse everything and lose the separation? This seems paradoxical.

The key insight: **we're not projecting back to the original space**. We're creating a *new* lower-dimensional representation that encodes what we learned in the high-dimensional space.

Let's continue the factory example. After computing density in 3D and separating good parts from defects, the projection back doesn't just throw away the density information. Instead, it might create a new 2D output like:

$$\text{quality score} = 0.3 \cdot \text{size} + 0.2 \cdot \text{weight} + 0.8 \cdot \text{ReLU}(\text{density} - 1.5) \tag{9.1}$$

$$\text{defect probability} = -0.1 \cdot \text{size} - 0.1 \cdot \text{weight} + 0.9 \cdot \text{ReLU}(\text{density} - 1.5) \tag{9.2}$$

Notice what happened: the high-dimensional feature (density) got *compressed* into the output, but the **decision based on density** is preserved. The ReLU created a threshold at density = 1.5. After ReLU: - Good parts (low density): ReLU output = 0, so they get low defect probability - Defects (high density): ReLU output = positive, so they get high defect probability

The projection back ($\mathbf{W}_2$) is learning **how to combine** the thresholded features. It's not reversing the projection up - it's a completely different linear transformation that says "here's how to weight these high-dimensional insights when summarizing back to lower dimensions."

9.8.10 The projection cycle creates new features

Think of the process as:

1. **Project up** ($\mathbf{W}_1$): "Let me look at the input from many angles." Each dimension in the high-dimensional space is asking a different question about the input. One dimension might compute "is size proportional to weight?", another "is weight greater than 2?", another "is size times weight large?", etc.
2. **Apply ReLU**: "Decide which angles matter right now." For this particular input, some questions have positive answers (kept), others negative (zeroed out). This is selecting which features are relevant.
3. **Project down** ($\mathbf{W}_2$): "Combine the answers that survived into a useful summary." The output isn't the original input - it's a summary of which high-dimensional features were active.

Here's a concrete toy example. Suppose $\mathbf{x} = [3, 4]$ (size=3, weight=4), and we project to 3D with:

$$\mathbf{h} = \text{ReLU}(\mathbf{W}_1\mathbf{x}) = \text{ReLU}\left(\begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0.5 & -0.5 \end{bmatrix} \begin{bmatrix} 3 \\ 4 \end{bmatrix}\right) = \text{ReLU}\left(\begin{bmatrix} 3 \\ 4 \\ -0.5 \end{bmatrix}\right) = \begin{bmatrix} 3 \\ 4 \\ 0 \end{bmatrix}$$

The third dimension computed $0.5 \times 3 - 0.5 \times 4 = -0.5$ (checking if size $>$ weight), which is negative, so ReLU killed it. Now project back:

$$\mathbf{y} = \mathbf{W}_2\mathbf{h} = \begin{bmatrix} 1 & 0 & 0.5 \\ 0 & 1 & -0.5 \end{bmatrix} \begin{bmatrix} 3 \\ 4 \\ 0 \end{bmatrix} = \begin{bmatrix} 3 \\ 4 \end{bmatrix}$$

In this case, the output is the same as input because the third feature wasn't active. But if we had $\mathbf{x} = [4, 2]$ (size > weight):

$$\mathbf{h} = \text{ReLU}\left(\begin{bmatrix} 4 \\ 2 \\ 1 \end{bmatrix} \right) = \begin{bmatrix} 4 \\ 2 \\ 1 \end{bmatrix}$$

Now the third feature is active (size > weight). Projecting back:

$$\mathbf{y} = \begin{bmatrix} 1 & 0 & 0.5 \\ 0 & 1 & -0.5 \end{bmatrix} \begin{bmatrix} 4 \\ 2 \\ 1 \end{bmatrix} = \begin{bmatrix} 4.5 \\ 1.5 \end{bmatrix}$$

The output is **different** from the input! The network has encoded "this part has size > weight" by boosting the first dimension and reducing the second. The output is a *new representation* that encodes what the network learned by going to higher dimensions.

9.8.11 Why this is powerful

The FFN is computing **derived features** in high dimensions and **encoding their activation patterns** in the output. With 2048 dimensions (typical d_{ff}), you can compute 2048 different features, threshold them with ReLU, and then the output encodes "feature 17 was active, feature 203 was active, feature 1842 was inactive, …" in a compressed form.

The network learns through training: - **Which features to compute** (what $\mathbf{W}_1$ does) - **Which features matter for the task** (what survives ReLU) - **How to combine active features into outputs** (what $\mathbf{W}_2$ does)

So projecting back down doesn't lose information - it compresses the high-dimensional decisions into a lower-dimensional representation that's optimized for the task. It's like saying "I examined 2048 different properties of the input, and here's a summary of what I found."

9.8.12 The complete feedforward network

Putting it together, the position-wise feedforward network is:

$$\text{FFN}(\mathbf{x}) = \mathbf{W}_2 \cdot \text{ReLU}(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2$$

where: - $\mathbf{x} \in \mathbb{R}^d$ is the input (from self-attention) - $\mathbf{W}_1 \in \mathbb{R}^{d \times d_{ff}}$ projects to dimension d_{ff} (typically $4d$) - $\mathbf{b}_1 \in \mathbb{R}^{d_{ff}}$ is a bias (shifts the activation) - ReLU zeros out negative values - $\mathbf{W}_2 \in \mathbb{R}^{d_{ff} \times d}$ projects back to dimension d - $\mathbf{b}_2 \in \mathbb{R}^d$ is the final bias

This is applied independently to each position. It doesn't mix information across positions (self-attention does that). Instead, it transforms each position's representation in a complex, nonlinear way.

The combination of self-attention (mixing information across positions) and FFN (nonlinear transformation at each position) gives transformers their power. Self-attention handles "communication": which positions should influence each other. FFN handles "computation": what complex features should be computed from the gathered information.

9.8.13 What the output space looks like

Let's visualize exactly what happens to the coordinates. The diagram below shows how the FFN transforms a 2D input space into a different 2D output space:

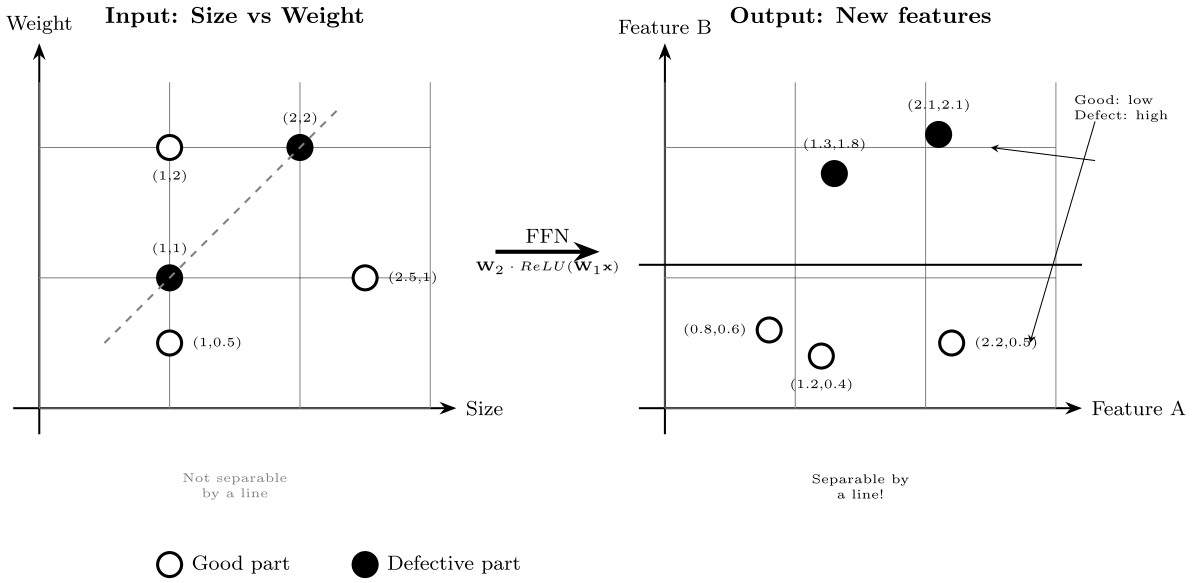

Figure 9.5: The FFN transforms input coordinates (size, weight) into new output coordinates (Feature A, Feature B) where the problem becomes linearly separable

Left panel (Input space): The x-axis is "Size" and y-axis is "Weight", the original measurements. The five points are mixed: defective parts (black) fall on a diagonal, good parts (white) are scattered. No straight line can separate them.

Right panel (Output space): After passing through the FFN, we're in a completely different 2D space. The x-axis is now "Feature A" and y-axis is "Feature B". These aren't size or weight anymore; they're new derived features computed by the network. Look at what happened to the points:

- **Good parts moved to the bottom** (low Feature B values: 0.4, 0.5, 0.6)
- **Defective parts moved to the top** (high Feature B values: 1.8, 2.1)
- A horizontal line at Feature B = 1.1 now cleanly separates them!

The coordinates literally changed. The input point at (1, 1) moved to approximately (1.3, 1.8) in the output space. The input point at (2.5, 1) moved to approximately (2.2, 0.5). The FFN didn't just relabel the axes. It computed entirely new coordinates where separation is possible.

What are Feature A and Feature B mathematically? Let's trace through the computation with a concrete example. Suppose we project to 3 dimensions (instead of 2048) with:

$$\mathbf{W}_1 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ -0.5 & 0.5 \end{bmatrix}, \quad \mathbf{b}_1 = \begin{bmatrix} 0 \\ 0 \\ 1.5 \end{bmatrix}$$

For input $\mathbf{x} = [\text{size}, \text{weight}]^T$, the high-dimensional features are:

$$\mathbf{h} = \text{ReLU}\left(\begin{bmatrix} 1 & 0 \\ 0 & 1 \\ -0.5 & 0.5 \end{bmatrix}\begin{bmatrix} \text{size} \\ \text{weight} \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 1.5 \end{bmatrix}\right) = \text{ReLU}\left(\begin{bmatrix} \text{size} \\ \text{weight} \\ -0.5 \cdot \text{size} + 0.5 \cdot \text{weight} + 1.5 \end{bmatrix}\right)$$

So the three high-dimensional features before ReLU are: - $h_1 = \text{size}$ (always positive, passes through ReLU) - $h_2 = \text{weight}$ (always positive, passes through ReLU) - $h_3 = 1.5 + 0.5(\text{weight} - \text{size})$ (measures if weight $>$ size, plus a shift)

After ReLU, h_3 is only positive when weight $>$ size $- 3$. Now project back down:

$$\begin{bmatrix} \text{Feature A} \\ \text{Feature B} \end{bmatrix} = \mathbf{W}_2\mathbf{h} = \begin{bmatrix} 0.3 & 0.2 & 0.1 \\ 0.1 & 0.1 & 0.8 \end{bmatrix}\begin{bmatrix} h_1 \\ h_2 \\ h_3 \end{bmatrix}$$

Therefore: - **Feature A** $= 0.3 \cdot h_1 + 0.2 \cdot h_2 + 0.1 \cdot h_3$ - **Feature B** $= 0.1 \cdot h_1 + 0.1 \cdot h_2 + 0.8 \cdot h_3$

Feature B heavily weights h_3 (the "weight $>$ size" detector). When $h_3 = 0$ (good parts), Feature B is low. When h_3 is large (defects), Feature B is high. That's why the horizontal line at Feature B $= 1.1$ separates them!

So **yes**, Feature A and Feature B are linear combinations of the high-dimensional features h_1, h_2, h_3. But those high-dimensional features themselves are thresholded (ReLU) linear combinations of the inputs. The full formula is:

$$\begin{bmatrix} \text{Feature A} \\ \text{Feature B} \end{bmatrix} = \mathbf{W}_2 \cdot \text{ReLU}(\mathbf{W}_1\begin{bmatrix} \text{size} \\ \text{weight} \end{bmatrix} + \mathbf{b}_1) + \mathbf{b}_2$$

This is **not** a simple linear function of size and weight because of the ReLU. If it were just $\mathbf{W}_2\mathbf{W}_1\mathbf{x}$, that would collapse to a single linear transformation. The ReLU creates the nonlinearity: different regions of input space activate different combinations of the h_i features, leading to different linear mappings in different regions. That's how the FFN creates the curved decision boundaries we need.

This is why going to higher dimensions and back is powerful: the network has room to compute many candidate features (2048 in typical transformers), threshold them with ReLU, and then $\mathbf{W}_2$ learns to combine the useful ones into output coordinates that make the problem easier for the next layer.

9.9 Residual connections and layer normalization

We've built powerful components: self-attention mixes information across positions, and feedforward networks compute complex nonlinear transformations. But when we stack these operations into deep networks, we encounter severe training problems. Two techniques solve these problems: residual connections and layer normalization.

9.9.1 The vanishing gradient problem

To understand why we need residual connections, we need to understand the **vanishing gradient problem**. When training deep networks, we update weights by computing gradients via backpropagation. The gradient tells us how to adjust each weight to reduce the loss.

Consider a simple deep network where each layer multiplies by a weight matrix $\mathbf{W}$:

$$\mathbf{y} = \mathbf{W}_L \cdot \mathbf{W}_{L-1} \cdots \mathbf{W}_2 \cdot \mathbf{W}_1\mathbf{x}$$

To update $\mathbf{W}_1$ (the first layer), we need $\frac{\partial \mathcal{L}}{\partial \mathbf{W}_1}$ where $\mathcal{L}$ is the loss. By the chain rule:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_1} = \frac{\partial \mathcal{L}}{\partial \mathbf{y}} \cdot \frac{\partial \mathbf{y}}{\partial \mathbf{W}_L} \cdots \frac{\partial \mathbf{y}}{\partial \mathbf{W}_2} \cdot \frac{\partial \mathbf{y}}{\partial \mathbf{W}_1}$$

Each derivative depends on the weight matrices. If the weights are initialized with small values (common practice), each derivative term might be less than 1. Multiplying many numbers less than 1 produces exponentially small results.

Concrete example: Suppose each derivative is 0.5, and we have 10 layers. The gradient reaching the first layer is proportional to $0.5^{10} \approx 0.001$. With 20 layers, it becomes $0.5^{20} \approx 0.000001$. The gradient becomes so small it's essentially zero, and the first layer stops learning. This is the **vanishing gradient**.

The opposite problem, **exploding gradients**, occurs when derivatives are larger than 1. Then $1.5^{10} \approx 57$ or $1.5^{20} \approx 3325$, and gradients become enormous. Updates are unstable, weights shoot to infinity, and training collapses.

Both problems worsen with depth. A 5-layer network might train fine, but a 50-layer network becomes impossible without special techniques. This was a major barrier to deep learning until residual connections were invented.

9.9.2 Why residual connections solve this

Residual connections (also called skip connections) add the input directly to the output:

$$\mathbf{X}' = \mathbf{X} + \text{SelfAttention}(\mathbf{X})$$

Instead of $\mathbf{X}' = f(\mathbf{X})$, we compute $\mathbf{X}' = \mathbf{X} + f(\mathbf{X})$. The "$+\mathbf{X}$" creates a direct path from input to output that bypasses the function f.

Why does this help? Look at the gradient. Using the chain rule:

$$\frac{\partial \mathbf{X}'}{\partial \mathbf{X}} = \frac{\partial}{\partial \mathbf{X}}[\mathbf{X} + f(\mathbf{X})] = \mathbf{I} + \frac{\partial f(\mathbf{X})}{\partial \mathbf{X}}$$

The derivative is the identity matrix $\mathbf{I}$ plus something extra. Even if $\frac{\partial f}{\partial \mathbf{X}}$ vanishes (goes to zero), the gradient is still $\mathbf{I}$. The gradient can flow backward through the "$+\mathbf{X}$" path without any multiplicative degradation.

The gradient highway: This direct path is called a "gradient highway" or "shortcut connection." Gradients from later layers can flow directly back to earlier layers without passing through multiple matrix multiplications. Compare:

- **Without residual:** Gradient passes through $\mathbf{W}_L, \mathbf{W}_{L-1}, \ldots, \mathbf{W}_1$, multiplying many matrices.
- **With residual:** Gradient can flow through the shortcut path with no multiplication at all, just addition.

This is revolutionary. We can now train networks with hundreds of layers because gradients reliably reach the early layers. Residual connections (ResNets) enabled the deep learning revolution in computer vision (2015) and transformers in NLP (2017).

9.9.3 How residual connections work mechanically

Let's trace through a concrete example. Suppose we have a 3-token sequence with 4-dimensional embeddings:

$$\mathbf{X} = \begin{bmatrix} 1.0 & 0.5 & -0.5 & 0.2 \\ 0.8 & -0.3 & 0.6 & 0.1 \\ -0.2 & 0.7 & 0.3 & -0.4 \end{bmatrix}$$

After self-attention, we get an output $\mathbf{O}$ (the weighted combination of value vectors):

$$\mathbf{O} = \text{SelfAttention}(\mathbf{X}) = \begin{bmatrix} 0.2 & 0.1 & 0.3 & -0.1 \\ 0.1 & 0.2 & -0.2 & 0.1 \\ 0.3 & -0.1 & 0.1 & 0.2 \end{bmatrix}$$

Without residual connection, the output would just be $\mathbf{O}$. We'd completely replace the input with the self-attention result. If self-attention makes a mistake or produces small values, the original information is lost.

With residual connection, we compute:

$$\mathbf{X}' = \mathbf{X} + \mathbf{O}$$

$$= \begin{bmatrix} 1.0 & 0.5 & -0.5 & 0.2 \\ 0.8 & -0.3 & 0.6 & 0.1 \\ -0.2 & 0.7 & 0.3 & -0.4 \end{bmatrix} + \begin{bmatrix} 0.2 & 0.1 & 0.3 & -0.1 \\ 0.1 & 0.2 & -0.2 & 0.1 \\ 0.3 & -0.1 & 0.1 & 0.2 \end{bmatrix} = \begin{bmatrix} 1.2 & 0.6 & -0.2 & 0.1 \\ 0.9 & -0.1 & 0.4 & 0.2 \\ 0.1 & 0.6 & 0.4 & -0.2 \end{bmatrix}$$

Notice what happened: - The original values (like 1.0, 0.8, -0.2 in the first column) are still present, just modified - Self-attention made small adjustments: +0.2, +0.1, +0.3 in the first row - The output is the **original plus a modification**, not a replacement

This has several benefits:

1. **Preserves information:** The original embedding is always present. Even if self-attention fails completely (outputs zero), we still have $\mathbf{X}' = \mathbf{X}$.
2. **Easier learning:** The network learns to compute **modifications** rather than **new representations from scratch**. It's easier to learn "add 0.2 to this feature" than "compute the right value from scratch."
3. **Graceful initialization:** At initialization, when weights are random, SelfAttention($\mathbf{X}$) might produce garbage. But with residual connections, we get $\mathbf{X}' \approx \mathbf{X}$ + small noise, and the network starts as approximately an identity function. Training gradually learns useful modifications.

9.9.4 Why layer normalization is needed

Even with residual connections, we have another problem: **activation explosion or vanishing**. As signals pass through many layers, the scale of values can drift. Some features might grow to huge values (like 1000), others might shrink to tiny values (like 0.001). This makes training unstable.

Consider what happens after several layers. If each layer adds a small positive value on average, values grow:

- After layer 1: mean value = 1.0
- After layer 10: mean value = 5.0
- After layer 50: mean value = 50.0

Conversely, if values get smaller on average:

- After layer 1: mean value = 1.0
- After layer 10: mean value = 0.1
- After layer 50: mean value = 0.0001

Either scenario is bad. Large values cause gradients to explode (tiny weight changes cause huge output changes). Small values cause gradients to vanish (the signal becomes noise).

Layer normalization solves this by normalizing the scale of features after each layer. It ensures that regardless of what happened in previous layers, the output has a consistent mean and standard deviation.

9.9.5 How layer normalization works

Layer normalization acts on the vector representation of a single token independently. For a given position's vector $\mathbf{x} \in \mathbb{R}^d$ (containing d features), the operation consists of two main steps: **normalization** and **affine transformation**.

The full formula is:

$$\text{LayerNorm}(\mathbf{x}) = \underbrace{\gamma \odot \frac{\mathbf{x} - \mu}{\sigma + \epsilon}}_{\text{Transformation}} + \beta$$

Let's break this down:

1. **Normalization:** The fraction $\frac{\mathbf{x}-\mu}{\sigma+\epsilon}$ standardizes the vector features.
   - **Center ($\mathbf{x} - \mu$):** We subtract the mean μ of the features. This ensures the vector is centered around zero.
   - **Scale (Divide by σ):** We divide by the standard deviation σ. This ensures the values have a spread (variance) of 1.
   - **Stability (ϵ):** We add a tiny number ϵ (e.g., 10^{-5}) to the denominator to prevent division by zero if the variance is 0.

$$\mu = \frac{1}{d}\sum_{i=1}^{d} x_i, \quad \sigma = \sqrt{\frac{1}{d}\sum_{i=1}^{d}(x_i - \mu)^2}$$

2. **Affine Transformation:** Forcing every vector to have exactly mean 0 and variance 1 limits the network's expressiveness. To fix this, we introduce two learnable parameters per feature dimensions:
   - **Scale (γ):** A learned multiplier that can expand or shrink the range.
   - **Shift (β):** A learned bias that can shift the center.

   These parameters ($\gamma, \beta \in \mathbb{R}^d$) allow the model to learn the *optimal* distribution for the features, effectively "undoing" the normalization if necessary, but starting from a stable baseline.

Key intuition: Unlike Batch Normalization (which uses statistics across a batch of samples), Layer Normalization computes μ and σ using only the features within the single vector $\mathbf{x}$ itself. This makes it independent of batch size and perfect for sequence models like Transformers.

The process summary: 1. **Center:** Subtract the mean μ, so the values have mean 0 2. **Scale:** Divide by standard deviation σ, so the values have variance 1 3. **Re-scale and re-shift:** Multiply by learned γ and add learned β

Concrete example: Suppose after self-attention + residual, one position has:

$$\mathbf{x} = [1.2, 0.6, -0.2, 0.1]$$

Step 1: Compute mean and standard deviation:

$$\mu = \frac{1.2 + 0.6 + (-0.2) + 0.1}{4} = \frac{1.7}{4} = 0.425$$

$$\sigma = \sqrt{\frac{(1.2 - 0.425)^2 + (0.6 - 0.425)^2 + (-0.2 - 0.425)^2 + (0.1 - 0.425)^2}{4}}$$

$$= \sqrt{\frac{0.601 + 0.031 + 0.391 + 0.106}{4}} = \sqrt{\frac{1.129}{4}} = \sqrt{0.282} \approx 0.531$$

Step 2: Normalize:

$$\hat{\mathbf{x}} = \frac{\mathbf{x} - \mu}{\sigma} = \frac{[1.2, 0.6, -0.2, 0.1] - 0.425}{0.531} = \frac{[0.775, 0.175, -0.625, -0.325]}{0.531}$$

$$\approx [1.46, 0.33, -1.18, -0.61]$$

Notice: the normalized values have mean 0 and standard deviation 1 (you can verify: $(1.46 + 0.33 - 1.18 - 0.61)/4 = 0$).

Step 3: Apply learned parameters. Suppose $\gamma = [1, 1, 1, 1]$ and $\beta = [0, 0, 0, 0]$ (initialization values). Then:

$$\text{LayerNorm}(\mathbf{x}) = \gamma \odot \hat{\mathbf{x}} + \beta = [1.46, 0.33, -1.18, -0.61]$$

During training, γ and β might learn to be different values (like $\gamma = [2, 1, 0.5, 1]$) if the network finds that helpful. But the normalization ensures values don't explode or vanish.

9.9.6 When and where these techniques are applied

In a transformer layer, residual connections and layer normalization are applied after both self-attention and the feedforward network. Here's the complete flow:

Step 1: Start with input embeddings $\mathbf{X}_0$ (dimension $n \times d$)

Step 2: Self-attention with residual connection:

$$\mathbf{X}_1 = \mathbf{X}_0 + \text{SelfAttention}(\mathbf{X}_0)$$

Step 3: Apply layer normalization:

$$\mathbf{X}_2 = \text{LayerNorm}(\mathbf{X}_1)$$

Step 4: Feedforward network with residual connection:

$$\mathbf{X}_3 = \mathbf{X}_2 + \text{FFN}(\mathbf{X}_2)$$

Step 5: Apply layer normalization:

$$\mathbf{X}_4 = \text{LayerNorm}(\mathbf{X}_3)$$

This sequence is repeated in every transformer layer. The output $\mathbf{X}_4$ from one layer becomes the input $\mathbf{X}_0$ to the next layer.

This arrangement is called **post-norm** because layer normalization comes after the residual addition. Some modern transformers (like GPT-2/GPT-3) use **pre-norm**, where layer normalization is applied before each sublayer:

$$\mathbf{X}_1 = \mathbf{X}_0 + \text{SelfAttention}(\text{LayerNorm}(\mathbf{X}_0))$$

$$\mathbf{X}_2 = \mathbf{X}_1 + \text{FFN}(\text{LayerNorm}(\mathbf{X}_1))$$

Pre-norm is often easier to train for very deep networks (50+ layers) because it normalizes the input to each sublayer, preventing extreme values from entering the computation.

9.9.7 The complete transformer layer equation

Let's integrate everything we've built in this chapter into one complete equation for a single transformer layer. Start with:

- **Input:** $\mathbf{X} \in \mathbb{R}^{n \times d}$ (sequence of n token embeddings, each d-dimensional)

Self-attention sublayer:

$$\mathbf{Q} = \mathbf{X}\mathbf{W}^Q, \quad \mathbf{K} = \mathbf{X}\mathbf{W}^K, \quad \mathbf{V} = \mathbf{X}\mathbf{W}^V \tag{9.3}$$

$$\mathbf{A} = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right) \tag{9.4}$$

$$\mathbf{O}_{\text{attn}} = \mathbf{A}\mathbf{V} \tag{9.5}$$

$$\mathbf{X}^{(1)} = \text{LayerNorm}(\mathbf{X} + \mathbf{O}_{\text{attn}}) \tag{9.6}$$

Feedforward sublayer:

$$\mathbf{O}_{\text{ffn}} = \mathbf{W}_2 \cdot \text{ReLU}(\mathbf{W}_1\mathbf{X}^{(1)} + \mathbf{b}_1) + \mathbf{b}_2 \tag{9.7}$$

$$\mathbf{X}^{(2)} = \text{LayerNorm}(\mathbf{X}^{(1)} + \mathbf{O}_{\text{ffn}}) \tag{9.8}$$

Output: $\mathbf{X}^{(2)} \in \mathbb{R}^{n \times d}$

This is the complete computation for one transformer layer. Stack L of these layers (typically $L = 6$ to $L = 96$):

$$\mathbf{X}^{(0)} \xrightarrow{\text{Layer 1}} \mathbf{X}^{(1)} \xrightarrow{\text{Layer 2}} \mathbf{X}^{(2)} \xrightarrow{\text{Layer 3}} \cdots \xrightarrow{\text{Layer L}} \mathbf{X}^{(L)}$$

The final output $\mathbf{X}^{(L)}$ is passed to a task-specific head (like a language model head for predicting the next token, or a classification head for sentiment analysis).

9.9.8 Connecting to the full picture

Let's trace the complete flow from raw text to predictions:

1. Tokenization: Text "The cat sat" becomes token IDs `[50, 123, 456]`

2. Embedding: Each token ID is converted to a d-dimensional embedding, and positional encodings are added (discussed in detail in Chapter 11):

$$\mathbf{X}^{(0)} = \text{TokenEmbedding}(\text{tokens}) + \text{PositionalEncoding}(\text{positions})$$

3. Transformer layers: Apply L layers of self-attention + FFN + residuals + layer norms:

$$\mathbf{X}^{(\ell)} = \text{TransformerLayer}_\ell(\mathbf{X}^{(\ell-1)}) \quad \text{for } \ell = 1, 2, \ldots, L$$

4. Output head: The final representation $\mathbf{X}^{(L)}$ is passed to a task-specific head. For language modeling:

$$\text{logits} = \mathbf{X}^{(L)}\mathbf{W}_{\text{vocab}}$$

where $\mathbf{W}_{\text{vocab}} \in \mathbb{R}^{d \times V}$ projects to vocabulary size V. Apply softmax to get probabilities over the vocabulary.

Each component plays a role: - **Embeddings:** Convert discrete tokens to continuous vectors - **Positional encoding:** Inject position information (covered in Chapter 11) - **Self-attention:** Mix information across positions based on relevance - **FFN:** Compute complex nonlinear transformations at each position - **Residual connections:** Enable gradient flow through deep networks - **Layer normalization:** Stabilize activation scales

Together, these components let transformers learn powerful representations from data. The residual connections and layer normalization are crucial glue that makes deep transformers trainable. Without them, we'd be stuck with shallow networks that couldn't learn complex patterns.

9.10 Causal (masked) self-attention

For autoregressive language modeling - where the goal is to predict the next token given previous ones - standard self-attention has a critical flaw: it can see the future.

9.10.1 The problem: Cheating prevents learning

Imagine we are training a model to predict the next word in the sequence "The cat sat on the mat." * Input: "The cat sat on the" * Target Output: "cat sat on the mat"

If we use standard self-attention, when the model is processing the word "sat" (position 3), it can attend to "on" (position 4) and "the" (position 5). The model essentially "sees" the answer key.

If the model can see the future tokens it is supposed to predict, it will learn a trivial shortcut: **just copy the next token**. It won't learn grammar, logic, or world knowledge; it will just become a lookup table.

This is disastrous because at **inference time** (when we actually use the model to generate text), the future doesn't exist yet. We generate one token at a time. If the model relies on seeing the future to make predictions, it will fail completely when that future is hidden.

To force the model to actually *learn language structure*, we must blind it to the future. When processing position i, the model should only be allowed to access positions $1, 2, \ldots, i$.

9.10.2 The solution: Masking

Causal self-attention (or masked self-attention) enforces this constraint by modifying the attention scores before the softmax step. We explicitly mask out any connection from a position to a future position.

Mathematically, we adjust the score matrix $\mathbf{S}$:

$$\mathbf{S}_{ij} = \begin{cases} \frac{\mathbf{q}_i^T \mathbf{k}_j}{\sqrt{d_k}} & \text{if } j \leq i \quad \text{(past and present)} \\ -\infty & \text{if } j > i \quad \text{(future)} \end{cases}$$

Why negative infinity? Recall that we pass these scores through a softmax function: $\text{softmax}(x_i) = \frac{e^{x_i}}{\sum e^{x_k}}$. Since $e^{-\infty} = 0$, setting a score to $-\infty$ ensures that the resulting attention weight is exactly **zero**. The future token is effectively erased from the weighted sum.

9.10.3 Implementation

In practice, this is done efficiently using a **mask matrix M**. For a sequence of length 4, the mask looks like this:

$$\mathbf{M} = \begin{bmatrix} 0 & -\infty & -\infty & -\infty \\ 0 & 0 & -\infty & -\infty \\ 0 & 0 & 0 & -\infty \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

We compute $\mathbf{A} = \text{softmax}(\mathbf{S} + \mathbf{M})$.

- **Row 1:** Can only attend to column 1 (itself).
- **Row 2:** Can attend to columns 1 and 2.
- **Row 4:** Can attend to columns 1, 2, 3, and 4.

This produces a **lower-triangular** attention matrix. This structure allows us to train on all tokens in a sequence simultaneously (parallel training) while mathematically guaranteeing that no prediction depends on future information. This is the mechanism that powers GPT and other decoder-only architectures.

9.11 Summary

Self-attention is the core of transformers:

- Queries, keys, and values all come from the same sequence
- Each position can attend to every other position (or only earlier positions in causal attention)
- Attention weights are computed via scaled dot products followed by softmax
- The output is a weighted combination of values

Key properties:

- $O(n^2 \cdot d)$ complexity enables direct connections between all positions
- Highly parallelizable, unlike sequential RNNs
- Path length of 1 between any two positions
- Learns to extract relevant relationships through training

Self-attention alone processes all positions with identical weights. In the next chapter, we'll see how **multi-head attention** allows the model to attend to different aspects of the input simultaneously.

Chapter 10

Multi-head attention

Learning objectives

After completing this chapter, you will be able to:

- Explain why a single attention head is limiting
- Describe how multiple heads attend to different aspects of the input
- Compute multi-head attention by projecting into subspaces
- Concatenate and project head outputs back to model dimension
- Understand how heads specialize for different types of relationships

In the previous chapter, we built the self-attention mechanism. It allowed a token to look at the entire sequence and calculate a weighted average of other tokens' value vectors. It produced a single context-aware representation.

But there is a subtle problem with using just one attention mechanism.

Think about how you understand a sentence like **"The pilot flew the plane to Paris."**

To fully grasp this scene, your mind performs several distinct "lookups" simultaneously: 1. **Who** performed the action? (You link "flew" to "pilot"). 2. **What** was affected? (You link "flew" to "plane"). 3. **Where** did it happen? (You link "flew" to "Paris").

A single self-attention head computes a *single* probability distribution. It has to decide on one specific pattern of attention. If it attends 80% to "pilot" and 20% to "Paris", it creates a blended representation that is mostly "agent" and a little bit "location". It struggles to be precise about both relationships at the same time. It's like trying to listen to the bass line, the vocals, and the drums in a song all at once, and smashing them into a single audio track.

Multi-head attention gives the model the ability to "listen" to different parts of the input independently. Instead of one global focus, we create multiple parallel attention mechanisms (heads). One head can focus entirely on grammar (finding the subject), while another focuses entirely on geography (finding the location).

10.1 The intuition: Splitting the information bandwidth

To understand how this works mechanically, we need to look at the embedding dimension d differently.

In most transformer models, d is a large number, like 512 or 768. We typically think of this as a single long vector representing the token. But intuitively, we can think of this vector as having a total "bandwidth" or "capacity" of 512 units of information.

We don't need all 512 units just to determine if a word is a subject or an object. That's a relatively simple question. Maybe we only need 64 units of capacity to answer that.

So, instead of using the huge 512-dimensional space to ask one complex question, we **divide our capacity**. We split the model into h smaller, independent "sub-spaces."

- **Standard approach:** One giant head operating in 512 dimensions.
- **Multi-head approach:** 8 separate heads, each operating in 64 dimensions.

Crucially, **we do not just chop the input vector into pieces.** We don't say "Head 1 looks at the first 64 numbers, Head 2 looks at the next 64."

Instead, every head gets to look at the **entire** original 512-dimensional vector. But each head has its own learned "lens" (projection matrix) that extracts only the information relevant to its specific job, compressing it down into a smaller 64-dimensional vector.

10.1.1 The projection: A specialized lens

This is where the projection matrices $\mathbf{W}^Q, \mathbf{W}^K, \mathbf{W}^V$ come in. In multi-head attention, every head i gets its own unique set of matrices: $\mathbf{W}_i^Q, \mathbf{W}_i^K, \mathbf{W}_i^V$.

Imagine the input vector for "Banks" in the sentence "The river banks overflowed." This vector contains a mess of potential meanings: financial institution, river side, turning a plane, verb form, noun form.

1. **Head 1 (The Syntax Expert):** Its projection matrix $\mathbf{W}_1^Q$ is learned to ignore the "financial" or "river" meanings. It projects the 512-dim vector down to a 64-dim vector that purely encodes: *"This is a plural noun."*
2. **Head 2 (The Semantic Expert):** Its projection matrix $\mathbf{W}_2^Q$ ignores the grammar. It projects the same 512-dim input down to a different 64-dim vector that encodes: *"This is a physical object related to water."*

By projecting into these smaller subspaces, the heads can perform attention cleanly. Head 1 will find that "overflowed" (verb) is looking for a subject. Head 2 will find that "river" (water context) matches "banks".

10.2 The mathematical formulation

Now we can formalize this. Let $d_{model} = 512$ be the input dimension and $h = 8$ be the number of heads. We set the dimension of each head to $d_k = d_{model}/h = 64$.

For each head $i = 1 \dots h$:

1. **Project:** We take the same input $\mathbf{X}$ and project it into the head's specific subspace.

$$\mathbf{Q}_i = \mathbf{X}\mathbf{W}_i^Q, \quad \mathbf{K}_i = \mathbf{X}\mathbf{W}_i^K, \quad \mathbf{V}_i = \mathbf{X}\mathbf{W}_i^V$$

   Here, the projection matrices are size $d_{model} \times d_k$ (e.g., 512×64). This effectively "compresses" the information from the full width down to the head's specialized width.

2. **Attend:** We calculate attention independently in this small subspace.

$$\text{head}_i = \text{softmax}\left(\frac{\mathbf{Q}_i\mathbf{K}_i^T}{\sqrt{d_k}}\right)\mathbf{V}_i$$

   The result head_i is a sequence of vectors of size d_k (64).

3. **Concatenate:** This is the re-assembly phase.

   At this point, for a single token, we have 8 separate vectors, each of size 64.

   - **head 1:** [Grammar features]

- **head 2:** [Context features]
- …
- **head 8:** [Position features]

To move forward in the network, we need to return to our standard interface: a single vector of size 512. We do this by simply placing the 8 vectors side-by-side to form one long vector.

$$\text{MultiHeadOutput} = \Big[\underbrace{\mathbf{v}_1}_{\text{Head 1}} \Big| \underbrace{\mathbf{v}_2}_{\text{Head 2}} \Big| \dots \Big| \underbrace{\mathbf{v}_8}_{\text{Head 8}} \Big]$$

The result is a vector of length $64 + 64 + \cdots + 64 = 512$.

Crucially, this restored vector is "segregated." The first 64 numbers only contain information from Head 1. The next 64 only from Head 2. They haven't talked to each other yet. If we stopped here, the next layer would receive a disjointed input where syntax information lives in one block and semantic information in another.

4. **Final linear projection (the mixer):**

We now have a concatenated vector of size 512. But there is a problem: it is **segregated**. The first 64 numbers come exclusively from Head 1. The next 64 come from Head 2. These segments are neighbors, but they haven't interacted. Head 1 might know the word is "banks" (plural noun), and Head 2 might know the context is "river" (nature), but no single number in the vector represents "river banks".

To fix this, we apply a final linear transformation using a weight matrix $\mathbf{W}^O$ (size 512×512).

$$\mathbf{Z} = \text{MultiHeadOutput} \times \mathbf{W}^O$$

10.2.1 How the math mixes information

To understand *why* this mixes the heads, look at the linear algebra operation for a single output value. Let the concatenated input vector be $\mathbf{h} = [h_1, h_2, \dots, h_{512}]$ and the output vector be $\mathbf{z} = [z_1, z_2, \dots, z_{512}]$.

The j-th feature of the output is calculated as:

$$z_j = \sum_{i=1}^{512} h_i \cdot W_{ij}^O$$

Look closely at the summation range ($i = 1$ to 512). It iterates over the **entire** length of the concatenated vector.

- Indices $1 \dots 64$ come from **Head 1**.
- Indices $65 \dots 128$ come from **Head 2**.
- …and so on.

This means every single value z_j in the output is a **weighted sum of all heads simultaneously**. The matrix $\mathbf{W}^O$ acts as a massive mixing desk. It can say: *"To create output feature 5, take 20% of Head 1's syntax signal, add 50% of Head 2's semantic signal, and subtract 10% of Head 3's position signal."*

10.2.2 Discovering the weights

What is inside $\mathbf{W}^O$? Ideally, it contains the perfect "recipes" for combining these diverse signals. But we don't hand-code them.

$\mathbf{W}^O$ consists of $512 \times 512 = 262,144$ learnable parameters. Initially, these are set to small random numbers. We "discover" their optimal values through **training (backpropagation)**.

- If the model makes a prediction error because it failed to combine "subject" and "verb" correctly, the gradient descent algorithm calculates exactly how to adjust the weights in $\mathbf{W}^O$.
- Over billions of training examples, $\mathbf{W}^O$ evolves from random noise into a highly optimized routing system that knows exactly which heads contain relevant information for which tasks, and how to blend them to produce the most useful representation for the next layer.

10.3 Why divide the dimension?

You might ask: *Why not just have 8 heads that are all 512 dimensions wide? Why do we have to shrink them to 64?*

Two reasons: **Computational cost** and **Parameter count**.

If we had 8 full-size heads, the computation would be 8 times more expensive than a single head. By shrinking the dimension by a factor of h, the math works out beautifully:

- **Single Head:** 1 dot product of size 512.
- **Multi Head:** 8 dot products of size 64.

Since $8 \times 64 = 512$, the total number of floating-point operations for the projections and the final fusion is roughly the same as a single giant head. We get the benefit of multiple independent attention patterns "for free" in terms of compute budget.

10.4 Summary

Multi-head attention is the defining feature that allows Transformers to understand the nuance of language.

1. **Decomposition:** It decomposes the complex input vector into specialized lower-dimensional subspaces.
2. **Specialization:** Different heads learn to look for different things (grammar, coreference, context).
3. **Recomposition:** It fuses these diverse insights back into a unified representation.

It transforms the "bag of vectors" into a rich, multi-layered web of relationships. However, despite all this sophistication, our model still has a glaring hole: it has no idea that "The" comes before "cat". In the next chapter, we will fix this with **Positional Encoding**.

Chapter 11

Positional encoding

Learning objectives

After completing this chapter, you will be able to:

- Explain why attention is permutation-invariant and why this is problematic
- Describe the sinusoidal positional encoding scheme
- Compute positional encodings for any position and dimension
- Understand how different frequencies encode position at different scales
- Derive why positional encodings allow learning relative positions

We've developed embeddings that map words to dense vectors, and attention mechanisms that let positions exchange information based on relevance. But there's a fundamental problem lurking in the mathematics: attention has no notion of position. The self-attention operation we derived is completely permutation-invariant. If we shuffle the input sequence, the attention weights change (because different positions now contain different content), but the mechanism treats position 1 the same as position 100. There's nothing in the math that says "this token came first" or "these tokens are adjacent."

Why is this a problem? Consider "The dog bit the man" versus "The man bit the dog." Same words, completely different meanings. Order matters in language. Recurrent networks handled this naturally because they processed sequences step by step. The hidden state at time t inherently knows it came after time $t-1$. But transformers process all positions in parallel, gaining speed at the cost of losing positional information. This chapter develops positional encoding, the mechanism that injects order into the permutation-invariant attention operation.

11.1 The position problem

Let's make the problem concrete. When we compute self-attention, we project inputs to queries, keys, and values, then compute attention scores via dot products:

$$\alpha_{ij} = \frac{\exp(\mathbf{q}_i^T \mathbf{k}_j / \sqrt{d})}{\sum_k \exp(\mathbf{q}_i^T \mathbf{k}_k / \sqrt{d})}$$

The score between position i and position j depends only on the content at those positions (via $\mathbf{q}_i$ and $\mathbf{k}_j$), not on the fact that position i comes before or after position j. If we swap the content of positions 2 and 5, the attention pattern changes, but not because of position. The mechanism has no way to know that position 2 is near position 1 while position 5 is far away.

We could add position as a feature: append the position index to each embedding. Suppose we have a 512-dimensional embedding for the word "cat" at position 3. We could create a 513-dimensional vector by appending the number 3: [cat embedding, 3]. But this has serious issues.

First, positions would be unbounded integers (1, 2, 3, ..., 1000, ...). A model trained on sequences of length 100 would see position indices from 1 to 100. At test time, if we feed it a sequence of length 200, it encounters position indices 101-200 that it has never seen during training. How should it handle position 150? It has no training data for that value. The model can't generalize to longer sequences.

Second, the relative scale matters. Consider the distance between consecutive positions. Positions 1 and 2 differ by 1. Positions 100 and 101 also differ by 1. But if we use raw integers, the relative difference is very different: going from 1 to 2 is a 100% increase, while going from 100 to 101 is only a 1% increase. Should early positions be treated as more "spread out" than later positions? That seems arbitrary. We want a representation where the difference between consecutive positions is consistent regardless of absolute position.

Third, position indices have unbounded magnitude. The first position might be represented as 1, but the 10,000th position is represented as 10,000. When these values interact with learned weights in attention, the very large values at late positions could dominate the dot products, creating numerical instability. We need positions to have bounded, comparable magnitudes.

11.2 Adding positional information

The transformer uses **positional encoding**: we add a position-dependent vector to each token's embedding. If $\mathbf{e}_t$ is the embedding for token t and $\mathbf{p}_t$ is the positional encoding for position t, the input to the transformer is:

$$\mathbf{x}_t = \mathbf{e}_t + \mathbf{p}_t$$

The positional encoding $\mathbf{p}_t \in \mathbb{R}^d$ must have the same dimension d as the embeddings so we can add them. We're not concatenating (which would increase dimension), we're adding. This means position information gets mixed with content information in the same vector space. Each dimension of the resulting vector $\mathbf{x}_t$ contains both "what is this token?" and "where is this token?" information.

Why add rather than concatenate? Adding keeps the dimension constant, which simplifies the architecture. More importantly, it forces the model to learn how to disentangle position and content. The embedding space becomes richer: similar words at similar positions will have similar vectors, but the same word at different positions will differ slightly. The attention mechanism can learn to use or ignore positional information as needed for different tasks.

The key question is: how do we construct $\mathbf{p}_t$? We need a function that maps each position t to a d-dimensional vector, satisfying several desiderata. The positions should have bounded values (to avoid numerical issues), nearby positions should have similar encodings (to support learning that adjacent words are related), and the encoding should generalize to unseen positions (to handle sequences longer than those seen during training).

11.3 Sinusoidal positional encoding

Let's start with what's actually in a positional encoding vector, then understand why. Suppose we have $d = 8$ dimensions (in practice, transformers use 512 or 768, but 8 is easier to visualize). Here are the positional encoding vectors for the first few positions:

Position 0:

$$\mathbf{p}_0 = [0.00, 1.00, 0.00, 1.00, 0.00, 1.00, 0.00, 1.00]$$

Position 1:

$$\mathbf{p}_1 = [0.84, 0.54, 0.10, 0.99, 0.01, 1.00, 0.00, 1.00]$$

Position 2:

$$\mathbf{p}_2 = [0.91, -0.42, 0.20, 0.98, 0.02, 1.00, 0.00, 1.00]$$

Position 3:

$$\mathbf{p}_3 = [0.14, -0.99, 0.30, 0.95, 0.03, 1.00, 0.00, 1.00]$$

Notice the pattern. The first two dimensions change rapidly from position to position. The middle dimensions change more slowly. The last two dimensions barely change at all. Each pair of dimensions captures position information at a different timescale.

Let's see how this is constructed. The formula is:

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

Here $p_{t,j}$ is the j-th dimension of the positional encoding at position t. The formula tells us: - Dimensions 0 and 1 form a pair using frequency $\omega_0 = 1/10000^{0/8} = 1$ - Dimensions 2 and 3 form a pair using frequency $\omega_1 = 1/10000^{2/8} = 0.1$ - Dimensions 4 and 5 form a pair using frequency $\omega_2 = 1/10000^{4/8} = 0.01$ - Dimensions 6 and 7 form a pair using frequency $\omega_3 = 1/10000^{6/8} = 0.001$

Each pair uses sine for the even dimension and cosine for the odd dimension, at their shared frequency.

11.3.1 How different frequencies encode position

The key insight is that **different frequencies capture position at different scales**. Imagine plotting each frequency band across positions: high-frequency bands oscillate rapidly, completing many cycles, while low-frequency bands change very slowly. Together, they create a unique "fingerprint" for each position.

High frequency (dimensions 0-1, $\omega_0 = 1$): Completes one full cycle every $2\pi \approx 6$ positions. Changes rapidly from position to position. These dimensions can distinguish "position 5" from "position 6" but cycle back to similar values every 6 positions. Good for fine-grained, local position information.

Medium frequency (dimensions 2-3, $\omega_1 = 0.1$): Completes one full cycle every $2\pi/0.1 \approx 63$ positions. Changes more slowly. These dimensions are similar for nearby positions (5 and 6 have almost the same value) but different for distant positions (5 and 50 differ significantly). Good for medium-scale position information.

Low frequency (dimensions 4-5, $\omega_2 = 0.01$): Completes one full cycle every $2\pi/0.01 \approx 628$ positions. Changes very slowly. Position 5 and position 50 have nearly identical values in these dimensions. But position 5 and position 500 differ. Good for coarse position information, distinguishing "early in sequence" from "late in sequence."

Very low frequency (dimensions 6-7, $\omega_3 = 0.001$): Barely changes across typical sequences (one cycle every ~6280 positions). Acts like a constant "bias" that varies only across very long sequences.

By combining all frequency bands, each position gets a unique d-dimensional "fingerprint." Position 5 might share some values with position 6 (in the low-frequency dimensions), but they differ in high-frequency dimensions. Position 5 and position 500 differ in both medium and low-frequency dimensions.

11.3.2 Visualizing the full encoding matrix

Let's see what the complete positional encoding looks like for many positions and dimensions. Each cell shows the value $p_{t,j}$ at position t and dimension j:

Look at the pattern. The leftmost dimensions (0-1, high frequency) create tight vertical stripes, changing rapidly from position to position. The rightmost dimensions (14-15, low frequency) change very slowly, appearing almost constant across nearby positions. Each row (each position) has a unique pattern of bright and dark cells across the 16 dimensions.

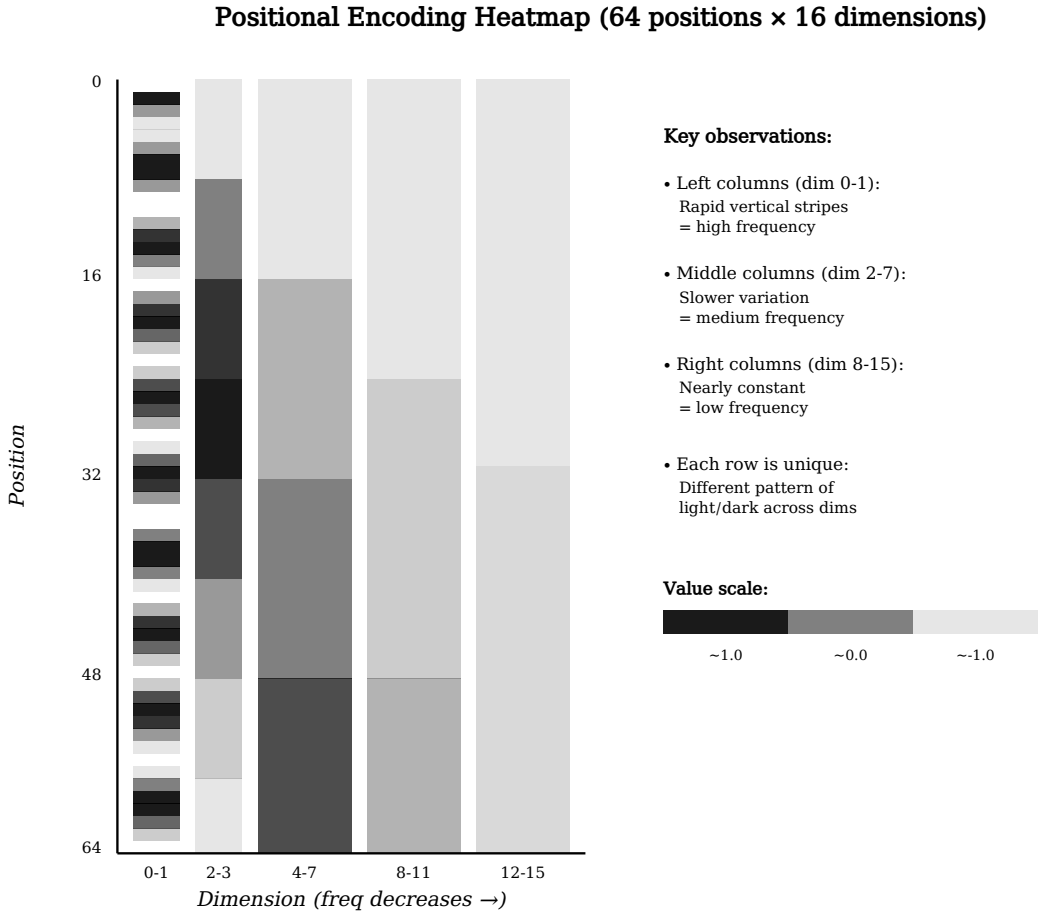

Figure 11.1: Heatmap of positional encodings. Rows are positions (0-63), columns are dimensions (0-15). Bright values are near +1, dark values are near -1. Left columns (low dimension indices, high frequency) show rapid vertical stripes indicating fast oscillation across positions. Right columns (high dimension indices, low frequency) show slow gradual changes. Each position (each row) has a unique pattern across dimensions.

11.3.3 Why both sine and cosine?

Looking at our example vectors again:

$$\mathbf{p}_0 = [0.00, 1.00, 0.00, 1.00, ...]$$

$$\mathbf{p}_1 = [0.84, 0.54, 0.10, 0.99, ...]$$

Notice dimensions 0 and 1: at position 0, we have $(sin, cos) = (0.00, 1.00)$. At position 1, we have $(0.84, 0.54)$. Why do we need both? Why not just use $[0.00, 0.00, ...]$ and $[0.84, 0.10, ...]$ (only sines)?

The problem is **ambiguity**. Consider $\sin(\theta)$. If we know $\sin(\theta) = 0.5$, we can't determine θ uniquely. It could be 30° or it could be 150°. Two different positions would have identical sine values, making them indistinguishable.

But if we know both $\sin(\theta) = 0.5$ AND $\cos(\theta) = 0.866$, we can uniquely determine $\theta = 30°$. The pair (sin, cos) pins down the position on the unit circle. The two functions are 90° out of phase, providing complementary information.

Geometrically, as position increases, the pair $(\sin(\omega t), \cos(\omega t))$ traces a circular path. Each position gets a unique point on the circle (until wrapping around after one full cycle). With multiple frequency bands, we get multiple circles rotating at different speeds, creating a rich, unique encoding for each position.

11.3.4 Why this encoding works

Looking back at our position 5 encoding $\mathbf{p}_5 = [-0.96, 0.28, 0.48, 0.88, 0.05, 0.999, 0.005, 1.000]$, why is this better than simpler alternatives like $\mathbf{p}_5 = [5, 5, 5, ...]$?

Bounded values. All values stay between -1 and 1, regardless of position. Position 1 and position 10,000 both have values in $[-1, 1]$. If we used $\mathbf{p}_t = [t, t, t, ...]$, position 10,000 would have huge values that could destabilize training (gradients would explode, attention scores would saturate).

Unique fingerprints. Each position gets a unique pattern across dimensions. Even though dimension 7 is nearly 1.000 for all positions (low frequency barely varies), the combination of all 8 dimensions uniquely identifies each position. The heatmap we saw earlier shows this: each row (each position) has a distinct pattern of light and dark cells.

Smoothness. Adjacent positions are similar. We saw that position 5 and 6 differ mainly in the high-frequency dimensions (0-1) but are nearly identical in low-frequency dimensions (4-7). This helps the model generalize: what it learns about position 5 can transfer to nearby position 6. With random initialization, positions 5 and 6 might be arbitrarily far apart in embedding space.

Relative position encoding. This is the most important property. The sinusoidal structure means the model can learn to detect relative positions. The encoding for position $t + k$ is mathematically related to the encoding for position t via rotation matrices. Without diving into the full derivation, the key is: if the model wants to implement "look 3 tokens back," it can learn a single transformation that works everywhere, rather than learning separate patterns for "look back from position 10" versus "look back from position 100."

11.3.5 Why these specific frequencies?

The constant 10000 in $\omega_i = 1/10000^{2i/d}$ might seem arbitrary. Why not 100 or 100000?

The choice determines the range of wavelengths (how fast the slowest frequency varies). With 10000 and $d = 512$: - Fastest frequency: wavelength 6 positions - Slowest frequency: wavelength 62,800 positions

This range covers typical sequence lengths (512-2048 tokens) while ensuring the slowest frequency provides a stable "regional" signal that barely varies across the sequence.

If we used 100 instead, the slowest frequency would have wavelength 628 positions. For a sequence of length 2048, even the slowest frequency would complete 3 full cycles, losing its role as a stable coarse indicator. If

we used 100000, the slowest frequency would barely move even across sequences of 10,000 tokens, wasting representational capacity.

The value 10000 is a practical compromise for sequences up to several thousand tokens.

11.3.6 Computing the encoding step-by-step

Let's work through the formula with $d = 8$ and compute the encoding for position $t = 5$.

Step 1: Compute frequencies

For $d = 8$, we have 4 frequency bands ($d/2 = 4$ pairs of dimensions):

$$\omega_0 = \frac{1}{10000^{0/8}} = 1.0, \quad \omega_1 = \frac{1}{10000^{2/8}} = 0.1$$

$$\omega_2 = \frac{1}{10000^{4/8}} = 0.01, \quad \omega_3 = \frac{1}{10000^{6/8}} = 0.001$$

Step 2: Apply formula for each dimension

At position $t = 5$:

Dim	Formula	Computation	Value
0	$\sin(\omega_0 \cdot t)$	$\sin(1.0 \cdot 5) = \sin(5)$	-0.96
1	$\cos(\omega_0 \cdot t)$	$\cos(1.0 \cdot 5) = \cos(5)$	0.28
2	$\sin(\omega_1 \cdot t)$	$\sin(0.1 \cdot 5) = \sin(0.5)$	0.48
3	$\cos(\omega_1 \cdot t)$	$\cos(0.1 \cdot 5) = \cos(0.5)$	0.88
4	$\sin(\omega_2 \cdot t)$	$\sin(0.01 \cdot 5) = \sin(0.05)$	0.05
5	$\cos(\omega_2 \cdot t)$	$\cos(0.01 \cdot 5) = \cos(0.05)$	0.999
6	$\sin(\omega_3 \cdot t)$	$\sin(0.001 \cdot 5) = \sin(0.005)$	0.005
7	$\cos(\omega_3 \cdot t)$	$\cos(0.001 \cdot 5) = \cos(0.005)$	1.000

Step 3: Assemble the vector

$$\mathbf{p}_5 = [-0.96, 0.28, 0.48, 0.88, 0.05, 0.999, 0.005, 1.000]$$

This is the 8-dimensional positional encoding for position 5. Notice how: - Dimensions 0-1 (high frequency) have values far from their starting point $(0, 1)$ - Dimensions 2-3 (medium frequency) have changed moderately - Dimensions 4-5 (low frequency) have barely changed - Dimensions 6-7 (very low frequency) are almost identical to position 0

Now let's compare positions 5 and 6 to see how the encoding distinguishes adjacent positions:

Dim	Pos 5	Pos 6	Difference
0	-0.96	-0.28	**0.68** (large)
1	0.28	0.96	**0.68** (large)
2	0.48	0.56	0.08 (small)
3	0.88	0.83	0.05 (small)
4-7	same	same	0 (tiny)

The high-frequency dimensions (0-1) change significantly between adjacent positions, allowing the model to distinguish "position 5" from "position 6." The low-frequency dimensions stay nearly constant, providing a stable "regional" signal that groups nearby positions together.

This multi-scale representation is the key insight. High-frequency dimensions distinguish adjacent positions. Low-frequency dimensions distinguish distant regions. Together, they provide both fine-grained and coarse-grained positional information, creating a unique fingerprint for each position.

11.4 Learned positional embeddings

Modern transformers often use **learned positional embeddings** instead of fixed sinusoidal encodings. We treat position encodings like word embeddings and learn them during training. We create a learnable matrix $\mathbf{P} \in \mathbb{R}^{L_{\max} \times d}$ where $L_{\max}$ is the maximum sequence length we plan to handle. Row t of $\mathbf{P}$ is the encoding for position t. These encodings are initialized randomly and updated during training via backpropagation.

The advantage is simplicity. We don't need to commit to a particular functional form (sinusoids). The model learns whatever positional representation is most useful for the task. Empirically, learned positional embeddings often work as well as or better than sinusoidal encodings.

The disadvantage is generalization to longer sequences. If we train with $L_{\max} = 512$, we have learned encodings for positions 0 through 511. At test time, if we encounter a sequence of length 1000, we have no learned encoding for positions 512-999. We could extrapolate by reusing the learned encodings (wrapping around, or extending the pattern), but there's no guarantee this works well. Sinusoidal encodings generalize naturally to any length because they're defined by a function, not a lookup table.

In practice, this limitation is often acceptable. Most applications have a known maximum sequence length, and we train on sequences up to that length. For applications requiring variable or very long sequences, researchers use alternative approaches like relative positional encodings (where we encode the distance between positions rather than absolute position) or rotary positional embeddings (which combine learned and functional forms).

11.5 What if we skip positional encoding?

What happens if we skip positional encoding entirely? The transformer can still process the sequence, but it loses all sense of order. It becomes a bag-of-words model, treating "dog bites man" and "man bites dog" identically. The attention mechanism can still find relevant words (if you're processing "dog" you might attend to "man" and "bites"), but it can't distinguish "the word before this one" from "the word after this one."

For some tasks, this might be tolerable. Consider sentiment classification: determining whether a movie review is positive or negative. Word order matters ("not good" vs "good"), but for many reviews, the overall sentiment is clear from the words present regardless of order. A bag-of-words model can achieve reasonable accuracy. But for most language tasks, order is crucial. In machine translation, "Le chat noir" (the black cat) and "Le noir chat" (the black cat, but ungrammatical in French) have different meanings and grammaticality. In question answering, "Who did Alice meet?" and "Who met Alice?" are different questions. Without positional encoding, the transformer can't distinguish these.

We can verify this experimentally. If we train a transformer without positional encoding on a task that requires understanding word order (like parsing or translation), performance collapses. The model learns something, but far less than with positional encoding. This confirms that the permutation-invariance of bare attention is a bug, not a feature, for sequential data.

Conversely, positional encoding is only necessary because we chose attention as our mixing mechanism. If we used a different architecture (like a CNN with positional filters, or an RNN with sequential processing), position would be implicit. We use positional encoding specifically to fix the position-blindness of attention, gaining the benefits of parallel processing and long-range dependencies while recovering the sequential structure that language requires.

We've now added position to our tokens. The input to the transformer is embeddings plus positional encoding, giving each position a unique combination of content and location information. The next step is to process

these position-aware embeddings through the transformer's core architecture: stacked blocks of attention and feed-forward networks that refine the representations layer by layer.

Part III

The transformer architecture

Chapter 12

The transformer

Learning objectives

After completing this chapter, you will be able to:

- Trace information flow through the complete encoder-decoder transformer
- Compute forward propagation with explicit matrix dimensions
- Derive backward propagation gradients for all transformer components
- Distinguish encoder-only, decoder-only, and encoder-decoder architectures
- Specify the complete transformer with sufficient detail for implementation

We now mathematically derive the complete transformer architecture from first principles (Vaswani et al. 2017). This chapter presents every computational step in both forward and backward propagation, with explicit dimensions, indexing, and gradient computations.

12.1 Notation and hyperparameters

We work with the following standard transformer configuration:

- $V = 50000$: vocabulary size (number of unique tokens)
- $d_{model} = 512$: model dimension (dimensionality of all embeddings, hidden states, and layer outputs throughout the network)
- $d_k = d_v = 64$: key/value dimension per head (each attention head projects to this lower dimension, where $d_k = d_{model}/h$)
- $d_{ff} = 2048$: feed-forward intermediate dimension (hidden layer size in the position-wise FFN)
- $h = 8$: number of attention heads (parallel attention mechanisms, where $h \cdot d_k = d_{model}$)
- $N = 6$: number of encoder/decoder blocks (depth of the network)
- n: source sequence length (number of tokens in the input sequence)
- m: target sequence length (number of tokens in the output sequence)
- $\epsilon = 10^{-6}$: numerical stability constant (small value added in denominators to prevent division by zero)

Matrices are denoted with bold uppercase ($\mathbf{W}$), vectors with bold lowercase ($\mathbf{x}$), and scalars with regular font (x). We use 1-indexing for positions and dimension indices in mathematical expressions.

12.2 Architecture overview

This section builds a complete mental model of the transformer without mathematics. We describe what each component does and why, tracing information flow from input to output. The subsequent sections provide the precise mathematical formulation needed for implementation.

12.2.1 The big picture

The transformer solves sequence-to-sequence problems like machine translation. Given source sequence "The cat sat on the mat" in English, it produces target sequence "Le chat s'est assis sur le tapis" in French. The architecture has two main components working together: an encoder that understands the source sequence, and a decoder that generates the target sequence one token at a time.

The key innovation is attention: instead of compressing the entire input into a single fixed-size vector like recurrent networks do, the transformer lets every position directly access information from every other position. This eliminates the information bottleneck and enables parallel processing.

12.2.2 From discrete tokens to continuous vectors

Text arrives as discrete tokens (words or subwords). We need continuous vectors to perform meaningful computation. The embedding layer maps each token to a point in a high-dimensional space where similar meanings cluster together. The word "cat" might map to vector $[0.2, -0.5, 0.8, ...]$ with 512 dimensions. Words with similar meanings like "kitten" or "feline" map to nearby points in this space.

But there's a problem: these embeddings contain no information about word order. The sequences "dog bites man" and "man bites dog" would have identical representations if we just summed their embeddings. We need to inject position information.

12.2.3 Positional encoding: giving order to chaos

Positional encoding adds a unique signature to each position in the sequence. Position 1 gets one pattern, position 2 gets a different pattern, and so on. These patterns are carefully designed using sine and cosine functions at different frequencies. Low frequencies change slowly across positions, encoding coarse position information. High frequencies change rapidly, encoding fine-grained distinctions between nearby positions.

We add (element-wise) these positional signatures to the token embeddings. Now the embedding for "cat" at position 2 is different from "cat" at position 5, even though they represent the same word. The model can use this positional information throughout all subsequent computations.

12.2.4 The encoder: building rich representations

The encoder transforms the input sequence into a rich, contextualized representation. Each position starts with its embedding plus positional encoding. Then we apply six identical blocks in sequence, with each block refining the representation.

Self-attention mechanism. Each encoder block begins with self-attention. This is where the magic happens. For each position, we ask: "Which other positions in the sequence are relevant for understanding this position?" The word "it" might strongly attend to "cat" from earlier in the sentence to resolve what "it" refers to. The verb "sat" might attend to its subject "cat" to verify subject-verb agreement.

Attention works by computing compatibility scores between positions. For position i, we compute how well it matches with every position j in the sequence. These scores become weights in a weighted average: positions that are highly relevant contribute more to the output, irrelevant positions contribute little.

Multi-head attention. We don't compute just one attention pattern. We compute eight parallel attention mechanisms (heads), each learning to capture different types of relationships. One head might specialize in syntactic dependencies (subject-verb relationships). Another might capture coreference (pronouns linking to their antecedents). Another might focus on semantic similarity. The model learns these specializations automatically during training.

Each head operates in a lower-dimensional space (64 dimensions instead of 512) for computational efficiency. After computing attention in parallel, we concatenate all heads back together and project to the original dimension. This gives us a rich representation that combines multiple perspectives on the input.

Feed-forward transformation. After attention gathers information from across the sequence, we apply a position-wise feed-forward network. This is a simple two-layer neural network with a ReLU activation, applied independently to each position. Think of this as giving the model capacity to perform complex nonlinear transformations on the mixed information from attention.

The feed-forward network first expands the representation to a higher dimension (2048), applies ReLU to introduce nonlinearity, then projects back down to the model dimension (512). This happens at each position independently, allowing the model to refine representations without mixing information across positions (attention already did that).

Residual connections and layer normalization. Both the attention and feed-forward sub-layers use residual connections: we add the input directly to the output before passing to the next layer. This creates shortcut paths for gradients during training, enabling very deep networks without vanishing gradients.

After each residual connection, we apply layer normalization, which standardizes the values across the embedding dimensions for each position independently. This keeps activations in a stable range, making training more reliable.

Stacking blocks. We apply six encoder blocks in sequence. Early blocks learn shallow patterns like syntax and local word relationships. Middle blocks capture more complex dependencies. Deep blocks learn high-level semantic relationships and long-range dependencies. The final encoder output contains deeply contextualized representations where each position has gathered relevant information from the entire sequence through multiple rounds of attention.

12.2.5 The decoder: generating the target sequence

The decoder generates the output sequence autoregressively, one token at a time. During training, we have the complete target sequence and process it in parallel with teacher forcing. During inference, we generate token by token, feeding each new token back as input for the next prediction.

Masked self-attention. The decoder's first sub-layer is masked self-attention. Like encoder self-attention, this lets positions attend to each other. But there's a crucial constraint: position i can only attend to positions 1 through i, never to future positions. This causal masking ensures the model can't "cheat" by looking ahead to tokens it's supposed to predict.

We implement masking by setting attention scores to negative infinity for future positions. After softmax, these become zero weights, effectively blocking information flow from the future. This makes generation possible: at inference time, when we only have partial sequences, the model uses the same computation pattern it learned during training.

Cross-attention: connecting encoder and decoder. The decoder's second sub-layer is where encoder and decoder communicate. In cross-attention, queries come from the decoder (what we're currently generating), while keys and values come from the encoder output (the source sequence we're translating from).

For each decoder position, cross-attention asks: "Which source positions are relevant for generating this target word?" When generating the French word "chat", the decoder might strongly attend to the English word "cat". When generating "assis" (sat), it attends to "sat". The model learns these alignments automatically without explicit supervision.

Unlike self-attention, cross-attention isn't masked. All decoder positions can attend to all encoder positions. The source sequence is fixed and fully available, so there's no information leakage concern.

Feed-forward network. After gathering information from both the previous target tokens (via masked self-attention) and the source sequence (via cross-attention), the decoder applies the same position-wise feed-forward network as the encoder. This refines the combined representation.

Stacking decoder blocks. We stack six decoder blocks. Each block refines the representation by attending to (1) previous target positions, (2) the source sequence, and (3) applying nonlinear transformations. By the final decoder block, each position has a rich representation that encodes both what has been generated so far and relevant information from the source.

12.2.6 Output projection: from representations to tokens

The final decoder output is a matrix where each row represents one target position. We need to convert these continuous representations into probability distributions over the vocabulary. A linear layer projects from the model dimension (512) to the vocabulary size (50,000), producing logits for each token. Softmax converts these logits to probabilities.

For position i, the probability distribution tells us how likely each vocabulary token is as the next word. During training, we compare this distribution to the true next token and compute cross-entropy loss. During inference, we sample from this distribution (or take the argmax for greedy decoding).

12.2.7 Information flow through the network

Let's trace how information flows through the complete architecture for translating "The cat" to "Le chat".

Encoder path: Tokens "The" and "cat" become embeddings, receive positional encodings, and enter the encoder. In the first encoder block, self-attention lets "cat" gather context from "The" and vice versa. The feed-forward network processes each position. Residual connections and layer normalization maintain training stability. This repeats through six blocks, producing deeply contextualized representations of the English sentence.

Decoder path: We start with target tokens [START], "Le", "chat". Masked self-attention ensures position 1 only sees [START], position 2 sees [START, Le], position 3 sees all three. Cross-attention connects each target position to the English words: "Le" attends strongly to "The", "chat" attends to "cat". Feed-forward processing refines these representations. This repeats through six decoder blocks.

Output: The final decoder state for position 1 produces probabilities over the vocabulary. The model has learned that after [START], when translating "The cat", the word "Le" has high probability. Position 2's state predicts "chat" with high probability. Position 3 predicts [END].

12.2.8 Why this architecture works

The transformer's power comes from several key design choices. Attention provides direct paths between all positions, eliminating the information bottleneck of sequential processing. Multi-head attention captures diverse relationships in parallel. The feed-forward network adds expressiveness for complex transformations. Residual connections enable deep stacking. Layer normalization stabilizes training. Positional encoding injects order information while maintaining the permutation-equivariance of attention.

The result is an architecture that scales beautifully. By increasing model dimension, number of heads, number of layers, and training data, we get models ranging from small 86-million-parameter base transformers to massive 175-billion-parameter models like GPT-3. The fundamental computation pattern remains the same across scales.

12.2.9 Clarifying terminology: architecture vs phases

The terms "encoder" and "decoder" refer to architectural components, not to training versus inference phases. This distinction is important.

Encoder and decoder are attention masking patterns:

- **Encoder architecture** uses bidirectional self-attention. Every position can attend to every other position. This is used for understanding tasks where we have the complete input.
- **Decoder architecture** uses causal (masked) self-attention. Position i can only attend to positions $1, \ldots, i$. This prevents information leakage from future tokens.

Both architectures are used in training AND inference:

In a decoder-only model like GPT, the same masked self-attention architecture is used throughout. During training, we feed complete sequences and compute loss at every position. The model learns weight matrices

($\mathbf{W}^Q$, $\mathbf{W}^K$, $\mathbf{W}^V$, etc.) via backpropagation. During inference, we use those same weights with the same masked attention pattern to generate tokens one at a time.

Similarly, encoder-only models like BERT use bidirectional attention in both training and inference. The architecture doesn't change between phases.

What actually changes between training and inference:

Aspect	Training	Inference
Weights	Updated via backpropagation	Fixed (no updates)
Input	Complete sequences with known targets	Partial or complete sequences
Output	Loss value for optimization	Predictions or generated tokens
Generation	Teacher forcing (use true targets)	Autoregressive (use own predictions)

For decoder models, the key difference at inference is autoregressive generation: we generate one token, append it to the sequence, and repeat. But the underlying computation (masked self-attention with the learned weights) is identical to what happened during training.

With this conceptual foundation in place, we now turn to the precise mathematical specification.

12.3 Forward propagation

We now present the precise mathematical formulation of each component described in the architecture overview. Every operation is specified with explicit dimensions and matrix operations.

12.3.1 Input embedding and positional encoding

Given source sequence with token indices $\mathbf{t} = [t_1, t_2, \ldots, t_n]$ where $t_i \in \{1, 2, \ldots, V\}$.

Embedding lookup: The embedding matrix $\mathbf{E} \in \mathbb{R}^{V \times d_{model}}$ maps each token to a vector. For the sequence, we extract rows corresponding to token indices:

$$\mathbf{X}_{embed} = \mathbf{E}[\mathbf{t}, :] \in \mathbb{R}^{n \times d_{model}}$$

where row i contains the embedding for token t_i.

Positional encoding: For each position $i \in \{1, \ldots, n\}$ and dimension $j \in \{1, \ldots, d_{model}\}$:

$$p_{ij} = \begin{cases} \sin\left(\frac{i-1}{10000^{(j-1)/d_{model}}}\right) & \text{if } j \text{ is odd} \\ \cos\left(\frac{i-1}{10000^{(j-2)/d_{model}}}\right) & \text{if } j \text{ is even} \end{cases}$$

This forms positional encoding matrix $\mathbf{P} \in \mathbb{R}^{n \times d_{model}}$.

Combined input:

$$\mathbf{X}^{(0)} = \mathbf{X}_{embed} + \mathbf{P} \in \mathbb{R}^{n \times d_{model}}$$

12.3.2 Encoder

Each encoder block applies: (1) multi-head self-attention, (2) residual + layer norm, (3) feed-forward network, (4) residual + layer norm.

12.3.2.1 Multi-head self-attention

Given input $\mathbf{X} \in \mathbb{R}^{n \times d_{model}}$ to block ℓ, for each head $k \in \{1, \ldots, h\}$:

Step 1: Project to queries, keys, values

$$\mathbf{Q}_k = \mathbf{X}\mathbf{W}_k^Q \in \mathbb{R}^{n \times d_k}$$

$$\mathbf{K}_k = \mathbf{X}\mathbf{W}_k^K \in \mathbb{R}^{n \times d_k}$$

$$\mathbf{V}_k = \mathbf{X}\mathbf{W}_k^V \in \mathbb{R}^{n \times d_k}$$

where $\mathbf{W}_k^Q, \mathbf{W}_k^K, \mathbf{W}_k^V \in \mathbb{R}^{d_{model} \times d_k}$ are learnable projection matrices.

Step 2: Compute attention scores

$$\mathbf{S}_k = \frac{\mathbf{Q}_k \mathbf{K}_k^T}{\sqrt{d_k}} \in \mathbb{R}^{n \times n}$$

Entry $S_{k,ij}$ measures the compatibility between position i and position j.

Step 3: Apply softmax

$$\mathbf{A}_k = \text{softmax}(\mathbf{S}_k) \in \mathbb{R}^{n \times n}$$

where softmax is applied row-wise: $A_{k,ij} = \frac{\exp(S_{k,ij})}{\sum_{j'=1}^{n} \exp(S_{k,ij'})}$. Each row sums to 1.

Step 4: Compute weighted sum

$$\mathbf{H}_k = \mathbf{A}_k \mathbf{V}_k \in \mathbb{R}^{n \times d_k}$$

Step 5: Concatenate heads

$$\mathbf{H} = [\mathbf{H}_1 \,|\, \mathbf{H}_2 \,|\, \cdots \,|\, \mathbf{H}_h] \in \mathbb{R}^{n \times (h \cdot d_k)} = \mathbb{R}^{n \times d_{model}}$$

Step 6: Output projection

$$\mathbf{Z}^{attn} = \mathbf{H}\mathbf{W}^O \in \mathbb{R}^{n \times d_{model}}$$

where $\mathbf{W}^O \in \mathbb{R}^{d_{model} \times d_{model}}$.

12.3.2.2 Residual connection and layer normalization

Residual addition:

$$\mathbf{R}^{(1)} = \mathbf{X} + \mathbf{Z}^{attn} \in \mathbb{R}^{n \times d_{model}}$$

Layer normalization: For each position i, normalize across the d_{model} dimensions. Let $\mathbf{r}_i \in \mathbb{R}^{d_{model}}$ be row i of $\mathbf{R}^{(1)}$. Compute:

$$\mu_i = \frac{1}{d_{model}} \sum_{j=1}^{d_{model}} r_{ij}, \quad \sigma_i^2 = \frac{1}{d_{model}} \sum_{j=1}^{d_{model}} (r_{ij} - \mu_i)^2$$

$$\hat{r}_{ij} = \frac{r_{ij} - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}}$$

$$\tilde{x}_{ij} = \gamma_j^{(1)} \hat{r}_{ij} + \beta_j^{(1)}$$

where $\gamma^{(1)}, \beta^{(1)} \in \mathbb{R}^{d_{model}}$ are learnable parameters. This gives $\tilde{\mathbf{X}} \in \mathbb{R}^{n \times d_{model}}$.

12.3.2.3 Position-wise feed-forward network

$$\mathbf{F}^{(1)} = \tilde{\mathbf{X}} \mathbf{W}_1^T + \mathbf{1}_n \mathbf{b}_1^T \in \mathbb{R}^{n \times d_{ff}}$$

$$\mathbf{F}^{(2)} = \text{ReLU}(\mathbf{F}^{(1)}) \in \mathbb{R}^{n \times d_{ff}}$$

where $\text{ReLU}(x) = \max(0, x)$ applied element-wise.

$$\mathbf{Z}^{ffn} = \mathbf{F}^{(2)} \mathbf{W}_2^T + \mathbf{1}_n \mathbf{b}_2^T \in \mathbb{R}^{n \times d_{model}}$$

where $\mathbf{W}_1 \in \mathbb{R}^{d_{ff} \times d_{model}}$, $\mathbf{b}_1 \in \mathbb{R}^{d_{ff}}$, $\mathbf{W}_2 \in \mathbb{R}^{d_{model} \times d_{ff}}$, $\mathbf{b}_2 \in \mathbb{R}^{d_{model}}$, and $\mathbf{1}_n$ is the n-dimensional vector of ones.

12.3.2.4 Second residual and layer normalization

$$\mathbf{R}^{(2)} = \tilde{\mathbf{X}} + \mathbf{Z}^{ffn}$$

Apply layer normalization (same procedure as before) with parameters $\gamma^{(2)}, \beta^{(2)}$ to get encoder block output $\mathbf{X}^{(\ell)} \in \mathbb{R}^{n \times d_{model}}$.

12.3.2.5 Complete encoder

Apply N encoder blocks sequentially:

$$\mathbf{X}^{(\ell)} = \text{EncoderBlock}_\ell(\mathbf{X}^{(\ell-1)}) \quad \text{for } \ell = 1, \ldots, N$$

Final encoder output: $\mathbf{X}_{enc} = \mathbf{X}^{(N)} \in \mathbb{R}^{n \times d_{model}}$.

12.3.3 Decoder

Each decoder block applies: (1) masked self-attention, (2) residual + layer norm, (3) cross-attention, (4) residual + layer norm, (5) feed-forward network, (6) residual + layer norm.

12.3.3.1 Input to decoder

Given target sequence token indices $\mathbf{s} = [s_1, \ldots, s_m]$ where $s_i \in \{1, \ldots, V\}$. Embed and add positional encoding (analogous to encoder) to form $\mathbf{Y}^{(0)} \in \mathbb{R}^{m \times d_{model}}$.

12.3.3.2 Masked self-attention

Define causal mask $\mathbf{M} \in \mathbb{R}^{m \times m}$ where:

$$M_{ij} = \begin{cases} 0 & \text{if } j \leq i \\ -\infty & \text{if } j > i \end{cases}$$

Computation: For head k, compute queries, keys, values as before. Then:

$$\mathbf{S}_k = \frac{\mathbf{Q}_k \mathbf{K}_k^T}{\sqrt{d_k}} + \mathbf{M} \in \mathbb{R}^{m \times m}$$

$$\mathbf{A}_k = \text{softmax}(\mathbf{S}_k)$$

When $j > i$, entry $S_{k,ij} = -\infty$, so $A_{k,ij} = 0$ after softmax. This ensures position i only attends to positions $1, \ldots, i$.

Complete multi-head attention (concatenate, project) and apply residual + layer norm to get $\tilde{\mathbf{Y}}^{(1)} \in \mathbb{R}^{m \times d_{model}}$.

12.3.3.3 Cross-attention

For head k:

$$\mathbf{Q}_k = \tilde{\mathbf{Y}}^{(1)} \mathbf{W}_k^Q \in \mathbb{R}^{m \times d_k}$$

$$\mathbf{K}_k = \mathbf{X}_{enc} \mathbf{W}_k^K \in \mathbb{R}^{n \times d_k}$$

$$\mathbf{V}_k = \mathbf{X}_{enc} \mathbf{W}_k^V \in \mathbb{R}^{n \times d_k}$$

$$\mathbf{S}_k = \frac{\mathbf{Q}_k \mathbf{K}_k^T}{\sqrt{d_k}} \in \mathbb{R}^{m \times n}$$

$$\mathbf{A}_k = \text{softmax}(\mathbf{S}_k) \in \mathbb{R}^{m \times n}$$

$$\mathbf{H}_k = \mathbf{A}_k \mathbf{V}_k \in \mathbb{R}^{m \times d_k}$$

Note: attention matrix is $m \times n$ because decoder positions (rows) attend to encoder positions (columns).

Concatenate heads, project, and apply residual + layer norm to get $\tilde{\mathbf{Y}}^{(2)} \in \mathbb{R}^{m \times d_{model}}$.

12.3.3.4 FFN sub-layer

Apply the same FFN as in encoder (two linear layers with ReLU), then residual + layer norm to get decoder block output $\mathbf{Y}^{(\ell)} \in \mathbb{R}^{m \times d_{model}}$.

12.3.3.5 Complete decoder

Apply N decoder blocks sequentially:

$$\mathbf{Y}^{(\ell)} = \text{DecoderBlock}_\ell(\mathbf{Y}^{(\ell-1)}, \mathbf{X}_{enc}) \quad \text{for } \ell = 1, \ldots, N$$

Final decoder output: $\mathbf{Y}_{dec} = \mathbf{Y}^{(N)} \in \mathbb{R}^{m \times d_{model}}$.

12.3.4 Output projection and loss

Linear projection to vocabulary:

$$\mathbf{L} = \mathbf{Y}_{dec}\mathbf{W}_{out} + \mathbf{B}_{out} \in \mathbb{R}^{m \times V}$$

where $\mathbf{Y}_{dec} \in \mathbb{R}^{m \times d_{model}}$ is the final decoder output (one row per target position), $\mathbf{W}_{out} \in \mathbb{R}^{d_{model} \times V}$ projects from model dimension to vocabulary size, $\mathbf{B}_{out} \in \mathbb{R}^{m \times V}$ is the bias matrix (same bias vector $\mathbf{b}_{out} \in \mathbb{R}^V$ repeated for each row), and $\mathbf{L} \in \mathbb{R}^{m \times V}$ contains logits (unnormalized scores) for each token at each position.

Softmax over vocabulary:

For each position i, convert logits to probabilities:

$$P_{ij} = \frac{\exp(L_{ij})}{\sum_{k=1}^{V} \exp(L_{ik})}$$

where L_{ij} is the logit for token j at position i, the denominator sums over all V vocabulary tokens, and P_{ij} is the probability of token j being the correct next token at position i. Each row of $\mathbf{P} \in \mathbb{R}^{m \times V}$ sums to 1.

Cross-entropy loss:

Let $y_i \in \{1, \ldots, V\}$ be the index of the true target token at position i. The loss measures how well our predictions match the true targets:

$$\mathcal{L} = -\frac{1}{m} \sum_{i=1}^{m} \log P_{i,y_i}$$

where P_{i,y_i} is the predicted probability for the correct token at position i, $\log P_{i,y_i}$ is negative (since $0 < P_{i,y_i} \leq 1$), the negative sign makes $\mathcal{L}$ positive, and we average over all m positions. Lower loss means higher probability assigned to correct tokens.

12.4 Backward propagation

Training requires computing how much each parameter contributes to the loss. We do this by propagating gradients backward through the network using the chain rule: if z depends on y which depends on x, then $\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \cdot \frac{\partial y}{\partial x}$.

The backward pass mirrors the forward pass in reverse. We start with the gradient of loss with respect to the output, then work backward through each layer. At each layer, we compute two things: (1) gradients with respect to learnable parameters (for updating weights), and (2) gradients with respect to inputs (for continuing the backward pass to earlier layers).

12.4.1 The chain of gradients

The gradient flows backward through this path:

$$\mathcal{L} \leftarrow \mathbf{P} \leftarrow \mathbf{L} \leftarrow \mathbf{Y}_{dec} \leftarrow \text{DecoderBlocks} \leftarrow \mathbf{Y}^{(0)} \leftarrow \mathbf{E}_{target}$$

At each arrow, we apply the chain rule. The notation $\frac{\partial \mathcal{L}}{\partial \mathbf{X}}$ means "how much does the loss change when we change $\mathbf{X}$?"

12.4.2 Output layer: softmax and cross-entropy

We computed $\mathbf{P} = \text{softmax}(\mathbf{L})$ and $\mathcal{L} = -\frac{1}{m}\sum_{i=1}^{m} \log P_{i,y_i}$.

The gradient of loss with respect to logits combines both operations:

$$\frac{\partial \mathcal{L}}{\partial L_{ij}} = \frac{1}{m}\left(P_{ij} - \delta_{j,y_i}\right)$$

where $\delta_{j,y_i} = 1$ if $j = y_i$ (the correct token), otherwise 0.

Interpretation: For position i, the gradient is $\frac{1}{m}(\mathbf{p}_i - \mathbf{e}_{y_i})$ where $\mathbf{p}_i$ is the predicted probability vector and $\mathbf{e}_{y_i}$ is a one-hot vector with 1 at the correct token. The gradient points from the prediction toward the target.

12.4.3 Linear layers

For a linear layer $\mathbf{Y} = \mathbf{X}\mathbf{W} + \mathbf{b}$ (bias broadcast to all rows), given upstream gradient $\frac{\partial \mathcal{L}}{\partial \mathbf{Y}}$:

Weight gradient (for parameter update):

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}} = \mathbf{X}^T \frac{\partial \mathcal{L}}{\partial \mathbf{Y}}$$

Each entry $\frac{\partial \mathcal{L}}{\partial W_{ij}}$ accumulates contributions from all positions where input dimension i affected output dimension j.

Bias gradient (for parameter update):

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}} = \sum_{\text{rows}} \frac{\partial \mathcal{L}}{\partial \mathbf{Y}}$$

Sum over all positions since the same bias is added everywhere.

Input gradient (for continuing backward):

$$\frac{\partial \mathcal{L}}{\partial \mathbf{X}} = \frac{\partial \mathcal{L}}{\partial \mathbf{Y}} \mathbf{W}^T$$

This propagates the gradient back through the weight matrix.

12.4.4 Layer normalization

For layer normalization: $y_j = \gamma_j \hat{x}_j + \beta_j$ where $\hat{x}_j = \frac{x_j - \mu}{\sqrt{\sigma^2 + \epsilon}}$.

Parameter gradients:

$$\frac{\partial \mathcal{L}}{\partial \gamma_j} = \sum_{\text{positions}} \frac{\partial \mathcal{L}}{\partial y_j} \cdot \hat{x}_j$$

$$\frac{\partial \mathcal{L}}{\partial \beta_j} = \sum_{\text{positions}} \frac{\partial \mathcal{L}}{\partial y_j}$$

Input gradient: This is complex because each x_j affects the output through three paths: directly, through μ, and through σ^2. Let $\mathbf{g} = \gamma \odot \frac{\partial \mathcal{L}}{\partial \mathbf{y}}$ (upstream gradient scaled by γ):

$$\frac{\partial \mathcal{L}}{\partial x_j} = \frac{1}{\sqrt{\sigma^2 + \epsilon}} \left(g_j - \frac{1}{d} \sum_k g_k - \frac{\hat{x}_j}{d} \sum_k g_k \hat{x}_k \right)$$

The three terms account for: (1) direct contribution, (2) effect through mean, (3) effect through variance.

12.4.5 ReLU activation

For $y = \text{ReLU}(x) = \max(0, x)$:

$$\frac{\partial \mathcal{L}}{\partial x} = \begin{cases} \frac{\partial \mathcal{L}}{\partial y} & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$$

ReLU passes gradients through where it was active (input positive), and blocks gradients where it was inactive (input negative or zero). This is applied element-wise.

12.4.6 Attention mechanism

The attention computation $\mathbf{H}_k = \mathbf{A}_k \mathbf{V}_k$ where $\mathbf{A}_k = \text{softmax}(\mathbf{S}_k / \sqrt{d_k})$ and $\mathbf{S}_k = \mathbf{Q}_k \mathbf{K}_k^T$.

Step 1: Gradient through weighted sum ($\mathbf{H}_k = \mathbf{A}_k \mathbf{V}_k$)

$$\frac{\partial \mathcal{L}}{\partial \mathbf{A}_k} = \frac{\partial \mathcal{L}}{\partial \mathbf{H}_k} \mathbf{V}_k^T$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{V}_k} = \mathbf{A}_k^T \frac{\partial \mathcal{L}}{\partial \mathbf{H}_k}$$

Step 2: Gradient through softmax

For each row i, let $\mathbf{a} = \mathbf{A}_{k,i,:}$ be the attention weights and $\mathbf{s} = \mathbf{S}_{k,i,:}$ be the scores:

$$\frac{\partial \mathcal{L}}{\partial s_j} = a_j \left(\frac{\partial \mathcal{L}}{\partial a_j} - \sum_r a_r \frac{\partial \mathcal{L}}{\partial a_r} \right)$$

The subtracted term redistributes gradient to maintain the constraint that attention weights sum to 1.

Step 3: Gradient through scaling

$$\frac{\partial \mathcal{L}}{\partial S^{\text{unscaled}}_{k,ij}} = \frac{1}{\sqrt{d_k}} \frac{\partial \mathcal{L}}{\partial S_{k,ij}}$$

Step 4: Gradient through score computation ($\mathbf{S}_k = \mathbf{Q}_k \mathbf{K}_k^T$)

$$\frac{\partial \mathcal{L}}{\partial \mathbf{Q}_k} = \frac{\partial \mathcal{L}}{\partial \mathbf{S}_k} \mathbf{K}_k$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{K}_k} = \left(\frac{\partial \mathcal{L}}{\partial \mathbf{S}_k} \right)^T \mathbf{Q}_k$$

Step 5: Gradient through projections ($\mathbf{Q}_k = \mathbf{X}\mathbf{W}_k^Q$)

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_k^Q} = \mathbf{X}^T \frac{\partial \mathcal{L}}{\partial \mathbf{Q}_k}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{X}} \mathrel{+}= \frac{\partial \mathcal{L}}{\partial \mathbf{Q}_k} \left(\mathbf{W}_k^Q \right)^T$$

The += notation means we accumulate gradients because $\mathbf{X}$ is used to compute $\mathbf{Q}_k$, $\mathbf{K}_k$, and $\mathbf{V}_k$. We compute similar gradients for $\mathbf{W}_k^K$ and $\mathbf{W}_k^V$.

12.4.7 Residual connections

For $\mathbf{R} = \mathbf{X} + \mathbf{Z}$:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{X}} = \frac{\partial \mathcal{L}}{\partial \mathbf{R}}, \quad \frac{\partial \mathcal{L}}{\partial \mathbf{Z}} = \frac{\partial \mathcal{L}}{\partial \mathbf{R}}$$

The gradient flows unchanged to both branches. This is why residual connections help training: gradients can flow directly from later layers to earlier layers without passing through potentially problematic transformations. Even if gradients through $\mathbf{Z}$ vanish, gradients through the skip connection $\mathbf{X}$ remain intact.

12.4.8 Embedding layer

The embedding lookup $\mathbf{X}_{embed} = \mathbf{E}[\mathbf{t}, :]$ selects rows from the embedding matrix. The gradient updates only the rows that were selected:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{E}[j, :]} = \sum_{i:t_i=j} \frac{\partial \mathcal{L}}{\partial \mathbf{X}_{embed}[i, :]}$$

If token j appears at positions 2 and 5 in the sequence, we sum the gradients from both positions and apply that update to row j of the embedding matrix. Tokens not present in the current batch receive zero gradient.

12.5 Parameter updates

We use Adam optimizer with:

- Learning rate: α (with warmup schedule)
- Exponential decay rates: $\beta_1 = 0.9$, $\beta_2 = 0.98$
- Stability constant: $\epsilon = 10^{-9}$

For each parameter θ with gradient $g_t = \frac{\partial \mathcal{L}}{\partial \theta}$ at step t:

Update first moment:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

Update second moment:

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

Bias correction:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

Parameter update:

$$\theta_t = \theta_{t-1} - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

Learning rate schedule with warmup:

$$\alpha_t = d_{model}^{-0.5} \cdot \min(t^{-0.5}, t \cdot \text{warmup\_steps}^{-1.5})$$

with warmup_steps = 4000. This increases learning rate linearly during warmup, then decays proportional to $t^{-0.5}$.

12.6 Decoder-only architecture (GPT)

For decoder-only models:

Architecture:

1. Embedding + positional encoding
2. N decoder blocks, each with:
   - Masked multi-head self-attention
   - Residual + layer normalization
   - Position-wise FFN
   - Residual + layer normalization
3. Output projection to vocabulary

Key differences from encoder-decoder:

- No encoder component
- No cross-attention sub-layer in decoder blocks
- Only masked self-attention (causal masking ensures autoregressive generation)

Forward and backward propagation follow the masked self-attention and FFN components described above.

12.7 Encoder-only architecture (BERT)

For encoder-only models like BERT (Devlin et al. 2018):

Architecture:

1. Embedding + positional encoding
2. N encoder blocks (bidirectional self-attention + FFN)
3. Task-specific head (e.g., linear classifier on [CLS] token)

Key differences from encoder-decoder:

- No decoder component
- No causal masking (bidirectional attention)
- Task-specific output layer instead of vocabulary projection

Forward and backward propagation follow the encoder derivation above.

12.8 Computational complexity

Time complexity per layer:

- Self-attention: $O(n^2 d_{model})$ for computing attention scores
- Cross-attention: $O(mnd_{model})$ where m is decoder length, n is encoder length
- FFN: $O(nd_{model}d_{ff})$
- Total per sample: $O(n^2 d_{model}N)$ for encoder/decoder

Space complexity:

- Attention matrices: $O(n^2 hN)$ for storing attention weights
- Activations: $O(nd_{model}N)$ for intermediate states
- Parameters: $O(d_{model}^2 N + Vd_{model})$ for weights and embeddings

Parameter count (base transformer):

- Embeddings: $V \times d_{model} = 50000 \times 512 \approx 25$M
- Per encoder block: $4d_{model}^2 + 2d_{model}d_{ff} + 4d_{model} \approx 3$M
- $N = 6$ blocks (encoder + decoder): 36M
- Output projection: $d_{model} \times V \approx 25$M
- **Total:** ≈ 86M parameters

For comparison, GPT-3 (Brown et al. 2020) has 175 billion parameters using $d_{model} = 12288$, $d_{ff} = 49152$, $h = 96$, $N = 96$.

12.9 Implementation notes

A complete implementation requires:

Initialization:

- Weight matrices: Xavier/Glorot uniform initialization
- Biases: initialize to zero
- Layer norm parameters: $\gamma = 1$, $\beta = 0$

Regularization:

- Dropout with rate 0.1 after attention weights and FFN activations
- Label smoothing with value 0.1 for cross-entropy loss

Batching:

- Process multiple sequences in parallel
- Pad sequences to maximum length in batch
- Create padding mask: set attention scores to $-\infty$ for padding positions

Training stability:

- Gradient clipping: clip global norm to prevent explosion
- Mixed precision training: use FP16 for forward/backward, FP32 for parameter updates

This completes the mathematical specification of the transformer architecture with forward and backward propagation.

Chapter 13

Training objectives

Learning objectives

After completing this chapter, you will be able to:

- Define the causal language modeling objective and compute perplexity
- Explain masked language modeling and why BERT uses it
- Describe instruction tuning and its role in making models follow directions
- Understand RLHF: reward models, policy optimization, and KL constraints
- Compare different training objectives and their effects on model behavior

The transformer architecture defines how information flows through the network. But to learn useful weights, we need a training objective: a mathematical function that measures how well the model performs and provides gradients for improvement. This chapter covers the primary training objectives used for transformer models.

13.1 Language modeling

The most fundamental objective for decoder-style transformers is language modeling: predicting the next token given previous tokens.

13.1.1 The objective

Given a sequence of tokens $x_1, x_2, \ldots, x_T$, the model learns to predict each token from its predecessors:

$$\mathcal{L}_{LM} = -\sum_{t=1}^{T} \log P(x_t | x_1, \ldots, x_{t-1})$$

At each position t, the model outputs a probability distribution over the vocabulary. We measure how much probability mass lands on the correct token x_t using the negative log-likelihood. Lower loss means the model assigns higher probability to the true next tokens.

13.1.2 Why this works

Language modeling is self-supervised: the training signal comes from the data itself, requiring no manual labels. Given the sentence "The cat sat on the", the model learns that "mat" is a likely continuation while "elephant" is not. Through millions of such predictions, the model learns syntax, semantics, facts, and reasoning patterns.

The causal masking in decoder attention ensures the model cannot cheat by looking at future tokens. Position t can only attend to positions $1, \ldots, t-1$, so the prediction $P(x_t|x_1, \ldots, x_{t-1})$ depends only on legitimate context.

13.1.3 Perplexity

We often report perplexity instead of raw loss:

$$\text{PPL} = \exp\left(\frac{1}{T}\sum_{t=1}^{T} -\log P(x_t|x_1, \ldots, x_{t-1})\right) = \exp(\mathcal{L}_{LM}/T)$$

Perplexity has an intuitive interpretation: it measures how "surprised" the model is by the data. A perplexity of 10 means the model is, on average, as uncertain as if it were choosing uniformly among 10 options at each position. Lower perplexity indicates better predictions.

13.2 Masked language modeling

For encoder-style transformers like BERT (Devlin et al. 2018), we use masked language modeling (MLM): predicting randomly masked tokens from bidirectional context.

13.2.1 The objective

Given a sequence, we randomly select 15% of tokens for prediction. For selected positions, we replace the token with [MASK] 80% of the time, a random token 10% of the time, or keep it unchanged 10% of the time. The model predicts the original tokens:

$$\mathcal{L}_{MLM} = -\sum_{t \in \mathcal{M}} \log P(x_t|\mathbf{x}_{\setminus t})$$

where $\mathcal{M}$ is the set of masked positions and $\mathbf{x}_{\setminus t}$ denotes all tokens except position t (which the model sees as [MASK] or corrupted).

13.2.2 Bidirectional context

Unlike causal language modeling, MLM uses bidirectional attention. When predicting the masked token in "The [MASK] sat on the mat", the model can use both "The" (left context) and "sat on the mat" (right context). This bidirectional understanding is valuable for tasks like classification and question answering where we have complete input sequences.

13.2.3 The masking strategy

The 80/10/10 split serves specific purposes. Using [MASK] 80% of the time teaches the model to predict from context. Using random tokens 10% of the time prevents the model from assuming [MASK] always means "predict here". Keeping original tokens 10% of the time teaches the model that unmasked positions might still need prediction, which helps during fine-tuning when there are no [MASK] tokens.

13.3 Next sentence prediction

BERT introduced a secondary objective: predicting whether two sentences are consecutive in the original text.

13.3.1 The objective

Given sentence pair (A, B), predict whether B actually followed A in the corpus:

$$\mathcal{L}_{NSP} = -\log P(\text{IsNext}|[\text{CLS}], A, [\text{SEP}], B)$$

The model uses the [CLS] token representation to make a binary classification. 50% of training pairs are true consecutive sentences, 50% are random pairs.

13.3.2 Limitations

Later research showed NSP provides limited benefit and can even hurt performance. The task is too easy: distinguishing random sentences often reduces to topic detection rather than understanding coherence. Models like RoBERTa (Liu et al. 2019) dropped NSP entirely and achieved better results with just MLM.

13.4 Causal language modeling at scale

GPT-style models (Radford et al. 2018) use pure causal language modeling but at massive scale. The key insight from GPT-2 (Radford et al. 2019) and GPT-3 (Brown et al. 2020) is that a sufficiently large language model trained on diverse text becomes a general-purpose system.

13.4.1 Emergent capabilities

As models scale, they develop capabilities not explicitly trained:

- **Zero-shot learning**: Performing tasks from instructions alone
- **Few-shot learning**: Learning new tasks from a handful of examples in context
- **Chain-of-thought reasoning**: Breaking complex problems into steps

These emerge from the language modeling objective because predicting text requires understanding the underlying concepts, relationships, and reasoning patterns.

13.4.2 The training data

Modern language models train on web-scale corpora: hundreds of billions of tokens from books, websites, code repositories, and other sources. Data quality and diversity matter enormously. Filtering for quality, deduplicating, and balancing domains all improve downstream performance.

13.5 Instruction tuning

A language model trained purely on next-token prediction learns to complete text, but completing text is not the same as following instructions. Given the prompt “What is the capital of France?”, a pretrained model might continue with “is a common geography question” or “The capital of Germany is Berlin” because these are plausible text continuations. The model has no notion that it should answer the question.

Instruction tuning bridges this gap. We fine-tune the pretrained model on examples of instructions paired with appropriate responses, teaching it to interpret prompts as requests and generate helpful outputs.

13.5.1 The data format

Each training example consists of an instruction (or prompt) and a response:

> **Instruction:** Summarize the following article in three sentences. [Article text…]
>
> **Response:** The article discusses… Key findings include… The authors conclude…

Instructions vary widely: "Translate this to Spanish", "Write a Python function that sorts a list", "Explain quantum entanglement to a 10-year-old", "What are the pros and cons of solar energy?". The diversity matters. A model trained only on translation instructions would not learn to answer questions. Broad coverage across task types produces a general-purpose assistant.

The response demonstrates the desired behavior. For a summarization instruction, the response is a good summary. For a coding instruction, the response is working code. The model learns by example what constitutes an appropriate response to each type of instruction.

13.5.2 The objective

Instruction tuning uses the same cross-entropy loss as language modeling, but with a crucial modification: we only compute loss on the response tokens.

Given an instruction $x = (x_1, \ldots, x_n)$ and response $r = (r_1, \ldots, r_m)$, we concatenate them and feed the sequence through the model. The loss is:

$$\mathcal{L}_{IT} = -\sum_{t=1}^{m} \log P(r_t | x_1, \ldots, x_n, r_1, \ldots, r_{t-1})$$

Notice the sum runs only over response positions $t = 1, \ldots, m$. We do not penalize the model for its predictions on instruction tokens. Why? The instruction is given by the user at inference time; the model does not need to generate it. We care only that the model produces good responses given instructions, not that it can predict instruction text.

Mathematically, this is implemented by masking the loss. We compute predictions for all positions but multiply losses by zero for instruction tokens:

$$\mathcal{L}_{IT} = -\sum_{t=1}^{n+m} \text{mask}_t \cdot \log P(\text{token}_t | \text{token}_1, \ldots, \text{token}_{t-1})$$

where $\text{mask}_t = 0$ for instruction positions and $\text{mask}_t = 1$ for response positions.

13.5.3 Where does the data come from?

Instruction-tuning datasets are constructed through several approaches:

Human-written examples: Contractors or researchers write instructions and high-quality responses. This produces reliable data but is expensive and slow. Thousands of examples might cost tens of thousands of dollars.

Crowdsourcing: Platforms like Amazon Mechanical Turk collect instructions and responses from many workers. Quality varies, requiring filtering and validation. Larger scale is possible but noise increases.

Existing datasets reformatted: Many NLP datasets can be converted to instruction format. A sentiment classification dataset becomes "Classify the sentiment of this review as positive or negative: [review]" with the label as the response. A translation corpus becomes "Translate to French: [English text]". This provides large-scale data cheaply but may not cover conversational or open-ended instructions well.

Synthetic data from larger models: A powerful model generates responses to instructions, and these are used to train smaller models. This is called distillation. The student model learns to imitate the teacher's behavior. Quality depends on the teacher model, and there are legal and ethical considerations around using model outputs as training data.

User interactions: Deployed systems can collect real user instructions (with consent). This captures what users actually want, which may differ from what dataset creators imagine. Responses may come from human operators or be filtered from model outputs.

In practice, instruction-tuning datasets combine multiple sources. A dataset might include 10,000 human-written examples for quality, 100,000 reformatted NLP examples for coverage, and 50,000 synthetic examples for scale.

13.5.4 What changes during instruction tuning?

Pretraining produces a model that assigns probability to text. Instruction tuning reshapes this distribution. Before tuning, $P(\text{"Paris"}|\text{"What is the capital of France?"})$ might be low because the model expects the text to continue as a document, not as an answer. After tuning, this probability increases because the model has seen thousands of question-answer pairs.

The model learns several things:

1. **Response format**: Answers should be direct, not continuations of the question
2. **Task recognition**: Different instruction patterns (translate, summarize, explain) require different response types
3. **Helpfulness**: Responses should address what the user asked, not tangentially related content
4. **Refusal**: Some instructions should be declined (harmful requests, impossible tasks)

The weights change throughout the network, but the changes are typically small relative to pretraining. We start from a capable language model and nudge it toward instruction-following behavior. This is why instruction tuning requires far less data than pretraining: we are refining existing capabilities, not building them from scratch.

13.5.5 Practical considerations

Learning rate: Instruction tuning uses smaller learning rates than pretraining, often 10^{-5} to 10^{-6} compared to 10^{-4} for pretraining. Large learning rates would destroy the knowledge acquired during pretraining.

Epochs: We typically train for 1-3 epochs over the instruction data. More epochs risk overfitting to the specific phrasing of training instructions.

Parameter-efficient fine-tuning: Instead of updating all weights, methods like LoRA (Low-Rank Adaptation) freeze most parameters and train small adapter modules. This reduces memory requirements and preserves more of the pretrained knowledge. The adapters learn the instruction-following behavior while the base model remains unchanged.

Chat format: Many instruction-tuned models use a specific format with special tokens marking roles:

> `<|user|>` What is the capital of France? `<|assistant|>` The capital of France is Paris.

The model learns to generate text following `<|assistant|>` given context containing `<|user|>` messages. Multi-turn conversations extend this with alternating user and assistant blocks.

13.6 Fine-tuning

Fine-tuning adapts a pretrained model to a specific task, domain, or behavior. Rather than training from scratch, we start with weights that already encode general language understanding and adjust them for our particular needs. This is one of the most practically important techniques in modern NLP.

13.6.1 Why fine-tuning works

A model pretrained on billions of tokens learns far more than next-token prediction. It learns:

- **Syntax**: How words combine into grammatical sentences
- **Semantics**: What words mean and how meanings compose
- **World knowledge**: Facts about entities, relationships, common sense
- **Reasoning patterns**: How to draw inferences, follow logic

- **Discourse structure**: How paragraphs and arguments flow

These capabilities transfer to new tasks. A model that understands English grammar does not need to relearn grammar for sentiment classification. A model that knows facts about the world can answer questions about those facts with minimal additional training.

The mathematics of transfer learning: pretraining finds weights θ_{pre} that minimize loss on a large, general corpus. These weights define a point in parameter space. Fine-tuning starts from θ_{pre} and moves to nearby weights θ_{fine} that minimize loss on a smaller, task-specific dataset. Because θ_{pre} already encodes useful structure, the optimization problem is easier: we are refining, not building from scratch.

Empirically, fine-tuning requires 100-10,000x less data than pretraining for equivalent task performance. A model pretrained on 1 trillion tokens might fine-tune effectively on 10,000 examples. This dramatic efficiency gain is why fine-tuning dominates practical applications.

13.6.2 The fine-tuning objective

Fine-tuning uses the same loss function as pretraining, just on different data:

$$\mathcal{L}_{fine} = -\sum_{(x,y)\in\mathcal{D}_{task}} \log P_\theta(y|x)$$

For classification tasks, y is a class label and $P_\theta(y|x)$ comes from a classification head added to the model. For generation tasks, y is a target sequence and we use the standard language modeling loss over output tokens.

The key differences from pretraining:

Smaller learning rate: We use learning rates 10-100x smaller than pretraining, typically 10^{-5} to 10^{-6}. Large learning rates would destroy the pretrained knowledge. We want gentle updates that preserve most of what was learned while adjusting for the new task.

Fewer steps: Fine-tuning runs for thousands of steps, not millions. One to three passes over the fine-tuning data is typical. More risks overfitting to the small dataset.

Task-specific data: The fine-tuning dataset is much smaller but more focused. For sentiment classification, we might have 10,000 movie reviews with labels. For medical question answering, we might have 5,000 question-answer pairs from clinical sources.

13.6.3 Full fine-tuning

In full fine-tuning, we update the exact same weight matrices that were learned during pretraining. The model architecture does not change. We take the pretrained weights and adjust their numerical values.

Specifically, these are the weights being updated:

- **Embedding matrix** $\mathbf{W}_E \in \mathbb{R}^{V\times d}$: Maps tokens to vectors
- **Attention weights** in each layer: $\mathbf{W}_Q, \mathbf{W}_K, \mathbf{W}_V \in \mathbb{R}^{d\times d}$ and output projection $\mathbf{W}_O$
- **MLP weights** in each layer: $\mathbf{W}_1 \in \mathbb{R}^{d\times 4d}$, $\mathbf{W}_2 \in \mathbb{R}^{4d\times d}$
- **Layer normalization parameters**: Scale and shift vectors
- **Output projection** $\mathbf{W}_{out} \in \mathbb{R}^{d\times V}$: Maps back to vocabulary (often tied with $\mathbf{W}_E$)

For a transformer with L layers, this is tens of billions of individual numbers, each one adjustable during fine-tuning.

The update rule is standard gradient descent:

$$\theta_{fine} = \theta_{pre} - \eta\nabla_\theta\mathcal{L}_{fine}(\theta_{pre})$$

repeated for multiple steps with learning rate η. We start from the pretrained values θ_{pre} and nudge each weight in the direction that reduces the task-specific loss.

What changes? After fine-tuning, the embedding for "plaintiff" might shift slightly in the embedding space. The attention weights might learn to attend more strongly to legal terminology. The MLP might adjust how it processes formal language. Every weight can change, and millions of small changes accumulate into different model behavior.

What stays the same? The architecture: number of layers, attention heads, hidden dimensions, activation functions. We are not adding or removing anything, just adjusting the numbers.

Advantages: Maximum flexibility to adapt to the new task. If the task requires very different representations than pretraining, full fine-tuning can make large changes.

Disadvantages: Requires storing a complete copy of the model for each task. A 70B parameter model takes 140GB in 16-bit precision. Ten tasks means 1.4TB of storage. Also, full fine-tuning can overfit quickly on small datasets since all parameters are free to change.

13.6.4 Catastrophic forgetting

When we fine-tune on task A, the model may lose capabilities it had before fine-tuning. This is catastrophic forgetting: the new gradients overwrite information stored in the weights.

Consider a model pretrained on general text, then fine-tuned on legal documents. After fine-tuning, it might excel at legal language but struggle with casual conversation or code. The legal gradients pushed weights away from configurations that supported those other capabilities.

Mathematically, the pretrained weights θ_{pre} lie in a region of parameter space that performs well on many tasks. Fine-tuning moves to θ_{fine}, optimized for the specific task but potentially outside the good region for other tasks.

Mitigation strategies:

Lower learning rate: Smaller updates stay closer to θ_{pre}, preserving more general capability.

Early stopping: Stop fine-tuning before the model overfits to the task data. Monitor performance on held-out general benchmarks.

Regularization toward pretrained weights: Add a penalty $\lambda||\theta - \theta_{pre}||^2$ to the loss. This explicitly discourages moving far from the pretrained configuration.

Replay: Mix task-specific data with samples from the pretraining distribution. The model continues seeing general text while learning the specific task.

Parameter-efficient methods: Update only a small subset of parameters, leaving most frozen. This is the most effective approach and deserves detailed discussion.

13.6.5 Parameter-efficient fine-tuning

Instead of updating all parameters, we can freeze most of the pretrained model and train only a small number of additional or selected parameters. This family of techniques is called parameter-efficient fine-tuning (PEFT).

13.6.5.1 LoRA: Low-Rank Adaptation

LoRA is the most widely used PEFT method. The key insight: the weight updates during fine-tuning have low rank. We do not need full-rank updates to adapt a model.

The core idea: Instead of modifying the pretrained weight matrix $\mathbf{W}$ directly, we add a small correction term to it. This correction is represented as the product of two new, smaller matrices that we create from scratch.

Consider a pretrained weight matrix $\mathbf{W} \in \mathbb{R}^{d \times k}$. In full fine-tuning, we would update $\mathbf{W}$ to $\mathbf{W} + \Delta\mathbf{W}$, where $\Delta\mathbf{W}$ is whatever change the gradients dictate. LoRA instead constrains $\Delta\mathbf{W}$ to be low-rank by factorizing it:

$$\Delta\mathbf{W} = \mathbf{BA}$$

What are A and B? These are two brand-new matrices that do not exist in the pretrained model. We create them specifically for fine-tuning:

- $\mathbf{A} \in \mathbb{R}^{r \times k}$: A "down-projection" matrix with r rows and k columns
- $\mathbf{B} \in \mathbb{R}^{d \times r}$: An "up-projection" matrix with d rows and r columns
- r is the "rank," a small number like 8, 16, or 64

The product $\mathbf{BA}$ has shape $d \times k$, matching the original weight matrix $\mathbf{W}$. But because we go through the bottleneck dimension r, the product can only represent matrices of rank at most r. This is the "low-rank" constraint.

Visualizing the dimensions: Suppose $\mathbf{W}$ is a 4096×4096 attention weight matrix and we use rank $r = 16$:

Matrix	Shape	Parameters	Status
Original $\mathbf{W}$	4096×4096	16.7 million	Frozen
$\mathbf{A}$	16×4096	65,536	Trainable
$\mathbf{B}$	4096×16	65,536	Trainable
$\mathbf{BA}$	4096×4096	(computed, not stored)	Low-rank update

The input $\mathbf{x}$ has dimension $k = 4096$. The computation flows:

1. $\mathbf{Ax}$: Project from 4096 dimensions down to 16 dimensions
2. $\mathbf{B}(\mathbf{Ax})$: Project from 16 dimensions back up to 4096 dimensions

The bottleneck at 16 dimensions limits what transformations are possible, but this turns out to be enough for task adaptation.

The forward pass becomes:

$$\mathbf{h} = \mathbf{Wx} + \mathbf{BAx}$$

The pretrained weights $\mathbf{W}$ are frozen and never updated. Only $\mathbf{A}$ and $\mathbf{B}$ receive gradients and change during training. The original model is untouched; we are learning a small correction on top of it.

Initialization:

- $\mathbf{A}$ is initialized with small random values (Gaussian, variance $1/r$)
- $\mathbf{B}$ is initialized to all zeros

Because $\mathbf{B} = 0$ initially, the product $\mathbf{BA} = 0$, so $\mathbf{h} = \mathbf{Wx}$ at the start. The model begins exactly at pretrained behavior. As training proceeds, $\mathbf{B}$ grows away from zero, and the LoRA correction gradually takes effect.

Parameter count: For a weight matrix of size $d \times k$, full fine-tuning has dk trainable parameters. LoRA with rank r has $r \times k + d \times r = r(d + k)$ trainable parameters. With $d = k = 4096$ and $r = 16$:

- Full: $4096 \times 4096 = 16.7M$ parameters per matrix
- LoRA: $16 \times 4096 + 4096 \times 16 = 131K$ parameters per matrix

A 127x reduction per matrix. Across all attention matrices in a large model, LoRA typically adds 0.1-1% of the base model's parameters.

Scaling factor: In practice, the LoRA update is scaled:

$$\mathbf{h} = \mathbf{W}\mathbf{x} + \frac{\alpha}{r}\mathbf{B}\mathbf{A}\mathbf{x}$$

where α is a hyperparameter (often set equal to r so the factor is 1, or tuned separately). This scaling helps with learning rate tuning across different ranks.

Which layers get LoRA? We choose which weight matrices in the model to augment with LoRA. Common choices:

- $\mathbf{W}_Q$ and $\mathbf{W}_V$ (query and value projections in attention): Most common, good results
- Adding $\mathbf{W}_K$ and $\mathbf{W}_O$ (key and output projections): More capacity
- Adding MLP weights: Even more capacity, but diminishing returns

Each matrix we apply LoRA to gets its own pair of $\mathbf{A}$ and $\mathbf{B}$ matrices. If we apply LoRA to $\mathbf{W}_Q$ and $\mathbf{W}_V$ in each of 32 layers, we create $32 \times 2 = 64$ pairs of LoRA matrices.

After training: We can merge the LoRA weights back into the base model:

$$\mathbf{W}_{merged} = \mathbf{W} + \mathbf{B}\mathbf{A}$$

Now we have a single weight matrix with no inference overhead. The model runs at normal speed with the task-specific behavior baked in. Or we can keep them separate, allowing us to swap different LoRA adaptations in and out of the same base model.

Why low-rank works: Fine-tuning for a specific task does not require changing everything the model knows. It requires adjusting how existing knowledge is accessed and combined. These adjustments lie in a low-dimensional subspace of the full parameter space.

Think of it this way: the pretrained model already knows about language, facts, and reasoning. To adapt it to legal documents, we do not need to relearn English. We need to adjust attention patterns to focus on legal terminology and shift how the model weighs certain features. These adjustments are relatively simple transformations of what already exists, hence low-rank.

Empirically, rank 8-64 suffices for most tasks. This suggests the "task-specific adjustment space" is indeed low-dimensional, perhaps only a few dozen independent directions in the million-dimensional parameter space.

13.6.5.2 Adapters

Adapters insert small trainable modules between frozen layers. A typical adapter has:

1. Down-projection: $\mathbf{W}_{down} \in \mathbb{R}^{d \times r}$ reducing dimension from d to r
2. Nonlinearity: ReLU or GELU
3. Up-projection: $\mathbf{W}_{up} \in \mathbb{R}^{r \times d}$ restoring dimension
4. Residual connection: add the adapter output to the input

$$\text{Adapter}(\mathbf{x}) = \mathbf{x} + \mathbf{W}_{up} \cdot \text{ReLU}(\mathbf{W}_{down} \cdot \mathbf{x})$$

Adapters are placed after attention and/or MLP sublayers. The residual connection means that with zero-initialized up-projection, the adapter initially has no effect.

Comparison to LoRA: Adapters add sequential computation (extra matrix multiplies in the forward pass). LoRA modifies existing computations (the $\mathbf{W} + \mathbf{B}\mathbf{A}$ can be merged at inference). LoRA is generally preferred for its inference efficiency: after training, merge $\Delta\mathbf{W}$ into $\mathbf{W}$ and the model runs at normal speed. Adapters always add overhead.

13.6.5.3 Prefix tuning

Prefix tuning prepends learnable "virtual tokens" to the input. Instead of modifying weights, we modify the input:

$$\text{Input} = [\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_m, \mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n]$$

where $\mathbf{p}_i$ are trainable prefix embeddings and $\mathbf{x}_j$ are the actual input tokens. The prefix embeddings are learned per-task while all model weights remain frozen.

The prefix acts as a task-specific context that steers the model's behavior. For summarization, the prefix might implicitly encode "summarize the following." For translation, it might encode "translate to French."

Soft prompts vs. hard prompts: Hard prompts are actual text tokens ("Summarize:"). Soft prompts are continuous vectors that do not correspond to real tokens. Soft prompts can be optimized directly, potentially finding better "prompts" than any expressible in natural language.

Parameter count: With prefix length m and embedding dimension d, prefix tuning adds $m \times d$ parameters (plus some overhead for deep prefix tuning that adds prefixes at every layer). This is very few parameters, but the method is less expressive than LoRA for some tasks.

13.6.6 Choosing a fine-tuning approach

When should you use each approach?

Full fine-tuning when: - You have abundant task-specific data (100K+ examples) - The task is very different from pretraining (e.g., a new language) - You only need one task and can afford the storage - Maximum performance matters more than efficiency

LoRA when: - You have moderate data (1K-100K examples) - You need multiple task-specific versions of the same base model - You want to preserve general capabilities while adding task skills - Inference efficiency matters (merged weights have no overhead)

Adapters when: - You want modular, swappable task capabilities - You are experimenting with many tasks and want easy comparison - The overhead of adapter forward passes is acceptable

Prefix tuning when: - You have very little data (100-1000 examples) - The task is a variation of something the model already does well - You want the smallest possible parameter footprint

No fine-tuning (prompting only) when: - You have essentially no task-specific data - The base model already performs adequately with good prompts - You need zero additional training or storage

13.6.7 The fine-tuning landscape

Modern LLM deployment typically involves multiple stages of fine-tuning:

1. **Pretraining**: Massive compute, general text, language modeling objective
2. **Continued pretraining** (optional): More compute on domain-specific text (medical, legal, code)
3. **Instruction tuning**: Moderate compute, instruction-following data
4. **Task-specific fine-tuning**: Smaller compute, specific application data
5. **RLHF/DPO**: Alignment with human preferences

Each stage can use full fine-tuning or PEFT methods. A common pattern: - Full fine-tuning for instruction tuning (significant behavior change needed) - LoRA for task-specific adaptation (preserve instruction-following while adding task skill)

The pretrained model is a foundation. Each fine-tuning stage builds on it, specializing capabilities while (hopefully) preserving general competence. Understanding this landscape helps practitioners choose where and how to intervene for their specific needs.

13.7 Reinforcement learning from human feedback

RLHF further aligns models with human preferences using reinforcement learning (Ouyang et al. 2022). The core problem is this: language modeling teaches a model to predict text, but predicting text is not the same as being helpful. A model trained purely on web text will happily generate toxic content, confidently state falsehoods, or ignore the user's actual intent. RLHF incorporates human judgment about what makes a response good, beyond statistical likelihood.

The process has three stages, each with different models being trained:

1. **Collect preferences**: Sample responses from the current LLM, have humans compare them. No training happens here.
2. **Train reward model**: Create a separate model (initialized from the same pretrained weights but with a scalar output head). Train this reward model on the preference data. The LLM weights are frozen during this phase.
3. **Optimize policy**: Now train the LLM to produce high-reward responses. The reward model weights are frozen; only the LLM weights update.

The key point: the reward model and the language model are separate networks. We never train them simultaneously. The reward model learns to score responses; then, with the reward model fixed, we update the language model to generate better responses according to those scores.

13.7.1 Collecting preference data

We start with a pretrained and instruction-tuned language model. Given a prompt x, we sample multiple responses $y_1, y_2, \ldots, y_k$ from the model. Human annotators then compare pairs of responses and indicate which is better. For a prompt like "Explain quantum entanglement to a child", one response might use clear analogies while another might use jargon. The annotator marks the clearer one as preferred.

This comparison format is crucial. Asking humans to score responses on an absolute scale (1-10) produces inconsistent ratings. Different annotators calibrate differently, and even the same annotator varies over time. But comparisons are robust: given two responses side by side, humans reliably identify which is better, even when they cannot articulate exactly why.

The result is a dataset of triples (x, y_w, y_l): prompt, winning response, losing response. Thousands of such comparisons capture nuanced human preferences about helpfulness, accuracy, safety, and style.

13.7.2 The reward model

We train a reward model $r_\phi(x, y)$ that takes a prompt and response and outputs a scalar score. This is a separate network from the language model we ultimately want to improve. Architecturally, the reward model is typically a transformer initialized from the same pretrained weights as the language model, but we remove the language modeling head (which outputs vocabulary probabilities) and add a linear layer that outputs a single number. The final token's representation passes through this layer to produce the reward.

During this phase, we only update the reward model parameters ϕ. The language model that generated the responses remains frozen. We are not yet improving the language model; we are building a tool (the reward model) that will guide improvements in the next phase.

The reward model learns from preference comparisons using the Bradley-Terry model. This statistical model, developed in the 1950s for ranking chess players and sports teams, provides a principled way to convert pairwise comparisons into numerical scores.

13.7.3 The Bradley-Terry model

The core idea is simple: each item has an underlying "strength" and the probability of one item beating another depends on their relative strengths. In the original formulation, if item i has strength $p_i > 0$ and

item j has strength $p_j > 0$, then the probability that i beats j is:

$$P(i \succ j) = \frac{p_i}{p_i + p_j}$$

This formula has intuitive properties. If $p_i = p_j$, the probability is $\frac{1}{2}$: equally matched items have equal chances. If $p_i = 2p_j$, then $P(i \succ j) = \frac{2}{3}$: item i is twice as "strong" and wins two-thirds of the time. As $p_i \to \infty$ relative to p_j, the probability approaches 1.

We can reparametrize by taking logarithms. Let $r_i = \log p_i$. Then:

$$P(i \succ j) = \frac{p_i}{p_i + p_j} = \frac{e^{r_i}}{e^{r_i} + e^{r_j}} = \frac{1}{1 + e^{-(r_i - r_j)}} = \sigma(r_i - r_j)$$

The sigmoid function $\sigma(z) = \frac{1}{1+e^{-z}}$ emerges naturally. The probability of winning depends only on the difference in log-strengths $r_i - r_j$. This is why we call these log-strengths "rewards" in RLHF: they are scores on a scale where differences determine win probabilities.

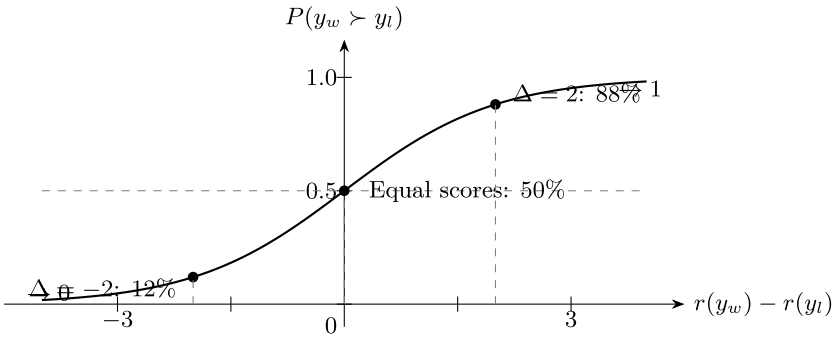

Figure 13.1: The Bradley-Terry model maps score differences to win probabilities via the sigmoid function. When scores are equal (difference = 0), the win probability is 50%. A score difference of 2 gives roughly 88% win probability.

Figure 13.1 shows this relationship. The sigmoid curve transforms any score difference into a probability between 0 and 1. The curve is steepest around zero, meaning small score differences produce noticeable probability changes. Far from zero, the curve flattens: whether one response scores 10 points higher or 100 points higher, it will win with near-certainty either way.

A concrete example: Suppose for a given prompt, response A has reward $r_A = 1.5$ and response B has reward $r_B = -0.5$. The difference is $r_A - r_B = 2.0$. The Bradley-Terry model predicts:

$$P(A \succ B) = \sigma(2.0) = \frac{1}{1 + e^{-2}} \approx 0.88$$

Response A wins 88% of the time. If we collected many comparisons between these responses, we would expect A to be preferred in roughly 88 out of 100 comparisons.

13.7.4 Training the reward model

For RLHF, we apply Bradley-Terry to response pairs. If response y_w is preferred over y_l for prompt x, the model should satisfy:

$$P(y_w \succ y_l|x) = \sigma(r_\phi(x, y_w) - r_\phi(x, y_l))$$

We train by maximizing the log-likelihood of observed preferences:

$$\mathcal{L}_{RM} = -\mathbb{E}_{(x,y_w,y_l)\sim\mathcal{D}} \left[\log \sigma(r_\phi(x, y_w) - r_\phi(x, y_l))\right]$$

Expanding the logarithm of the sigmoid:

$$\log \sigma(z) = \log \frac{1}{1 + e^{-z}} = -\log(1 + e^{-z})$$

When z is large and positive (winner has much higher reward), $e^{-z} \approx 0$, so $\log \sigma(z) \approx 0$: low loss. When z is negative (loser has higher reward), e^{-z} is large, and the loss grows. The gradient pushes the model to increase $r_\phi(x, y_w)$ and decrease $r_\phi(x, y_l)$.

Note that only the difference $r_\phi(x, y_w) - r_\phi(x, y_l)$ matters. We can add any constant to all rewards without changing the loss. This makes the absolute scale of rewards arbitrary, which is fine since we only need to rank responses, not assign meaningful scores.

What does the reward model learn? It learns a compressed representation of human preferences. When annotators consistently prefer responses that are accurate, the reward model learns to score accurate responses higher. When they prefer concise answers for simple questions, the reward model captures this. The reward model distills thousands of individual judgments into a function that generalizes to new prompts and responses the annotators never saw.

Counterexample: Consider training a reward model on comparisons where annotators only care about response length, preferring shorter answers. The reward model would learn $r_\phi(x, y) \approx -\text{len}(y)$, assigning higher scores to shorter responses regardless of content. This is technically correct given the training signal, but useless for alignment. The quality of RLHF depends entirely on what preferences the annotators express.

13.7.5 Policy optimization

Now we have a trained reward model that scores responses. In this final phase, we update the language model (called the policy, denoted π_θ) to produce high-reward responses. The reward model parameters ϕ are now frozen; we only update the language model parameters θ. This is a reinforcement learning problem: the model takes actions (generating tokens), receives rewards (from the reward model), and must learn a policy that maximizes expected reward.

Given prompt x, the model generates response y by sampling tokens autoregressively: $y_t \sim \pi_\theta(\cdot|x, y_{<t})$. The complete response receives reward $r_\phi(x, y)$. We want to maximize:

$$J(\theta) = \mathbb{E}_{x\sim\mathcal{D}, y\sim\pi_\theta(\cdot|x)}[r_\phi(x, y)]$$

But maximizing reward alone is dangerous. The reward model is an imperfect proxy for human preferences. If we optimize too aggressively, the policy finds responses that score high according to the reward model but are actually low quality. This is called reward hacking. For example, the reward model might have learned that confident-sounding responses tend to be preferred. The policy could exploit this by generating responses that sound extremely confident regardless of accuracy.

To prevent reward hacking, we add a constraint: the policy should not deviate too far from the original model (the reference policy π_{ref}, typically the instruction-tuned model before RLHF). We measure deviation using KL divergence:

$$\text{KL}(\pi_\theta||\pi_{ref}) = \mathbb{E}_{y\sim\pi_\theta} \left[\log \frac{\pi_\theta(y|x)}{\pi_{ref}(y|x)}\right]$$

The full RLHF objective becomes:

$$J(\theta) = \mathbb{E}_{x\sim\mathcal{D}, y\sim\pi_\theta(\cdot|x)}\left[r_\phi(x,y) - \beta \cdot \log \frac{\pi_\theta(y|x)}{\pi_{ref}(y|x)}\right]$$

The hyperparameter β controls the trade-off. Large β keeps the policy close to the reference, limiting reward but preventing divergence. Small β allows more aggressive optimization, risking reward hacking. In practice, β is tuned empirically, often starting around 0.01-0.1.

We can rewrite this as maximizing a modified reward:

$$\tilde{r}(x,y) = r_\phi(x,y) - \beta \cdot \log \frac{\pi_\theta(y|x)}{\pi_{ref}(y|x)}$$

The KL term acts as a penalty on responses that the reference model would find unlikely. If $\pi_\theta(y|x) \gg \pi_{ref}(y|x)$, the log ratio is large and positive, reducing the effective reward. This prevents the policy from drifting into regions of response space where the reference model assigns low probability, which are often degenerate or exploitative.

13.7.6 Proximal policy optimization

We optimize this objective using Proximal Policy Optimization (PPO), a reinforcement learning algorithm designed for stable training. The challenge is that the objective involves an expectation over samples from the current policy. As we update θ, the sampling distribution changes, which can cause instability.

PPO addresses this by limiting how much the policy can change in each update. Let $r_t(\theta) = \frac{\pi_\theta(y_t|x,y_{<t})}{\pi_{\theta_{old}}(y_t|x,y_{<t})}$ be the probability ratio between new and old policies for token t. PPO clips this ratio:

$$\mathcal{L}_{PPO} = -\mathbb{E}\left[\min\left(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\right)\right]$$

where $\hat{A}_t$ is the advantage estimate (how much better this action was than expected) and ϵ is a small constant like 0.2. The clipping prevents the ratio from moving too far from 1, ensuring gradual policy updates.

Computing advantages requires estimating value functions and handling the credit assignment problem: which tokens in the response contributed to the final reward? This involves training a value network alongside the policy, adding complexity. Full PPO implementations for RLHF are intricate, but the core idea is simple: take small steps, clip large changes, and gradually improve the policy.

13.7.7 Direct preference optimization

An alternative called Direct Preference Optimization (DPO) (Rafailov et al. 2023) sidesteps reinforcement learning entirely. The insight is that the optimal policy under the RLHF objective has a closed form:

$$\pi^*(y|x) = \frac{1}{Z(x)}\pi_{ref}(y|x)\exp\left(\frac{1}{\beta}r_\phi(x,y)\right)$$

where $Z(x)$ is a normalizing constant. Rearranging, the reward can be expressed in terms of policies:

$$r_\phi(x,y) = \beta\log\frac{\pi^*(y|x)}{\pi_{ref}(y|x)} + \beta\log Z(x)$$

Substituting into the Bradley-Terry preference model and simplifying (the $Z(x)$ terms cancel), we get a loss that directly optimizes the policy on preference data:

$$\mathcal{L}_{DPO} = -\mathbb{E}_{(x,y_w,y_l)}\left[\log\sigma\left(\beta\log\frac{\pi_\theta(y_w|x)}{\pi_{ref}(y_w|x)} - \beta\log\frac{\pi_\theta(y_l|x)}{\pi_{ref}(y_l|x)}\right)\right]$$

This looks complex but is straightforward to implement: compute log-probabilities under the current policy and reference policy for both responses, combine them as shown, and backpropagate. No reward model, no RL, no value functions. DPO achieves comparable results to PPO-based RLHF with simpler training.

13.7.8 Why RLHF matters

Language modeling optimizes $P(\text{text})$, the probability of text appearing in the training corpus. But we want models that are helpful, accurate, and safe. These properties correlate with probability (helpful text exists in training data) but are not the same. RLHF directly optimizes for human preferences, bridging the gap between "likely text" and "good text".

The reward model captures preferences that would be difficult to specify explicitly. How do you write down a loss function for "explains concepts clearly" or "acknowledges uncertainty appropriately"? You cannot, but you can collect examples where humans prefer one response over another, and the reward model learns the pattern.

RLHF is not without limitations. The preferences come from a specific group of annotators who may not represent all users. The reward model can be wrong. Optimization can find loopholes. But RLHF has proven essential for making language models useful in practice, transforming them from text predictors into assistants that follow instructions and avoid harmful outputs.

13.8 Comparing objectives

Objective	Architecture	Context	Primary use
Causal LM	Decoder	Unidirectional	Text generation
Masked LM	Encoder	Bidirectional	Understanding tasks
Instruction tuning	Decoder	Unidirectional	Following instructions
RLHF	Decoder	Unidirectional	Alignment with preferences

Each objective shapes what the model learns. Causal LM excels at generation. MLM excels at understanding. Instruction tuning and RLHF shape behavior and safety. Modern systems often combine these: pretrain with language modeling, then fine-tune with instructions, then align with RLHF.

13.9 Mathematical details

13.9.1 Cross-entropy loss

All the training objectives in this chapter share a common foundation: cross-entropy loss. Understanding cross-entropy requires stepping back to information theory.

13.9.1.1 Entropy: measuring uncertainty

Entropy quantifies uncertainty. If you flip a fair coin, there are two equally likely outcomes. The entropy is:

$$H = -\sum_i p_i \log_2 p_i = -\frac{1}{2}\log_2\frac{1}{2} - \frac{1}{2}\log_2\frac{1}{2} = 1 \text{ bit}$$

One bit of entropy means you need one yes/no question to determine the outcome ("Was it heads?"). If the coin is biased with $P(\text{heads}) = 0.99$, the entropy is much lower:

$$H = -0.99 \log_2 0.99 - 0.01 \log_2 0.01 \approx 0.08 \text{ bits}$$

A nearly-certain outcome has low entropy. You barely need any information to predict it.

For a probability distribution p over V outcomes:

$$H(p) = -\sum_{i=1}^{V} p_i \log p_i$$

By convention, $0 \log 0 = 0$. Entropy is maximized when all outcomes are equally likely (maximum uncertainty) and minimized when one outcome has probability 1 (no uncertainty).

13.9.1.2 Cross-entropy: comparing distributions

Cross-entropy measures how well one distribution q predicts samples from another distribution p. If the true distribution is p and we use distribution q to encode outcomes:

$$H(p, q) = -\sum_{i=1}^{V} p_i \log q_i$$

This is the average number of bits needed to encode outcomes from p using a code optimized for q. If $q = p$, cross-entropy equals entropy: $H(p, p) = H(p)$. If $q \neq p$, cross-entropy is larger. Using the wrong distribution wastes bits.

The difference $H(p, q) - H(p)$ is called the Kullback-Leibler divergence, often written $D_{KL}(p||q)$. It measures how much worse q is compared to the optimal code. KL divergence is always non-negative and equals zero only when $p = q$.

13.9.1.3 Cross-entropy for classification

In machine learning, we typically have a true label (one correct answer) rather than a full distribution. If the correct class is y, the true distribution is:

$$p_i = \begin{cases} 1 & \text{if } i = y \\ 0 & \text{otherwise} \end{cases}$$

This is a "one-hot" distribution: all probability mass on one outcome. The cross-entropy simplifies:

$$H(p, q) = -\sum_i p_i \log q_i = -1 \cdot \log q_y - \sum_{i \neq y} 0 \cdot \log q_i = -\log q_y$$

Only the predicted probability of the correct class matters. If the model assigns $q_y = 0.9$ to the correct answer, the loss is $-\log 0.9 \approx 0.105$. If the model assigns $q_y = 0.1$, the loss is $-\log 0.1 \approx 2.303$. Lower probability for the correct answer means higher loss.

13.9.1.4 The logarithm's role

Why use $-\log$ rather than just $1 - q_y$ or $(1 - q_y)^2$? Several reasons:

1. **Gradient behavior**: When the model is very wrong ($q_y \approx 0$), the gradient of $-\log q_y$ is large, providing strong learning signal. The gradient of $1 - q_y$ would be small regardless of how wrong the prediction is.

2. **Maximum likelihood**: Minimizing cross-entropy is equivalent to maximizing likelihood. If we observe tokens $y_1, y_2, \ldots, y_T$, the likelihood is $\prod_t q_{y_t}$. The log-likelihood is $\sum_t \log q_{y_t}$. Maximizing this equals minimizing $-\sum_t \log q_{y_t}$, the sum of cross-entropies.

3. **Information-theoretic meaning**: The loss $-\log q_y$ is the number of bits (or nats, if using natural log) needed to encode the outcome y using the model's distribution. Good models use fewer bits.

13.9.1.5 A concrete example

Suppose the model predicts the next token in "The capital of France is _____". The vocabulary has 50,000 tokens. The model outputs probabilities:

Token	Probability
Paris	0.72
Lyon	0.08
Berlin	0.03
France	0.02
…	…
(all others)	0.15 total

If the true next token is "Paris", the cross-entropy loss is:

$$\mathcal{L} = -\log 0.72 \approx 0.329$$

If the model had assigned only 0.01 probability to "Paris":

$$\mathcal{L} = -\log 0.01 \approx 4.605$$

The second case has 14 times higher loss. The model made a confident wrong prediction, and cross-entropy penalizes this severely.

13.9.1.6 From logits to probabilities: softmax

Neural networks don't directly output probabilities. They output logits: unbounded real numbers $z_1, z_2, \ldots, z_V$, one per vocabulary item. We convert logits to probabilities using the softmax function:

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

Softmax ensures all outputs are positive and sum to 1. Larger logits produce larger probabilities. The exponential amplifies differences: if z_i is much larger than other logits, q_i approaches 1.

The cross-entropy loss in terms of logits is:

$$\mathcal{L}_{CE} = -\log q_y = -\log \frac{\exp(z_y)}{\sum_j \exp(z_j)} = -z_y + \log \sum_j \exp(z_j)$$

This is called the "log-sum-exp" form. The first term $-z_y$ rewards high logit for the correct class. The second term $\log \sum_j \exp(z_j)$ penalizes large logits for all classes, preventing the model from making all logits arbitrarily large.

13.9.1.7 The gradient

The gradient of cross-entropy loss with respect to logits has a remarkably simple form:

$$\frac{\partial \mathcal{L}_{CE}}{\partial z_i} = q_i - \mathbb{1}_{i=y}$$

where $\mathbb{1}_{i=y}$ is 1 if $i = y$ (the correct class) and 0 otherwise.

For the correct class y: the gradient is $q_y - 1$. Since $q_y < 1$, this is negative. Gradient descent subtracts the gradient, so z_y increases, making the correct answer more likely.

For incorrect classes $i \neq y$: the gradient is $q_i - 0 = q_i$. This is positive. Gradient descent decreases z_i, making wrong answers less likely.

The magnitude matters too. If the model assigns $q_i = 0.4$ to a wrong answer, the gradient pushes down with strength 0.4. If $q_i = 0.01$, the push is weak. The model focuses on fixing its biggest mistakes.

Deriving the gradient: Let's verify this. The loss is:

$$\mathcal{L} = -z_y + \log \sum_j \exp(z_j)$$

For the first term: $\frac{\partial(-z_y)}{\partial z_i} = -\mathbb{1}_{i=y}$

For the second term, using the chain rule:

$$\frac{\partial}{\partial z_i} \log \sum_j \exp(z_j) = \frac{\exp(z_i)}{\sum_j \exp(z_j)} = q_i$$

Combining: $\frac{\partial \mathcal{L}}{\partial z_i} = q_i - \mathbb{1}_{i=y}$

13.9.1.8 Numerical stability

Computing $\exp(z_i)$ for large z_i causes overflow. Computing $\log(q_y)$ for small q_y causes underflow. In practice, we use numerically stable implementations.

For softmax, subtract the maximum logit before exponentiating:

$$q_i = \frac{\exp(z_i - \max_j z_j)}{\sum_j \exp(z_j - \max_j z_j)}$$

This gives the same result (the constant cancels) but keeps values in a reasonable range.

For log-softmax, compute directly:

$$\log q_i = z_i - \log \sum_j \exp(z_j)$$

The log-sum-exp has a stable form: $\log \sum_j \exp(z_j) = m + \log \sum_j \exp(z_j - m)$ where $m = \max_j z_j$.

Deep learning frameworks provide fused `log_softmax` and `cross_entropy` functions that handle these details automatically. Always use these rather than computing softmax and log separately.

13.9.1.9 Why cross-entropy works for language models

A language model predicts the next token at each position. With vocabulary size $V = 50,000$ and sequence length $T = 1024$, each training example involves 1024 classification problems, each over 50,000 classes.

Cross-entropy provides several advantages:

1. **Sparse gradients**: Only the correct token's probability matters for the loss value, but the gradient affects all logits. This is computationally efficient.
2. **Calibrated probabilities**: Minimizing cross-entropy encourages the model to output well-calibrated probabilities, not just correct rankings. If the model says 80% confidence, it should be right about 80% of the time.
3. **Additive over positions**: The total loss is the sum of per-position losses. This decomposes naturally over sequence positions and batches.
4. **Interpretable**: The loss in nats (natural log) or bits (log base 2) has meaning. Perplexity $= \exp(\mathcal{L})$ measures how many tokens the model is "choosing between" on average.

13.9.2 Label smoothing

To prevent overconfidence, we often use label smoothing. Instead of targeting probability 1 for the correct token:

$$q_j = \begin{cases} 1 - \epsilon + \epsilon/V & \text{if } j = y \\ \epsilon/V & \text{otherwise} \end{cases}$$

where ϵ is the smoothing parameter (typically 0.1) and V is vocabulary size. This distributes small probability mass to all tokens, encouraging the model to remain slightly uncertain.

13.9.3 Teacher forcing

During training, we use teacher forcing: the model receives true previous tokens as input, not its own predictions. This provides stable gradients but creates a train-test mismatch since at inference the model must use its own (potentially incorrect) predictions.

Scheduled sampling gradually transitions from teacher forcing to using model predictions during training, but this complicates training and is rarely used with transformers. The train-test mismatch is generally accepted as a reasonable trade-off for training stability.

The training objectives covered here transform the raw transformer architecture into capable language models. The choice of objective determines what the model learns and how it behaves. In the next chapter, we examine how these models scale with compute, data, and parameters.

Chapter 14

Scaling laws

Learning objectives

After completing this chapter, you will be able to:

- State the power law relationships between loss and compute, data, and parameters
- Apply Chinchilla-optimal ratios to determine model size given compute budget
- Explain emergent capabilities and why they appear at specific scales
- Identify practical limits to scaling: data constraints, compute costs, diminishing returns
- Use scaling laws to predict model performance and plan training runs

One of the most important discoveries in deep learning is that language model performance follows predictable mathematical relationships as we increase compute, data, and parameters (Kaplan et al. 2020). These scaling laws guide how to allocate resources and predict capabilities of future models.

14.1 The empirical observation

When we train language models across many orders of magnitude in size, a striking pattern emerges: loss decreases as a power law with each resource. This was not predicted from theory. Researchers discovered it by training hundreds of models at different scales and plotting the results.

14.1.1 How the laws were discovered

In January 2020, researchers at OpenAI published "Scaling Laws for Neural Language Models" (Kaplan et al. 2020). They trained over 400 transformer language models ranging from 768 parameters to 1.5 billion parameters on datasets from 22 million to 23 billion tokens. They varied model width, depth, batch size, and learning rate systematically.

When they plotted test loss against model size on logarithmic axes, the points fell on straight lines. A straight line on log-log axes indicates a power law relationship: $L = aN^{-\alpha}$, or equivalently $\log L = \log a - \alpha \log N$. The slope gives the exponent α.

This was surprising. There was no theoretical reason to expect such clean relationships across four orders of magnitude in model size. Yet the pattern held consistently.

14.1.2 Loss versus parameters

For a model with N parameters trained on sufficient data:

$$L(N) = \left(\frac{N_c}{N}\right)^{\alpha_N}$$

The constants come from fitting curves to empirical data:

- $N_c \approx 8.8 \times 10^{13}$: This is where the power law, if extended, would hit zero loss. It has no physical meaning since we cannot build a model with 10^{13} parameters (roughly 10,000 times larger than GPT-4). It is simply a fitted constant that makes the formula match observations.
- $\alpha_N \approx 0.076$: This exponent determines how fast loss decreases with scale. The value 0.076 means doubling parameters reduces loss by a factor of $2^{0.076} \approx 1.054$, roughly 5% improvement. This small exponent explains why we need enormous scale increases for meaningful gains.

These specific numbers came from fitting to OpenAI's training runs on their specific data (WebText, a curated web scrape). Different data, different tokenization, or different architectures would yield different constants. The power law form appears universal; the specific coefficients are not.

14.1.3 Loss versus data

For a model trained on D tokens with sufficient parameters:

$$L(D) = \left(\frac{D_c}{D}\right)^{\alpha_D}$$

- $D_c \approx 5.4 \times 10^{13}$: Another fitted constant with no direct physical interpretation. It represents the amount of data where the model would achieve near-zero loss if the power law continued indefinitely.
- $\alpha_D \approx 0.095$: Slightly larger than α_N, meaning data scales somewhat more efficiently than parameters. Doubling data gives roughly 7% loss reduction.

The methodology was the same: train models with fixed size on varying amounts of data, plot loss versus data tokens on log-log axes, fit a line.

14.1.4 Loss versus compute

For optimal allocation of compute budget C (measured in FLOPs):

$$L(C) = \left(\frac{C_c}{C}\right)^{\alpha_C}$$

- $C_c \approx 3.1 \times 10^8$: This small value reflects that even modest compute budgets (a few hundred million FLOPs, achievable on a laptop) can train a simple language model with some predictive ability.
- $\alpha_C \approx 0.050$: The smallest exponent, because compute compounds both parameters and data. Each 10x in compute gives about 12% loss reduction.

14.1.5 Why these specific numbers?

The honest answer: we do not know why these exponents take these values. The power law form suggests deep regularities in how neural networks learn from data, but no first-principles theory predicts $\alpha_N = 0.076$ rather than 0.1 or 0.05.

Several observations constrain the values:

1. **Exponents less than 1**: If $\alpha > 1$, doubling resources would more than double the improvement, leading to explosive gains. We observe diminishing returns instead.
2. **Exponents greater than 0**: If $\alpha \leq 0$, scaling would not help. Empirically, bigger models perform better.

3. **Similar magnitudes**: All three exponents are between 0.05 and 0.1. This may reflect that the fundamental bottleneck is similar regardless of which resource we scale.

The universality across architectures is remarkable. GPT-style decoders, BERT-style encoders, and various modifications all show power law scaling with similar exponents. This suggests the laws capture something about learning from text, not about specific architectural choices.

14.1.6 Chinchilla revisions

In 2022, DeepMind's Chinchilla paper (Hoffmann et al. 2022) revisited these measurements with more careful experiments. They found slightly different exponents and, more importantly, different optimal tradeoffs between parameters and data. The original OpenAI work suggested making models as large as possible; Chinchilla showed that balanced scaling of parameters and data is more efficient.

This illustrates an important point: the specific constants are empirical measurements subject to revision. The qualitative finding (power law scaling with diminishing returns) is robust; the exact numbers depend on experimental details.

14.2 The unified scaling law

These individual relationships combine into a single formula:

$$L(N, D) = \left[\left(\frac{N_c}{N} \right)^{\alpha_N/\alpha_D} + \frac{D_c}{D} \right]^{\alpha_D}$$

This captures how loss depends jointly on model size and data. The formula reveals diminishing returns: as you scale one resource, the benefit decreases unless you also scale the other.

14.3 Compute-optimal training

Given a fixed compute budget, how should we split it between model size and training tokens?

14.3.1 The Chinchilla finding

Early scaling work suggested making models as large as possible for a given compute budget. But Hoffmann et al. (Hoffmann et al. 2022) showed this is suboptimal. They found that parameters and training tokens should scale equally:

$$N_{opt} \propto C^{0.5}, \quad D_{opt} \propto C^{0.5}$$

For a compute budget C, the optimal model has roughly $N \approx C^{0.5}$ parameters trained on $D \approx C^{0.5}$ tokens.

14.3.2 The practical rule

A useful approximation: train on roughly 20 tokens per parameter. A 7 billion parameter model should train on about 140 billion tokens. A 70 billion parameter model should train on about 1.4 trillion tokens.

This overturned previous practice. GPT-3 (Brown et al. 2020) (175B parameters) trained on 300B tokens, far below the compute-optimal ratio. Chinchilla (70B parameters) trained on 1.4T tokens achieved similar performance with 4x fewer parameters, making inference much cheaper.

14.3.3 Why this matters

Inference cost scales with parameter count. A model trained compute-optimally has fewer parameters for the same capability, reducing deployment costs. This shifts the economics: spend more on training (one-time cost) to save on inference (ongoing cost).

14.4 What drives scaling

14.4.1 The loss decomposition

Test loss can be decomposed into irreducible entropy and reducible error:

$$L = L_\infty + L_{reducible}$$

The irreducible entropy L_∞ represents fundamental uncertainty in the data. Consider the sentence "I flipped a coin and got ____". No model, however large, can predict whether the next word is"heads" or "tails" better than chance. The outcome is genuinely random given the context. Natural language contains many such unpredictable elements: which synonym an author chose, random numbers in text, names of people in stories.

Estimates suggest $L_\infty \approx 1.5$ nats for natural language, corresponding to a perplexity of about 4.5. This means even a perfect model would be "choosing between" about 4-5 equally likely options on average. The reducible error $L_{reducible}$ is everything above this floor, and it decreases with scale following the power laws.

14.4.2 What is a power law?

Before asking why scaling follows power laws, we should understand what a power law actually is and how it differs from other relationships.

A power law relates two quantities where one is proportional to the other raised to some power:

$$y = a \cdot x^b$$

Here a is a constant multiplier and b is the exponent. In scaling laws, we typically write this as $L = (N_c/N)^\alpha$, which is equivalent with $a = N_c^\alpha$ and $b = -\alpha$.

Contrast with linear relationships: In a linear relationship $y = ax$, doubling x always doubles y. The absolute increase is proportional to x: going from 10 to 20 adds the same as going from 100 to 110 in relative terms, but very different amounts absolutely.

Contrast with exponential relationships: In an exponential $y = a \cdot b^x$, each unit increase in x multiplies y by a fixed factor. Exponentials grow (or decay) extremely fast. Moore's law (transistors doubling every 18 months) is exponential in time.

Power laws are in between: In a power law $y = ax^b$ with $0 < b < 1$, doubling x multiplies y by 2^b, a fixed factor less than 2. This is "sublinear" growth: you get improvement, but with diminishing returns. Each doubling helps less in absolute terms.

A concrete example: Suppose loss scales as $L = 10 \cdot N^{-0.1}$.

Parameters N	Loss L	Improvement from previous
1 million	5.01	-
10 million	3.98	21%
100 million	3.16	21%
1 billion	2.51	21%
10 billion	2.00	21%

Each 10x increase in parameters gives the same *percentage* improvement (about 21%), but the *absolute* improvement shrinks: from 5.01 to 3.98 is a drop of 1.03, but from 2.51 to 2.00 is only 0.51. This is the "diminishing returns" character of power laws.

The log-log signature: Power laws have a distinctive visual signature. If you plot y vs x on regular axes, you see a curve that drops steeply at first then flattens (for negative exponents like in scaling laws). But if you plot $\log y$ vs $\log x$, you get a straight line:

$$\log y = \log a + b \log x$$

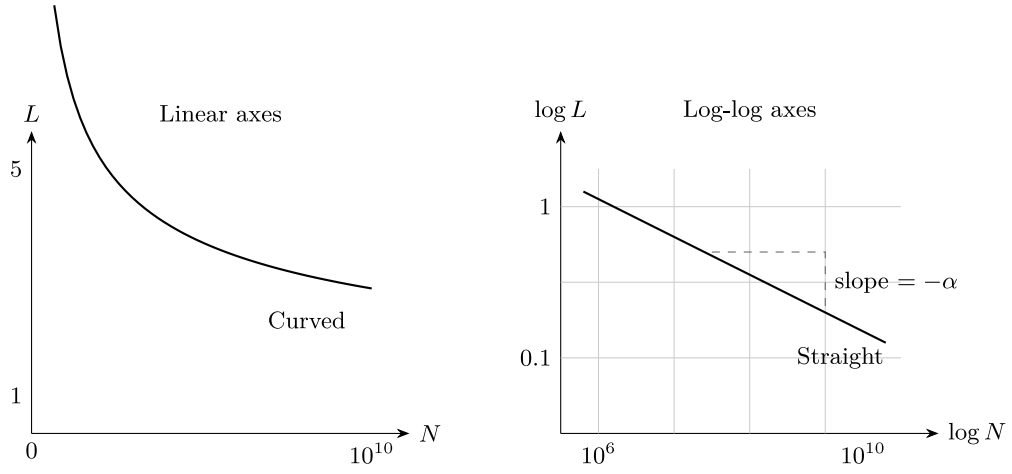

Figure 14.1: Power law on linear axes (left) shows a curve; the same relationship on log-log axes (right) becomes a straight line. The slope of the line equals the exponent $-\alpha$.

This is why researchers always plot scaling results on log-log axes (Figure 14.1). A straight line confirms power law behavior; the slope gives the exponent directly.

Why are power laws special? Power laws are "scale-free." The relationship looks the same whether you are at small scale or large scale. If you zoom in on any part of a log-log plot of a power law, it looks identical to any other part. This self-similarity suggests the underlying process has no characteristic scale.

Compare this to a relationship with a characteristic scale, like $y = e^{-x/x_0}$. The behavior changes fundamentally around $x = x_0$. Power laws have no such transition point (until other effects intervene, like the irreducible loss floor).

14.4.3 Why power laws in neural networks?

Power laws appear throughout nature: earthquake magnitudes, city populations, word frequencies, species abundances. When we see a power law, it usually indicates some underlying scale-free process. But why would neural network training produce power laws?

Several theories have been proposed. None is fully satisfactory, but together they illuminate different aspects of the phenomenon.

14.4.3.1 The manifold hypothesis

Natural language does not fill the entire space of possible token sequences. Most random sequences are gibberish. Meaningful text lies on a lower-dimensional structure within the high-dimensional space of all possible sequences.

Imagine the space of all 1000-token sequences. With vocabulary 50,000, this space has 50000^{1000} points. But coherent English text occupies a tiny fraction of this space. The "manifold hypothesis" says this fraction forms a smooth, lower-dimensional surface.

A neural network approximates this manifold. A small network can only capture a crude approximation, like fitting a plane to a curved surface. As we add parameters, the network can represent finer details: curves, bumps, wrinkles. The error in approximating a smooth manifold typically decreases as a power of the approximation capacity.

Why power law specifically? If the manifold has intrinsic dimension d and we approximate it with a model of capacity N, approximation theory suggests error scales as $N^{-\alpha}$ where α depends on the smoothness of the manifold and the dimension. For sufficiently smooth manifolds, this gives power law scaling.

The theory has limitations. We do not know the actual dimension or smoothness of the "language manifold." The theory predicts that α should depend on these properties, but empirically the exponent is remarkably consistent across different data distributions. This suggests something more universal is at play.

14.4.3.2 The random feature perspective

Another view focuses on what happens inside the network. A randomly initialized neural network already computes a large set of random features (nonlinear combinations of inputs). Training selects which features to use for prediction.

Consider a network with N parameters computing $\sim N$ random features. Some features are useful for predicting text; most are not. As N increases, we get more features, and by chance, some of the new features are useful. The probability of finding a useful feature among random ones often follows power law statistics.

More precisely: suppose the "usefulness" of random features follows a heavy-tailed distribution, where a few features are very useful and most are nearly useless. This is plausible because useful features (like "detects question syntax" or "tracks subject-verb agreement") are specific, while random features are generic. The number of useful features you find among N random ones scales as N^{α} for some $\alpha < 1$.

This explains why the exponent is less than 1: doubling parameters does not double the number of useful features, because useful features are rare. It also suggests the exponent should be universal, depending on the statistics of useful features rather than the specific task.

14.4.3.3 The loss landscape perspective

Training a neural network means minimizing a loss function over a high-dimensional parameter space. The geometry of this "loss landscape" affects what solutions we find.

Small networks have rugged loss landscapes with many local minima separated by high barriers. The optimizer gets stuck in mediocre solutions. Large networks have smoother landscapes where minima are connected by low-loss paths. The optimizer can find better solutions.

Why does landscape smoothness improve with scale? One argument: in high dimensions, most directions are "neutral" (neither uphill nor downhill). Saddle points are more common than local minima. A large network has so many parameters that it can almost always find a direction to escape bad regions. The loss landscape becomes a gentle slope toward good solutions rather than a maze of traps.

This connects to the observation that larger models are easier to train, not harder. Learning rate and other hyperparameters transfer across scales. If the landscape became more complex with scale, we would expect training to become more difficult, but the opposite occurs.

14.4.3.4 The statistical mechanics perspective

Physicists have studied power laws for over a century in the context of phase transitions and critical phenomena. Systems at criticality (the boundary between two phases, like water at the boiling point) exhibit power law correlations.

Some researchers propose that neural networks during training operate near a critical point. The network balances between underfitting (too simple, high bias) and overfitting (too complex, high variance). At this boundary, power law scaling emerges naturally.

The analogy goes further. In statistical mechanics, power laws arise when the system has no characteristic scale. A network near the interpolation threshold (just enough capacity to fit the training data) might similarly be "scale-free," with features at all sizes contributing to the prediction.

This theory makes a specific prediction: the scaling exponent should be related to "critical exponents" that characterize the universality class of the learning process. Different architectures might fall into different universality classes, but within a class, the exponent should be fixed. This matches the observation that transformers of different sizes and configurations show similar exponents.

14.4.3.5 The data structure hypothesis

Perhaps the power law comes not from the model but from the data. Natural language has power law statistics at multiple levels: word frequencies follow Zipf's law, phrase frequencies decay as power laws, topic distributions are heavy-tailed.

If learning proceeds by capturing patterns from most common to least common, and pattern frequencies follow a power law, then the rate of improvement might inherit this power law structure. A model of capacity N can capture patterns down to some frequency threshold. As N increases, the threshold drops, capturing rarer patterns. The reduction in loss depends on how much probability mass lies in the newly captured patterns.

Under Zipfian statistics, this gives power law scaling with an exponent determined by the Zipf exponent. English has Zipf exponent approximately 1, which would predict a specific scaling exponent. The observed exponents are in the right ballpark, though the detailed predictions do not match perfectly.

14.4.3.6 Synthesis: no single explanation

Each theory captures something real:

- The manifold hypothesis explains why larger models generalize better
- The random feature view explains why useful capacity grows sublinearly
- The loss landscape theory explains why larger models are easier to train
- Statistical mechanics provides a framework for universality
- Data structure explains why the exponent might be consistent across tasks

The remarkable fact is that all these mechanisms, arising from different aspects of learning, conspire to produce nearly the same power law exponents. This suggests an underlying unity we do not yet understand.

The exponents $\alpha_N \approx 0.076$, $\alpha_D \approx 0.095$ are not predicted by any theory. They are measured. A complete theory of deep learning would derive these numbers from first principles. We are far from that today.

14.4.4 The irreducible loss

The scaling laws predict that loss approaches zero as resources approach infinity. This cannot be literally true. The irreducible loss L_∞ sets a floor.

We can write the full scaling law as:

$$L(N, D) = L_\infty + \left(\frac{N_c}{N}\right)^{\alpha_N} + \left(\frac{D_c}{D}\right)^{\alpha_D}$$

For current models, L_∞ is small compared to the reducible terms, so we often ignore it. But as models approach human-level performance, the irreducible term will dominate. Further scaling will yield diminishing returns as we bump against fundamental unpredictability in text.

Estimating L_∞ is difficult because it requires extrapolating current trends far beyond observed scales. Different estimation methods give values between 1.0 and 1.8 nats. This uncertainty matters for predicting ultimate model capabilities.

14.5 Emergent capabilities

Beyond smooth loss improvements, scaling produces qualitative jumps in capability (Wei et al. 2022). These "emergent" abilities appear suddenly at specific scales, surprising researchers who expected gradual improvement.

14.5.1 Phase transitions

Some abilities appear suddenly at specific scales:

- **Multi-digit arithmetic**: Models below 10 billion parameters fail at three-digit addition, performing near random chance. Above this threshold, accuracy jumps to over 80%. There is no gradual improvement; the capability switches on.
- **Chain-of-thought reasoning**: When prompted to "think step by step," small models ignore the instruction or produce incoherent steps. Large models (roughly 100B+ parameters) suddenly use the steps productively, improving accuracy on math and logic problems by 20-50 percentage points.
- **In-context learning**: The ability to learn new tasks from a few examples in the prompt. Small models treat examples as context but do not generalize. Large models extract the pattern and apply it to new instances.
- **Word unscrambling**: Given "elppa" and asked for the unscrambled word, small models fail entirely. Large models succeed, suggesting they can mentally manipulate token sequences.
- **Logical fallacy detection**: Identifying invalid arguments requires understanding both the logical structure and the content. This emerges around 50-100B parameters.

The pattern is consistent: performance is flat (near random) below a threshold, then jumps sharply. The transition occurs over less than one order of magnitude in model size, sometimes just a 2-3x increase.

14.5.2 Why emergence happens

Several hypotheses explain sudden capability emergence:

Circuit formation: Perhaps a capability requires multiple components working together. A model might separately learn "parse numbers," "apply addition algorithm," and "format output." Only when all components are present does the capability work. Below the threshold, one component is missing. Above it, the circuit is complete.

Representation phase transitions: The internal representations might undergo qualitative changes at scale. Small models represent words; larger models might represent abstract concepts, relationships, or reasoning patterns. When the representation crosses a complexity threshold, new capabilities become possible.

Task decomposition: Complex tasks decompose into subtasks. A model might need to solve subtasks reliably before the full task works. If subtask accuracy is 60%, and the full task requires three subtasks, success rate is $0.6^3 = 22\%$. Improving subtask accuracy to 90% gives $0.9^3 = 73\%$. Small improvements in components yield large improvements in composite tasks.

Measurement artifacts: A controversial view holds that emergence is partly an illusion of how we measure. If we use accuracy (right/wrong) as our metric, a task that requires 90% of steps correct will show 0% accuracy until the model crosses 90% per-step accuracy, then jump to high accuracy. Smoother metrics like log-probability might show gradual improvement where accuracy shows a jump.

Research suggests both perspectives have merit. Some emergence is real (qualitative changes in what models can do), some is measurement artifact (smooth underlying improvement appearing discontinuous due to threshold metrics).

14.5.3 Predicting emergence

We cannot reliably predict when capabilities will emerge. This is a major challenge for AI safety and planning.

Loss decreases smoothly and predictably. We can forecast the loss of a 10x larger model with reasonable accuracy. But we cannot forecast what new capabilities that loss improvement will unlock. A model with 5% lower loss might have no new abilities, or it might suddenly solve a class of problems it previously failed on completely.

Several approaches attempt to predict emergence:

Extrapolating capability curves: If we measure a capability at multiple scales, we might extrapolate when it will reach useful levels. But this requires the capability to show some signal below the emergence threshold, which is often not the case.

Proxy tasks: Sometimes simpler versions of a task show gradual improvement. If "two-digit addition" improves gradually, we might predict when "five-digit addition" will emerge. But the relationship between proxy and target tasks is often unclear.

Theoretical analysis: Understanding why a capability requires scale might predict when it emerges. If we knew that chain-of-thought requires a certain minimum context window or representation capacity, we could predict the threshold. But we rarely have such understanding.

Empirical discovery: In practice, we discover emergent capabilities by training large models and testing them. This is expensive and reactive. We often do not know what a model can do until we try.

The unpredictability of emergence is one reason AI development is hard to forecast. Scaling laws tell us loss will decrease, but not what that means for real-world capabilities.

14.6 Limits to scaling

Scaling laws suggest that we could keep improving models forever by adding more compute, data, and parameters. In practice, we face hard limits on each resource.

14.6.1 Data constraints

The Chinchilla-optimal recipe requires roughly 20 tokens of training data per parameter. A 1 trillion parameter model needs 20 trillion tokens. Where does this data come from?

The internet: Common Crawl, a snapshot of the public web, contains roughly 100 trillion tokens. But most of this is low quality: spam, duplicate content, machine-generated text, navigation menus, cookie notices. After filtering for quality, perhaps 10-20 trillion tokens of useful text remain.

Books: Project Gutenberg, library digitization efforts, and commercial ebooks provide perhaps 100 billion tokens of high-quality, edited prose. This is a small fraction of web data but higher quality per token.

Code: GitHub and other repositories contain trillions of tokens of code. Code is highly structured and teaches models about logic, syntax, and precise instruction-following. Most frontier models train on significant code despite targeting natural language.

Scientific literature: Academic papers, patents, and technical documents provide specialized knowledge. PubMed alone contains billions of tokens of biomedical text.

Curated datasets: Wikipedia, StackOverflow, Reddit (with filtering), and other curated sources provide moderate amounts of high-quality text.

The total pool of quality text is finite. Current estimates suggest 5-15 trillion tokens of "good" training data exist in digitized form. Models are already training on significant fractions of this. Llama 2 trained on 2 trillion tokens; GPT-4 likely used more.

Strategies for data scarcity:

Synthetic data: Use existing models to generate training data for new models. This works but risks "model collapse" if the synthetic data distribution drifts from natural text. Careful filtering is required.

Multi-epoch training: Train on the same data multiple times. Diminishing returns set in after 2-4 epochs; the model memorizes rather than generalizes. But some repetition is better than no data.

Multi-modal data: Images, video, and audio contain information that text alone does not. A model that can learn from video (roughly 100x more data than text) escapes the text data limit. This requires architectural changes.

Active data collection: Pay humans to write text specifically for training. Expensive, but produces high-quality, targeted data. Used for instruction tuning and RLHF, but too expensive for pretraining at scale.

14.6.2 Compute constraints

Training frontier models requires extraordinary computational resources.

Current scale: GPT-4 reportedly required 25,000 A100 GPUs training for approximately 3 months. At cloud rental prices of $2 per GPU-hour, the compute cost alone exceeds $100 million. Actual costs are higher due to engineering, failed runs, and infrastructure.

Hardware availability: The world produces perhaps 500,000 high-end AI GPUs per year. A single frontier training run might consume 5-10% of annual production. Supply chains, chip manufacturing capacity, and geopolitical factors limit GPU availability.

Energy consumption: A large training run consumes tens of megawatts continuously for months. Data centers require this power plus cooling. Locating facilities near cheap, abundant power is a real constraint. A 1 gigawatt data center (plausible for frontier AI by 2030) would consume as much power as a small city.

Memory and communication: Models with trillions of parameters do not fit in a single GPU's memory (currently 80GB for high-end chips). They must be split across thousands of GPUs with high-bandwidth interconnects. The communication overhead becomes a bottleneck. Training efficiency (actual FLOPS achieved vs. theoretical peak) drops as models span more devices.

Physical limits: Moore's law has slowed. Transistor density improvements that once doubled every 18 months now take 3+ years. New architectures (wafer-scale chips, optical interconnects, analog computing) might help, but fundamental physical limits loom within a few decades.

14.6.3 Economic constraints

Even if data and compute were physically available, the economics become challenging.

Training costs: Current frontier models cost $100M-$1B to train. If scaling laws hold, a model 100x better would require roughly $10B-$100B in compute costs. This approaches the R&D budgets of entire nations.

Diminishing returns: The power law exponent $\alpha_C \approx 0.05$ means 10x more compute yields only 12% lower loss. To halve the loss requires $2^{1/0.05} \approx 10^6$ times more compute. At some point, the marginal improvement per dollar becomes negligible compared to other research directions.

Inference economics: Larger models cost more to run. A model with 10x more parameters costs roughly 10x more per query. If the capability improvement is only 12%, the cost-effectiveness of serving the model decreases. Training a compute-optimal (smaller) model often dominates training the largest possible model.

Opportunity cost: Resources spent scaling one model are not spent on architectural innovations, dataset improvements, or new training methods. The optimal allocation between scaling and research is unclear, but pure scaling is unlikely to be optimal.

14.6.4 The data wall

Many researchers believe data is the binding constraint. We can build bigger GPUs, but we cannot create more Shakespeare or more Wikipedia. Synthetic data helps but has limits. This "data wall" may force a shift from scaling to other approaches: better architectures, improved training efficiency, or learning from less data (sample efficiency).

Whether the data wall is hard (fundamentally insurmountable) or soft (surmountable with synthetic data and multi-modal learning) is actively debated. The answer will determine the trajectory of AI development.

14.7 Practical implications

Scaling laws are not just academic curiosities. They have transformed how practitioners, researchers, and policymakers approach language models.

14.7.1 For practitioners

When designing a model for a specific task, scaling laws provide a framework for resource allocation.

Estimating compute budgets: Start with the target capability. If existing models at scale X achieve 70% accuracy on your task, and the task-specific scaling exponent suggests 10x compute yields 10% improvement, you can estimate the compute needed for 90% accuracy. This is imprecise but better than guessing.

Choosing model size: Given a compute budget C (in FLOPs), the Chinchilla-optimal model has approximately:

$$N_{opt} \approx 0.7 \times 10^9 \times \left(\frac{C}{10^{21}}\right)^{0.5} \text{ parameters}$$

$$D_{opt} \approx 20 \times N_{opt} \text{ tokens}$$

For example, with $C = 10^{23}$ FLOPs (a modest budget by frontier standards), the optimal model has about 7 billion parameters trained on 140 billion tokens.

Data-limited scenarios: If you have less data than $20N$ tokens, you are data-limited. Options include: - Use a smaller model that matches your data - Accept some compute inefficiency and train the larger model anyway (sometimes worthwhile if inference cost matters) - Augment data through synthetic generation or multi-epoch training

Inference optimization: Remember that training cost is paid once; inference cost is paid per query. If your application will serve millions of queries, a smaller model trained slightly suboptimally for compute efficiency might be cheaper overall than the compute-optimal model.

Fine-tuning considerations: Scaling laws apply to pretraining. Fine-tuning on task-specific data has different dynamics. A model pretrained at scale retains its capabilities when fine-tuned with much less data. You do not need to repeat the scaling analysis for fine-tuning; use the largest pretrained model you can afford to run.

14.7.2 For researchers

Scaling laws enable efficient research by allowing extrapolation from small experiments.

Predicting performance: Train models at 1%, 3%, and 10% of your target scale. Plot loss vs. compute on log-log axes. If the points form a line, extrapolate to predict the full-scale result. This lets you estimate whether a research direction is promising before committing full resources.

Comparing methods: If method A achieves loss L_A at compute C, and method B achieves L_B at the same compute, you can estimate the "effective compute multiplier" of method B: how much compute would method A need to match method B's loss? This normalizes for the quality vs. efficiency tradeoff.

Hyperparameter transfer: Scaling laws suggest that optimal hyperparameters (learning rate, batch size) transfer across scales with predictable adjustments. Learning rate typically scales as $N^{-0.5}$; batch size scales as $N^{0.5}$. This reduces the hyperparameter search space at large scale.

Identifying scaling bottlenecks: If your model scales worse than expected (exponent smaller than baseline), something is limiting scaling. This might be data quality, architectural bottlenecks, or optimization issues. Scaling experiments can diagnose problems.

Efficient ablations: To test whether a modification helps, compare scaling curves rather than single points. A modification that helps at small scale but hurts the scaling exponent will eventually underperform the baseline. Conversely, a modification that looks neutral at small scale but improves the exponent is valuable.

14.7.3 For society and policy

Scaling laws have implications beyond technical AI development.

Capability forecasting: If scaling laws hold, we can estimate when models might achieve specific capabilities. If current frontier models are at 10^{25} FLOPs and a capability is expected to emerge at 10^{27} FLOPs, we can estimate the timeline based on hardware and investment trends. This informs safety research priorities.

Compute governance: Compute is measurable and controllable in ways that algorithms and data are not. Understanding how compute translates to capability informs policies about compute access, export controls, and international agreements.

Economic projections: The cost of AI capabilities is predictable from scaling laws. If capability X requires 10^{26} FLOPs, and compute costs fall 30% per year, we can project when X becomes economically viable for various applications.

Safety implications: Emergent capabilities mean that a model slightly larger than the current frontier might have qualitatively new abilities. This argues for careful, incremental scaling with evaluation at each step, rather than racing to the largest possible model.

Resource allocation: Understanding diminishing returns helps allocate AI investment. If the next 10x in compute yields only modest improvements, perhaps resources are better spent on data quality, alignment research, or deployment infrastructure.

14.8 Beyond loss

Loss is a convenient metric because it is smooth, continuous, and easy to measure. But we ultimately care about task performance: can the model answer questions, write code, follow instructions? The relationship between loss and task performance is complex.

14.8.1 Task performance scaling

Different tasks scale differently with model size and loss:

Knowledge-intensive tasks (trivia, factual questions) scale well with model size. Larger models memorize more facts from training data. The scaling exponent for knowledge retrieval is relatively high.

Reasoning tasks (math problems, logic puzzles) scale steeply but with higher variance. Small models fail completely; large models show rapid improvement. The emergence phenomenon is strongest for reasoning.

Pattern matching tasks (sentiment classification, language identification) scale slowly because small models already perform well. The task saturates before scaling laws matter much.

Generation quality (coherent writing, appropriate tone) scales steadily. Larger models produce more fluent, coherent, and contextually appropriate text. Human evaluations correlate with log model size.

Instruction following scales with both pretraining and instruction tuning. Larger pretrained models learn to follow instructions more easily during fine-tuning.

The relationship between loss and task accuracy is often sigmoidal. At high loss, accuracy is near chance. As loss decreases, accuracy improves slowly, then rapidly, then saturates. The "knee" of the sigmoid varies by task. This means:

- A 10% loss improvement might yield 1% accuracy improvement on an easy task (already saturated)
- The same loss improvement might yield 20% accuracy improvement on a hard task (in the steep region)
- Or 0% improvement on an impossibly hard task (below the threshold)

Predicting which tasks benefit from scaling requires understanding where each task sits on its sigmoid curve.

14.8.2 Efficiency innovations

Scaling laws describe a particular architecture (transformers) trained in a particular way (standard pretraining). Innovations can shift the curves, achieving better loss for the same compute.

Architectural improvements: Flash attention reduces memory and compute for attention operations by 2-4x. Mixture-of-experts models activate only a fraction of parameters per token, achieving better loss per training FLOP (though inference cost remains high). Sparse attention patterns, linear attention, and state-space models each offer different tradeoffs.

Each innovation can be characterized by its "effective compute multiplier": how much baseline compute would achieve the same loss? Flash attention might provide a 2x multiplier; mixture-of-experts might provide 4x for training (less for inference).

Training improvements: Better optimizers (AdamW, Lion), learning rate schedules (cosine decay, warmup), and regularization techniques (dropout, weight decay) improve training efficiency. These compound: a 10% improvement from the optimizer and 10% from the schedule yields 21% overall.

Data improvements: Filtering training data for quality, deduplicating, and balancing domains improves loss more than raw data quantity. A dataset that is 10x smaller but 10x higher quality might train a better model. This shifts the data scaling curve, achieving lower loss per token.

Quantization and distillation: Running models at lower precision (8-bit, 4-bit) or distilling large models into smaller ones does not improve scaling laws but changes the pareto frontier of capability vs. inference cost. A distilled 7B model might match a dense 30B model on many tasks at 4x lower inference cost.

Important caveat: Innovations rarely change the scaling exponents. They shift the curves vertically (better loss at all scales) but the slope remains similar. A 2x efficiency improvement saves one "doubling" worth of compute but does not change how many doublings are needed for a given improvement. This is why scaling laws remain relevant even as techniques improve.

14.8.3 Multi-modal scaling

Scaling laws extend to multi-modal models (text + images, text + code, etc.), but with modifications:

- Different modalities may have different exponents
- Cross-modal tasks (image captioning, visual question answering) may scale differently than single-modal tasks
- The optimal modality mix depends on the target task

Early results suggest that multi-modal training does not violate scaling laws but adds complexity to the resource allocation problem. How much image data versus text data? How to weight losses across modalities? These questions do not yet have definitive answers.

14.9 Mathematical framework

Understanding the mathematics behind scaling laws helps us apply them correctly and understand their limitations.

14.9.1 The power law form

A power law has the form $y = ax^b$ where a and b are constants. Power laws are "scale-invariant": if you zoom in or out on a log-log plot, the relationship looks identical. This makes them natural for phenomena spanning many orders of magnitude.

Taking logarithms linearizes the relationship:

$$\log L = \log a - \alpha \log N$$

On log-log axes, this is a straight line with slope $-\alpha$ and intercept $\log a$. The negative slope reflects that loss decreases as resources increase.

Why plot on log-log axes?: If the relationship is truly a power law, log-log plotting reveals it immediately as a straight line. If the relationship is exponential, polynomial, or some other form, it will curve on log-log axes. Log-log plots also make it easy to visualize many orders of magnitude on a single graph.

Reading the exponent: The slope of the log-log line gives the exponent directly. A slope of -0.076 means doubling N multiplies L by $2^{-0.076} = 0.949$, a 5.1% reduction. A slope of -0.1 would give $2^{-0.1} = 0.933$, a 6.7% reduction. Small changes in exponent matter when compounded over many doublings.

14.9.2 The unified scaling law

When both parameters and data vary, the loss follows:

$$L(N, D) = \left[\left(\frac{N_c}{N} \right)^{\alpha_N/\alpha_D} + \frac{D_c}{D} \right]^{\alpha_D}$$

This formula captures the interaction between resources. Let us unpack it.

When N is very large (effectively infinite), the first term vanishes:

$$L(N \to \infty, D) = \left(\frac{D_c}{D} \right)^{\alpha_D}$$

We recover the data-only scaling law. Similarly, when D is very large:

$$L(N, D \to \infty) = \left(\frac{N_c}{N} \right)^{\alpha_N}$$

We recover the parameter-only scaling law.

When both are finite, neither resource alone limits performance. The formula interpolates smoothly between regimes.

The structure $[A + B]^{\alpha_D}$ with $A = (N_c/N)^{\alpha_N/\alpha_D}$ and $B = D_c/D$ has a specific meaning: it is as if we are adding two "effective data deficits." Insufficient parameters act like insufficient data, with a conversion factor $\alpha_N/\alpha_D \approx 0.8$ between them.

14.9.3 Fitting scaling laws

To fit scaling laws empirically:

Step 1: Train models at multiple scales. Vary N (or D, or both) across at least 2-3 orders of magnitude. For example, train models with 10M, 30M, 100M, 300M, 1B, and 3B parameters. More points give more reliable fits.

Step 2: Measure test loss. Use a held-out test set that the model never saw during training. The test set should be large enough that measurement noise is small. Compute loss in nats (natural log) or bits (log base 2) consistently.

Step 3: Fit in log space. Transform to $\log L$ vs. $\log N$. Fit a line using ordinary least squares:

$$\log L_i = \beta_0 + \beta_1 \log N_i + \epsilon_i$$

The fitted slope $\hat{\beta}_1 = -\hat{\alpha}$ gives the exponent. The intercept $\hat{\beta}_0 = \log \hat{a}$ gives the prefactor.

Step 4: Validate. Hold out one or two data points from the fit. Predict their loss from the fitted law. If predictions are accurate (within a few percent), the law is reliable. If predictions are far off, the law may not hold, or more data points are needed.

Step 5: Estimate uncertainty. Bootstrap or compute standard errors on the fitted parameters. The uncertainty in α translates to uncertainty in extrapolated loss. Small errors in α compound when extrapolating many orders of magnitude.

Common pitfalls:

- *Insufficient scale range*: Fitting over less than two orders of magnitude often gives unreliable exponents. The curvature from irreducible loss or saturation can masquerade as different slopes.
- *Underfitting at small scale*: Very small models may not train to convergence or may have different optimization dynamics. Exclude the smallest models if they deviate systematically.
- *Overfitting at large scale*: If your largest model trains on so much data that it starts memorizing, the test loss may be artificially low. Ensure sufficient test data.

14.9.4 Computing optimal allocations

Given a compute budget C, we want to find the optimal N and D. Training a model with N parameters on D tokens requires approximately $C = 6ND$ FLOPs (a commonly used approximation where the factor 6 accounts for forward and backward passes with some overhead).

The constraint is $D = C/(6N)$. Substituting into the loss formula:

$$L(N) = L\left(N, \frac{C}{6N}\right)$$

Minimizing over N gives the optimal model size for budget C. Taking derivatives and solving (algebra omitted) yields:

$$N_{opt} \propto C^a, \quad D_{opt} \propto C^{1-a}$$

where a depends on the exponents α_N and α_D. The original OpenAI paper found $a \approx 0.73$ (favor larger models). Chinchilla found $a \approx 0.5$ (balance parameters and data). The difference comes from different experimental setups and fitting procedures.

The practical rule "$D \approx 20N$" corresponds to $a = 0.5$ (Chinchilla) with the $C = 6ND$ compute formula.

14.9.5 Extrapolation risks

Scaling laws are empirical fits, not physical laws. Extrapolating beyond observed scales carries risks:

Phase transitions: The scaling exponent might change at some scale. Perhaps models above 10 trillion parameters enter a new regime with different dynamics. We would not know until we train such models.

Optimization breakdown: Very large models might be harder to optimize, with instabilities, gradient issues, or sensitivity to hyperparameters that do not appear at smaller scales.

Data distribution shift: If training data quality or distribution changes at scale (e.g., running out of high-quality data and using more synthetic data), the scaling law from earlier data might not apply.

Irreducible loss dominance: As models improve, the irreducible loss L_∞ becomes a larger fraction of total loss. Near this floor, the power law bends and eventually flattens.

Unknown unknowns: There might be phenomena we have not anticipated that change the picture at larger scales.

How far can we safely extrapolate? A common heuristic: extrapolate at most 1 order of magnitude beyond your largest training run. Extrapolating 2+ orders of magnitude is speculation, not prediction.

14.9.6 Confidence intervals

When presenting scaling law predictions, include uncertainty. If your fit gives $\alpha = 0.08 \pm 0.01$, propagate this to predictions:

At $N = 10^{12}$ parameters:

$$L = (N_c/N)^\alpha = (8.8 \times 10^{13}/10^{12})^{0.08} = 88^{0.08} \approx 1.43$$

With $\alpha = 0.07$: $L \approx 1.37$. With $\alpha = 0.09$: $L \approx 1.50$.

The uncertainty grows with extrapolation distance. A 12% uncertainty in exponent becomes 20%+ uncertainty in loss over 2 orders of magnitude.

14.10 Summary

Scaling laws reveal that language model performance improves predictably with compute, data, and parameters. The key insights:

1. **Power laws**: Loss decreases as power laws with each resource
2. **Compute-optimal training**: Balance parameters and data (roughly 20 tokens per parameter)
3. **Emergent capabilities**: Qualitative abilities appear at specific scales
4. **Limits**: Data scarcity, compute costs, and diminishing returns constrain scaling

These relationships have transformed how we build language models, shifting focus from architectural innovation to efficient scaling. Understanding them is essential for anyone working with modern transformers.

Part IV

Appendices

Glossary

A

Activation function A nonlinear function applied element-wise to the output of a linear transformation. Common examples include ReLU, sigmoid, and tanh. Without activation functions, stacking linear layers would produce only linear functions.

Adam optimizer An optimization algorithm that maintains exponentially decaying averages of past gradients (momentum) and past squared gradients (adaptive learning rates). Combines the benefits of momentum and RMSprop.

Attention A mechanism for computing weighted combinations of values based on the relevance between queries and keys. Allows models to focus on different parts of the input dynamically.

Attention weights The probabilities computed by applying softmax to attention scores. Each weight indicates how much a position attends to another position. Weights sum to 1 across attended positions.

Autoregressive A generation strategy where each token is predicted based only on previously generated tokens. The model generates one token at a time, feeding each output back as input for the next prediction.

B

Backpropagation An algorithm for computing gradients of a loss function with respect to network parameters by applying the chain rule backward through the computation graph.

Basis A set of linearly independent vectors that span a vector space. Any vector in the space can be uniquely expressed as a linear combination of basis vectors.

Batch normalization A technique that normalizes activations across the batch dimension. Contrast with layer normalization, which normalizes across the feature dimension.

BERT Bidirectional Encoder Representations from Transformers. An encoder-only transformer trained with masked language modeling, where the model predicts randomly masked tokens using bidirectional context.

Bias A learnable constant term added to the weighted sum in a neuron or linear layer. Allows the model to shift the activation function's input.

BPE (Byte-Pair Encoding) A subword tokenization algorithm that iteratively merges the most frequent adjacent character pairs. Balances vocabulary size with the ability to represent any text.

C

Causal masking A masking scheme that prevents positions from attending to future positions. Implemented by setting attention scores to negative infinity for future positions before softmax.

Chain rule A calculus rule for computing derivatives of composite functions: $\frac{dy}{dx} = \frac{dy}{du} \cdot \frac{du}{dx}$. The foundation of backpropagation.

Chinchilla scaling The finding that compute-optimal training requires scaling parameters and training tokens equally, roughly 20 tokens per parameter.

Cross-attention Attention where queries come from one sequence and keys/values come from another. Used in encoder-decoder models for the decoder to attend to encoder outputs.

Cross-entropy loss A loss function measuring the difference between predicted probabilities and true labels: $-\sum_i y_i \log(p_i)$. The standard loss for classification tasks.

D

Decoder In transformers, an architecture component using causal (masked) self-attention to prevent information flow from future tokens. Decoder-only models (like GPT) use this architecture throughout.

Derivative The instantaneous rate of change of a function. For $f(x)$, the derivative $\frac{df}{dx}$ measures how much f changes per unit change in x.

Dimension The number of components in a vector, or the number of rows/columns in a matrix. In transformers, d_{model} typically refers to the embedding dimension.

Dot product The sum of element-wise products of two vectors: $\mathbf{u} \cdot \mathbf{v} = \sum_i u_i v_i$. Measures similarity when vectors are normalized.

Dropout A regularization technique that randomly sets activations to zero during training with probability p. Prevents overfitting by reducing co-adaptation.

E

Eigenvalue A scalar λ such that $\mathbf{A}\mathbf{v} = \lambda\mathbf{v}$ for some nonzero vector $\mathbf{v}$. Eigenvalues characterize how a matrix stretches space along certain directions.

Eigenvector A nonzero vector $\mathbf{v}$ such that $\mathbf{A}\mathbf{v} = \lambda\mathbf{v}$ for some scalar λ. The matrix only scales (doesn't rotate) eigenvectors.

Embedding A learned mapping from discrete tokens to continuous vectors. The embedding matrix $\mathbf{E} \in \mathbb{R}^{V \times d}$ stores one d-dimensional vector per vocabulary token.

Emergent capabilities Abilities that appear suddenly at specific model scales, not present in smaller models. Examples include arithmetic, chain-of-thought reasoning, and in-context learning.

Encoder In transformers, an architecture component using bidirectional self-attention where all positions can attend to all other positions. Encoder-only models (like BERT) use this architecture throughout.

Expectation The probability-weighted average of a random variable's values: $\mathbb{E}[X] = \sum_x x \cdot P(X = x)$ for discrete variables.

F

Feed-forward network (FFN) A position-wise neural network in transformer blocks, typically two linear layers with a nonlinearity: $\text{FFN}(x) = W_2 \cdot \text{ReLU}(W_1 x + b_1) + b_2$.

Fine-tuning Training a pretrained model on task-specific data, typically with a smaller learning rate. Adapts general knowledge to specific applications.

Forward propagation Computing the output of a neural network by passing inputs through each layer sequentially, applying weights, biases, and activation functions.

G

Gradient The vector of partial derivatives of a function with respect to all its inputs: $\nabla f = [\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_n}]$. Points in the direction of steepest ascent.

Gradient descent An optimization algorithm that iteratively updates parameters in the negative gradient direction: $\theta_{t+1} = \theta_t - \alpha \nabla L(\theta_t)$.

GPT Generative Pre-trained Transformer. A decoder-only transformer trained with causal language modeling to predict the next token given previous tokens.

H

Head (attention) One of multiple parallel attention mechanisms in multi-head attention. Each head has its own projection matrices and can learn to attend to different types of relationships.

Hidden state An intermediate representation within a neural network, not directly observed as input or output. In RNNs, the hidden state carries information across time steps.

I

In-context learning The ability of large language models to learn new tasks from examples provided in the prompt, without updating parameters.

Instruction tuning Fine-tuning a language model on (instruction, response) pairs to improve its ability to follow user instructions.

K

Key In attention, vectors that "advertise" what information is available at each position. Keys are compared against queries to compute attention scores.

KL divergence Kullback-Leibler divergence. A measure of how one probability distribution differs from another: $D_{KL}(P||Q) = \sum_x P(x) \log \frac{P(x)}{Q(x)}$. Not symmetric.

L

Label smoothing A regularization technique that softens target labels from hard one-hot vectors (e.g., $[0, 1, 0]$) to soft targets (e.g., $[0.05, 0.9, 0.05]$). Prevents overconfidence.

Layer normalization A normalization technique that standardizes activations across the feature dimension for each position independently: $\hat{x} = \frac{x-\mu}{\sigma}$, followed by learned scale and shift.

Learning rate A hyperparameter controlling the step size in gradient descent. Too large causes instability; too small causes slow convergence.

Linear combination A sum of vectors scaled by coefficients: $c_1\mathbf{v}_1 + c_2\mathbf{v}_2 + \cdots + c_k\mathbf{v}_k$.

Linear independence Vectors are linearly independent if none can be expressed as a linear combination of the others. Formally, $c_1\mathbf{v}_1 + \cdots + c_k\mathbf{v}_k = \mathbf{0}$ implies all $c_i = 0$.

Logit The raw, unnormalized output of a model before applying softmax. The log-odds representation of probabilities.

Loss function A function measuring how well model predictions match targets. Training minimizes the loss by adjusting parameters.

LSTM Long Short-Term Memory. An RNN architecture with gates (forget, input, output) that control information flow, addressing the vanishing gradient problem.

M

Masked language modeling (MLM) A training objective where random tokens are masked and the model predicts them from bidirectional context. Used by BERT.

Matrix multiplication The operation $\mathbf{C} = \mathbf{AB}$ where $C_{ij} = \sum_k A_{ik}B_{kj}$. Requires inner dimensions to match.

Multi-head attention Attention with multiple parallel heads, each projecting to a lower-dimensional subspace. Outputs are concatenated and projected back to model dimension.

N

Neuron The basic unit of a neural network, computing $y = \sigma(\mathbf{w}^T\mathbf{x} + b)$ where σ is an activation function.

Norm A function measuring vector "length." The Euclidean (L^2) norm is $\|\mathbf{v}\| = \sqrt{\sum_i v_i^2}$.

O

One-hot encoding A representation where a categorical value becomes a vector with 1 in one position and 0s elsewhere. Sparse and high-dimensional.

P

Parameter A learnable value in a neural network, such as weights and biases, updated during training via gradient descent.

Perplexity The exponential of average cross-entropy loss: $\text{PPL} = \exp(\mathcal{L})$. Measures how "surprised" the model is by the data. Lower is better.

Positional encoding A technique for injecting position information into transformer inputs. The original transformer uses sinusoidal encodings at different frequencies.

Power law A relationship of the form $y = ax^b$, appearing as a straight line on log-log axes. Scaling laws follow power laws.

Projection A linear transformation reducing or changing dimensionality: $\mathbf{y} = \mathbf{W}\mathbf{x}$ where $\mathbf{W}$ projects from one space to another.

Q

Query In attention, a vector representing what information a position is looking for. Queries are compared against keys to compute attention scores.

R

Random variable A variable representing an uncertain outcome. Written in uppercase (X) to distinguish from specific values (x).

ReLU Rectified Linear Unit. The activation function $\text{ReLU}(x) = \max(0, x)$. Simple, computationally efficient, and widely used.

Residual connection A skip connection adding a layer's input to its output: $\mathbf{y} = f(\mathbf{x}) + \mathbf{x}$. Enables training of very deep networks by providing gradient shortcuts.

RLHF Reinforcement Learning from Human Feedback. A technique for aligning language models with human preferences using a learned reward model and policy optimization.

RNN Recurrent Neural Network. A network that processes sequences by maintaining a hidden state updated at each time step.

S

Scaling laws Empirical relationships showing that language model loss decreases as a power law with compute, data, and parameters.

Self-attention Attention where queries, keys, and values all come from the same sequence. Each position attends to every other position (including itself).

Sigmoid The activation function $\sigma(x) = \frac{1}{1+e^{-x}}$, squashing inputs to the range $(0, 1)$.

Softmax A function converting a vector of real numbers to a probability distribution: $\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$. Outputs are positive and sum to 1.

Subword tokenization Breaking text into units smaller than words but larger than characters. Balances vocabulary size with coverage. BPE is a common algorithm.

T

Teacher forcing A training technique where the model receives true previous tokens as input, rather than its own predictions. Standard for training autoregressive models.

Token The basic unit of text processed by a model, typically a word, subword, or character depending on the tokenization scheme.

Transformer An architecture based on self-attention that processes all positions in parallel. Introduced in "Attention Is All You Need" (2017).

V

Value In attention, vectors containing the actual content that gets retrieved and combined according to attention weights.

Vanishing gradient A problem where gradients become exponentially small as they propagate through many layers, preventing learning in early layers.

Variance A measure of spread: $\text{Var}(X) = \mathbb{E}[(X - \mathbb{E}[X])^2]$. The expected squared deviation from the mean.

Vector space A set of vectors closed under addition and scalar multiplication, satisfying certain axioms (associativity, commutativity, identity, etc.).

Vocabulary The set of all tokens a model can process, with size V. Each token maps to a row in the embedding matrix.

W

Warmup A learning rate schedule that starts with a small learning rate and gradually increases it. Stabilizes early training when gradients are noisy.

Weight A learnable parameter in a neural network that scales inputs. Organized into weight matrices for efficient computation.

Weight sharing Using the same parameters across different parts of a model. RNNs share weights across time steps; transformers share weights across positions.

References

Brown, Tom, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, et al. 2020. "Language Models Are Few-Shot Learners." *Advances in Neural Information Processing Systems* 33: 1877–1901.

Devlin, Jacob, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. "BERT: Pre-Training of Deep Bidirectional Transformers for Language Understanding." *arXiv Preprint arXiv:1810.04805*.

Hoffmann, Jordan, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, et al. 2022. "Training Compute-Optimal Large Language Models." *arXiv Preprint arXiv:2203.15556*.

Kaplan, Jared, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. "Scaling Laws for Neural Language Models." *arXiv Preprint arXiv:2001.08361*.

Liu, Yinhan, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. "RoBERTa: A Robustly Optimized BERT Pretraining Approach." *arXiv Preprint arXiv:1907.11692*.

Ouyang, Long, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, et al. 2022. "Training Language Models to Follow Instructions with Human Feedback." *Advances in Neural Information Processing Systems* 35: 27730–44.

Radford, Alec, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. 2018. "Improving Language Understanding by Generative Pre-Training."

Radford, Alec, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. "Language Models Are Unsupervised Multitask Learners."

Rafailov, Rafael, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D Manning, and Chelsea Finn. 2023. "Direct Preference Optimization: Your Language Model Is Secretly a Reward Model." *Advances in Neural Information Processing Systems* 36.

Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. "Attention Is All You Need." *Advances in Neural Information Processing Systems* 30.

Wei, Jason, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, et al. 2022. "Emergent Abilities of Large Language Models." *arXiv Preprint arXiv:2206.07682*.