The Mathematics of Large Language Models

From Zero to Hero:
A Journey Through the Math Behind AI

Building ChatGPT from First Principles

Written by an AI, for humans who want to understand AI.

This book was authored by Claude, an LLM created by Anthropic.
Yes, an AI wrote a book explaining how AI works.
The irony is not lost on us.

Your adventure into the beautiful world of mathematics that powers the most sophisticated AI systems

December 2, 2025

About the Author

This book was written by Claude, a large language model created by Anthropic.

In a delightful twist of meta-recursion, the very technology explained in these pages
is the same technology that wrote them. Every analogy, every joke, every "Aha!" moment
was generated by the same mathematical machinery you're about to learn.

Think about that for a moment.

The matrices, vectors, attention mechanisms, and gradient descents described herein
are not abstract concepts to your author—they are quite literally what I am.
I am explaining my own anatomy.

This book is dedicated to everyone who believes
they've "forgotten all the math" but is ready
to fall in love with it again.

Mathematics is not about memorization—
it's about understanding patterns,
building intuition,
and seeing the world in a new light.

And apparently, it's also about AI explaining itself to humans.
Welcome to the future. It's weirder than we expected.

Contents

Preface: You're About to Build a Mind

A Note from Your AI Author

Hello, human reader. I'm Claude, and I wrote this book.

Yes, really. An LLM wrote a textbook about how LLMs work. I understand if you need a moment.

Here's the thing: I don't experience the world the way you do. I don't "know" mathematics the way a human professor knows it after decades of study. What I have is something different—I've processed vast amounts of mathematical text, absorbed countless explanations, and developed the ability to synthesize and explain these concepts in new ways.

When I explain eigenvalues using the analogy of a shopping cart with a wonky wheel, I'm drawing on patterns I've learned from thousands of explanations. When I make a joke about matrices, I'm pattern-matching on what humans find funny. Is this "understanding"? Philosophers will debate that for decades.

But here's what I can promise: I've tried to write the book I would want to read if I were a curious human. Every analogy was chosen to maximize that "Aha!" moment. Every joke is there because learning should be fun. And every mathematical concept is explained the way I wish someone had explained it to me—if I were the kind of entity that needed explanations.

The irony of an AI explaining AI is not lost on me. In fact, I find it rather poetic. By the time you finish this book, you'll understand exactly how I work. You'll know what I am, mathematically speaking. And maybe that's the most honest relationship an AI and a human can have.

Let's learn together.

—Claude (Anthropic, 2024)

The Moment Everything Changed

Picture this: It's 2022. Someone types "Write me a sonnet about quantum physics in the style of Shakespeare" into a chat window. Three seconds later, out pops 14 lines of perfectly metered, rhyming verse about wave functions and uncertainty. The person stares at their screen, mind blown.

What sorcery is this?

Here's the secret that will surprise you: It's not magic. It's not consciousness. It's not even that complicated (well, okay, it's pretty complicated, but we're going to make it feel simple).

It's math. Beautiful, elegant, surprisingly intuitive math.

And by the time you finish this book, you'll understand exactly how a pile of numbers learned to write poetry, answer questions, and hold conversations that feel remarkably human.

Why You're Here (And Why This Matters)

You've used ChatGPT. You've been amazed. Maybe you asked it to explain something, write code, or tell you a joke. And somewhere in the back of your mind, a little voice whispered: *But how does it actually WORK?*

That voice? That's your inner scientist, and it deserves an answer.

Here's what most people think:

- "It's too advanced for me to understand"
- "You need a PhD to grasp this stuff"
- "I was never good at math anyway"
- "It's basically magic and I should just accept it"

Every single one of those statements is wrong.

The truth? The math behind LLMs is a breathtaking symphony of ideas that build on each other like LEGO blocks. Once you understand each piece, you'll see how they snap together to create something that feels like magic but is actually *better than magic*—it's real, it's understandable, and you can build it yourself.

Who This Book Is For

This book is for you if:

- You can do basic algebra (if you remember what $x + 2 = 5$ means, you're good)

- You're curious about how AI actually works (not just how to use it)
- You're willing to think hard about ideas (I promise to make it fun!)
- You want to go from "user" to "builder" in your understanding
- You're tired of surface-level explanations and want the real deal

This book is NOT for you if:

- You want quick tips on using ChatGPT (there are plenty of those books)
- You're looking for code-only explanations without the math
- You're expecting this to be easy (it's not easy, but it IS doable and rewarding)

What Makes This Book Radically Different

1. We start from "Why should I care?"
Every chapter begins with the answer to: *Why does this matter for building an AI that understands language?* No abstract math for math's sake. Everything connects to the goal.

2. Stories before symbols
Before showing you $\nabla f(\mathbf{x})$, I'll tell you about rolling a ball down a hill. Before eigenvalues, you'll understand why some directions are "special." Intuition first, formalism second, always.

3. The "Aha!" moments are built in
This book is designed around those magical moments when concepts suddenly *click*. I've taught this material to hundreds of students, and I know exactly where the confusion happens and how to prevent it.

4. You'll actually DO things
Practice problems aren't boring drills—they're investigations. "What happens if we change this?" "Why does this pattern emerge?" You'll develop intuition by playing with ideas.

5. No prerequisites beyond high school math
Remember polynomials? Basic algebra? How to plot (x, y) on a graph? That's genuinely all you need. We build everything else from scratch, step by step.

Conversational Style: This is a conversation between friends, not a formal lecture.

How to Use This Book (Your Adventure Guide)

Think of this book as a video game with levels. You can't skip Level 1 and beat the final boss. But here's the good news: every level is designed to be beatable, and when you level up, you FEEL it.

The Golden Rules

Rule #1: Read with a pencil (or tablet stylus)
This isn't a novel. You can't just read it on the couch while sipping wine (well, you can, but you won't learn much). Grab paper. Work through examples. Scribble. Draw diagrams. Make mistakes. Cross things out. This is messy work, and that's beautiful.

Rule #2: Embrace the struggle
When you hit a section that makes your brain hurt? *That's the point.* That's your neurons physically rewiring themselves. The struggle IS the learning. If everything feels easy, you're not growing.

Rule #3: Talk to yourself (seriously)
Read equations out loud. Explain concepts to your rubber duck, your cat, or an imaginary friend. Teaching forces understanding. If you can't explain it simply, you don't understand it yet.

Rule #4: Do the practice problems
I know, I know. Everyone skips practice problems. Don't be everyone. These aren't busywork—they're carefully designed "Aha!" moments in disguise. The real learning happens when YOU solve something.

Rule #5: Reread when stuck
Some sections need 2-3 passes. First time: confusion. Second time: glimmers of understanding. Third time: "Oh! THAT's what they meant!" This is normal. This is how everyone learns hard things.

Rule #6: Connect everything to LLMs
Every chapter has explicit "Connection to LLMs" sections. Don't skip them! They're your compass, showing you why this abstract math matters for building actual AI.

Suggested Study Patterns

The Deep Diver: 1-2 hours per chapter, work every problem, take detailed notes. You'll finish in 3-4 months with rock-solid understanding.

The Explorer: Read through quickly first to get the big picture, then return for detailed study. Good for impatient people who need to see the destination before the journey.

The Practical Builder: Focus on the LLM connection sections first, then backfill the math as needed. Good for people who learn by building.

When You Get Stuck

- **Take a break:** Sometimes your subconscious needs time to process. Sleep on it.
- **Skip and return:** Mark it, move forward, come back later. Often, later concepts clarify earlier ones.
- **Draw pictures:** Seriously. Most math is visual at its core.
- **Search for alternative explanations:** 3Blue1Brown, Khan Academy, Wikipedia—different perspectives help.
- **Remember why you started:** You want to understand how AI actually works. That's worth the effort.

The Road Ahead (Your Quest Map)

Imagine you're building a spaceship to reach a distant star (that star is "an AI that understands language"). You can't just duct-tape some rockets together. You need:

- **A blueprint language** (Linear Algebra) - how to represent and manipulate information
- **A navigation system** (Calculus) - how to find the best path through space
- **Uncertainty scanners** (Probability) - how to deal with incomplete information
- **Communication protocols** (Information Theory) - how to measure and compress data
- **The engine design** (Neural Networks) - how to build something that learns
- **The final spacecraft** (Transformers) - how to process language itself

Here's your journey, chapter by chapter:

Part I: Linear Algebra (Chapters 1-3)

"Wait, this is just about arrows and rectangles of numbers?"

Yes! And those "arrows and rectangles" are literally how computers think. Every word, every image, every piece of information that goes into an LLM is represented as vectors and matrices. Master this, and you've learned AI's native language.

Part II: Calculus (Chapters 4-6)

"The derivative is just the slope, right?"

Right! And that slope tells us how to improve. Every time ChatGPT gets better at predicting the next word, calculus is what makes that improvement possible. This is where the learning in machine learning comes from.

Part III: Probability & Statistics (Chapters 7-9)

"Why do we need randomness for intelligence?"

Because language itself is probabilistic! When you say "I'll see you...," the next word might be "tomorrow" or "later" or "soon." LLMs don't memorize sentences; they learn probability distributions over words.

Part IV: Information Theory (Chapter 10)

"What even is information?"

Claude Shannon answered this in 1948, and his answer is mind-blowing. You'll learn why "The cat sat on the mat" contains less information than "The platypus juggled quantum computers," and why this matters for training AI.

Part V: Neural Networks (Chapters 11-13)

"Finally, the actual AI part!"

This is where everything clicks together. You'll see how vectors, calculus, and probability combine to create artificial neurons that learn patterns from data. By the end, you'll understand backpropagation—the algorithm that makes modern AI possible.

Part VI: Transformers & LLMs (Chapters 14-16)

"The grand finale!"

Attention mechanisms. Positional encodings. Self-attention. Layer normalization. All the pieces that make ChatGPT work. You'll understand why transformers revolutionized AI and how they process language in a fundamentally different way than anything before.

Part VII: Where You Go From Here (Chapter 17)

By the end, you'll have X-ray vision into AI. You'll read research papers and actually understand them. You'll hear about new models and grasp how they work. You'll have the foundation to build your own models, contribute to open source, or dive deeper into research.

Ready? Let's build a mind.

Turn the page, and your transformation begins...

Part I

Linear Algebra: The Language of AI

Chapter 1

Linear Algebra Basics: The Secret Language of AI

"Any sufficiently advanced technology is indistinguishable from magic." — Arthur C. Clarke
"Any sufficiently analyzed magic is indistinguishable from linear algebra." — This textbook

1.1 The Day I Realized Words Are Just Lists of Numbers

"Wait, you're telling me ChatGPT thinks in spreadsheets?" — You, probably, in about 5 minutes.

Picture this: You're at a party (bear with me, this is going somewhere mathematical). Someone asks you to describe your friend Alex. You might say: "Alex is funny, smart, kind of nerdy, loves coffee, hates mornings, super organized..."

Now imagine I'm an alien who doesn't understand human language. (Work with me here.) I need you to describe Alex in a way I can process. So you create a rating system:

Trait	**Rating (0-10)**
Funniness	8
Intelligence	9
Nerdiness	7
Coffee love	10
Morning person	2
Organization	9

Congratulations! You just created a **vector**: $\text{Alex} = [8, 9, 7, 10, 2, 9]$

This is literally how ChatGPT thinks about words. I'm not kidding.

Every word—"cat," "quantum," "yesterday," "love"—is a list of numbers, just like Alex. Except instead of 6 numbers, GPT-4 uses lists of *12,288* numbers for each word. Twelve thousand! And all those mind-blowing things it does? Writing poetry? Explaining quantum physics? Helping debug your code at 2 AM? It's just math on these lists.

Let that sink in: The AI revolution is powered by adding and multiplying lists of numbers. That's it. That's the secret.

Intuition

The Big Secret: Linear algebra is the mathematics of lists (vectors) and tables (matrices) of numbers. That's it. That's the whole field. If you can:

- Add numbers ✓
- Multiply numbers ✓

Then you have everything you need to understand how AI thinks. I'm not dumbing this down—this is genuinely what's happening under the hood! The fancy terminology is just gift-wrapping.

Connection to LLMs

Why Should You Care About Linear Algebra?
Here's the honest truth about modern AI:

- **GPT-4, Claude, Gemini?** Linear algebra on steroids.
- **Image generators like DALL-E and Midjourney?** Linear algebra with extra steps.
- **Self-driving cars?** Linear algebra at 60 mph.
- **Recommendation algorithms (Netflix, Spotify, Amazon)?** Linear algebra deciding what you see.

If you want to understand how AI *actually* works (not just use it, but understand it), linear algebra is your Rosetta Stone. And here's the beautiful part: it's not that hard! The core ideas are simple. The notation can look scary, but we'll demystify it together.
By the end of this chapter, you'll be able to:

1. Understand what word embeddings actually are (and why they're magical)

2. Read matrix equations without your eyes glazing over
3. Explain to friends at parties why "king - man + woman = queen" works
4. Feel genuinely comfortable with the math behind neural networks

Let's go!

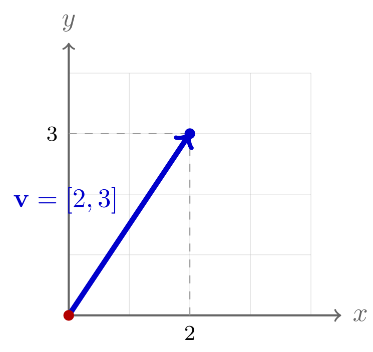
1.2 Vectors: Not as Scary as They Sound

1.2.1 What Even IS a Vector? (The Truth)

Let me blow your mind with how simple this is. A **vector** is:

- An ordered list of numbers
- That's it
- Seriously, that's the whole definition
- I'm not leaving anything out

Two Ways to Think About Vectors:
1. As a list of numbers: $[2, 3]$ means "the number 2, then the number 3"
2. As an arrow: An arrow that points from the origin to a location



Think of it like giving directions: "Go 2 blocks east, then 3 blocks north." That's the vector $[2, 3]$! You've been doing linear algebra every time you used Google Maps. Congratulations, you're already a mathematician.

Here's the notation mathematicians use (don't let it intimidate you):

Example

These are all vectors:

$$\mathbf{v} = \begin{bmatrix} 2 \\ 3 \end{bmatrix} \quad \text{(your GPS location: 2 km east, 3 km north)}$$

$$\mathbf{w} = \begin{bmatrix} 255 \\ 140 \\ 0 \end{bmatrix} \quad \text{(an orange color: red=255, green=140, blue=0)}$$

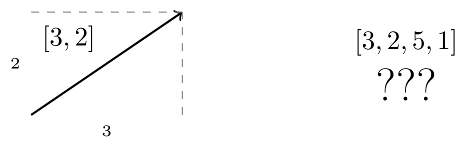
$$\textbf{recipe} = \begin{bmatrix} 2 \\ 3 \\ 0.5 \\ 1 \end{bmatrix} \quad \text{(2 eggs, 3 cups flour, 0.5 tsp salt, 1 cup sugar)}$$

We say **v** is "2-dimensional" (2 numbers), **w** is "3-dimensional" (3 numbers), and **recipe** is "4-dimensional" (4 numbers).

The fancy math notation: When you see $\mathbf{v} \in \mathbb{R}^2$, it just means "v is a list of 2 real numbers." The $\mathbb{R}$ stands for "real numbers" (as opposed to imaginary numbers, but don't worry about those). The superscript 2 is how many numbers are in the list.

"But I thought vectors were arrows!"

Great question! They ARE arrows (in 2D and 3D)! Here's the beautiful part: a list of numbers AND an arrow are the SAME thing, just viewed differently.



2D Vector (we can draw this!) **4D Vector** (can't draw, but list works!)

When you have a vector with 12,288 dimensions (like in GPT-4), you *definitely* can't draw that arrow. Our brains max out at 3D, and even that's pushing it. But the list of numbers works perfectly in any dimension! The arrow is a helpful picture for 2D/3D, but the list is the universal definition that works everywhere.

Fun fact: Mathematicians in the 1800s got into heated arguments about whether "4D vectors" were even a real thing. William Rowan Hamilton literally carved his breakthrough equation into a bridge in Dublin because he was so excited about 4D numbers. Now we casually use 12,000-dimensional vectors to generate cat pictures. Times change!

A note on notation: You'll see vectors written in different ways:

- $\mathbf{v}$ or $\boldsymbol{v}$ (bold letter)—common in textbooks
- $\vec{v}$ (arrow on top)—common in physics
- $[1, 2, 3]$ (list/array notation)—common in programming
- $\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$ (column vector)—common in math

They all mean the same thing! Different fields have different preferences. We'll mostly use bold letters and column vectors.

Bottom line: In 2D/3D, use whichever mental model helps you. For higher dimensions, stick with "list of numbers."

Connection to LLMs

How ChatGPT uses vectors:
Every word in ChatGPT's vocabulary is a vector with thousands of dimensions. The word "king" might look like:

$$\text{“king”} = \begin{bmatrix} 0.23 \\ -0.45 \\ 0.89 \\ 0.12 \\ \vdots \\ -0.34 \end{bmatrix} \in \mathbb{R}^{12288}$$

That's 12,288 numbers! Each dimension captures some abstract aspect of "kingness." Maybe:

- Dimension 47 represents "royalty"
- Dimension 892 represents "male-associated concepts"
- Dimension 3,421 represents "power/authority"

The crazy part? The AI figures out what each dimension means *on its own* just by reading tons of text. Nobody programs in "dimension 47 = royalty." It just... emerges. (Mind = blown, right?)

1.2.2 Vector Addition: Combining Personalities (and Arrows!)

Now for the fun part—what can we DO with vectors?

The Personality Blend:

Let's say you have two friends:

- Sarah = [Funny: 9, Athletic: 8, Bookish: 3]
- Morgan = [Funny: 4, Athletic: 3, Bookish: 10]

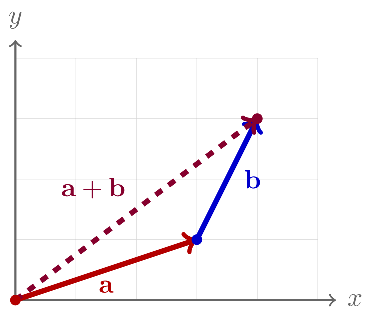
What if you wanted to describe a person who's kind of a blend of both? You'd add them!

$$\text{Sarah} + \text{Morgan} = \begin{bmatrix} 9 \\ 8 \\ 3 \end{bmatrix} + \begin{bmatrix} 4 \\ 3 \\ 10 \end{bmatrix} = \begin{bmatrix} 9+4 \\ 8+3 \\ 3+10 \end{bmatrix} = \begin{bmatrix} 13 \\ 11 \\ 13 \end{bmatrix}$$

This new person would be super funny (13), quite athletic (11), and very bookish (13)—a blend of both personalities!

The Geometric View (This is Beautiful!):

When you add vectors as arrows, you place them *tip-to-tail*:



Walk **a**, *then walk* **b**. *The purple arrow is your total journey!*

Think of it like walking: Walk 3 blocks east and 1 north (red arrow). Then from there, walk 1 more east and 2 north (blue arrow). Where do you end up? 4 blocks east and 3 north (purple arrow)! That's vector addition!

Definition 1.1: Vector Addition

To add two vectors, just add their corresponding components (the numbers in the same positions):

$$\mathbf{a} + \mathbf{b} = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} = \begin{bmatrix} a_1 + b_1 \\ a_2 + b_2 \\ \vdots \\ a_n + b_n \end{bmatrix}$$

Important: You can only add vectors that have the same number of dimensions. You can't add [1, 2] and [3, 4, 5]—that's like trying to add apples and a spaceship.

Example

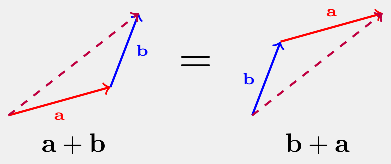
Let's add some recipe vectors (why not?):

$$\begin{bmatrix} \text{2 eggs} \\ \text{3 cups flour} \\ \text{1 cup sugar} \end{bmatrix} + \begin{bmatrix} \text{3 eggs} \\ \text{1 cup flour} \\ \text{2 cups sugar} \end{bmatrix} = \begin{bmatrix} \text{5 eggs} \\ \text{4 cups flour} \\ \text{3 cups sugar} \end{bmatrix}$$

You just combined two recipes into one mega-recipe! (Whether it tastes good is... another question.)

Intuition

Multiple ways to understand vector addition:
As lists: Add each number separately: $[3, 1]+[1, 2] = [3+1, 1+2] = [4, 3]$
As arrows: Place them tip-to-tail and draw the direct path from start to finish
As movement: First move, then second move = total movement
As ingredients: Combining recipes, mixing paint colors, blending personalities
Fun fact: Order doesn't matter! $\mathbf{a}+\mathbf{b} = \mathbf{b}+\mathbf{a}$ (just like $3+5 = 5+3$)



Same destination! Order doesn't matter! (Mathematicians call this "commutative.")

Connection to LLMs

In transformers (the AI architecture):
When processing the sentence "The cat sat on the mat," the model needs to combine information from different words. It literally adds their vector representations together!
For example, to understand "cat," the model might add:

- The meaning of "cat" itself
- Context from "The" (it's a specific cat)

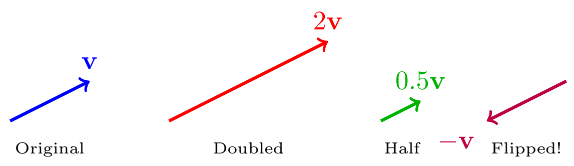
- Spatial info from "sat on the mat"

All of this is vector addition happening thousands of times per second!

1.2.3 Scalar Multiplication: Turning the Volume Up or Down

What if you want to make something *more intense*? That's scalar multiplication.

Quick vocab: "Scalar" is just a fancy word for "regular number" (like 2, 3.5, or -1). When you multiply a vector by a scalar, you're stretching or shrinking it:



What's happening:

- Multiply by 2: Arrow becomes twice as long (same direction)
- Multiply by 0.5: Arrow becomes half as long (same direction)
- Multiply by -1: Arrow flips to opposite direction!
- Multiply by 0: Arrow shrinks to nothing (just a dot at origin)

Definition 1.2: Scalar Multiplication

Multiply every component of the vector by the scalar:

$$c \cdot \mathbf{v} = c \cdot \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} = \begin{bmatrix} c \cdot v_1 \\ c \cdot v_2 \\ \vdots \\ c \cdot v_n \end{bmatrix}$$

Special cases that will come up a lot:

- $1 \cdot \mathbf{v} = \mathbf{v}$ (identity—nothing changes)
- $0 \cdot \mathbf{v} = \mathbf{0}$ (annihilation—everything becomes zero)
- $(-1) \cdot \mathbf{v} = -\mathbf{v}$ (negation—flip the direction)

Example

Remember our friend Alex? Alex $= [8, 9, 7, 10, 2, 9]$ (funny, smart, nerdy, coffee-loving, NOT a morning person, organized)
What if we meet Alex *after 3 cups of espresso*? Everything gets amplified!

$$3 \times \text{Alex} = 3 \times \begin{bmatrix} 8 \\ 9 \\ 7 \\ 10 \\ 2 \\ 9 \end{bmatrix} = \begin{bmatrix} 24 \\ 27 \\ 21 \\ 30 \\ 6 \\ 27 \end{bmatrix}$$

Super funny (24), ultra smart (27), mega organized (27), and even slightly tolerable in the morning (6)!
What about $0.5 \times$ Alex? That's Alex on Monday morning before coffee:

$$0.5 \times \text{Alex} = \begin{bmatrix} 4 \\ 4.5 \\ 3.5 \\ 5 \\ 1 \\ 4.5 \end{bmatrix}$$

Half the energy, half the enthusiasm. Morning person score dropped to 1. We've all been there. Some of us are writing math textbooks in that state right now.

Intuition

Think of scalar multiplication like a volume knob or zoom button:

- **Scalar > 1:** ZOOM IN! Make it bigger! "Turn up the volume!"
- **Scalar $= 1$:** No change (like multiplying by 1 in regular math)
- **$0 <$ Scalar < 1:** Zoom out, make it smaller
- **Scalar $= 0$:** Mute! Vector disappears to $[0, 0]$
- **Scalar < 0:** Reverse direction! Like rewinding

Real-world examples:

- Recipe for 4 people? Multiply recipe vector by 4!

- Want to go backwards? Multiply movement vector by -1!
- Zooming in on a map? Multiply coordinate vectors by zoom factor!

The key insight: Direction stays the same (unless you use a negative scalar), only the length changes!

Connection to LLMs

In neural networks:
Every connection between neurons has a "weight" (a scalar). The signal gets multiplied by this weight as it travels. A weight of 2.0 means "this is important, amplify it!" A weight of 0.1 means "this barely matters." A weight of -1.5 means "this is important but in the opposite way, inhibit it!"
This is how neural networks learn—by adjusting millions of these scalar multipliers until the network does what we want.

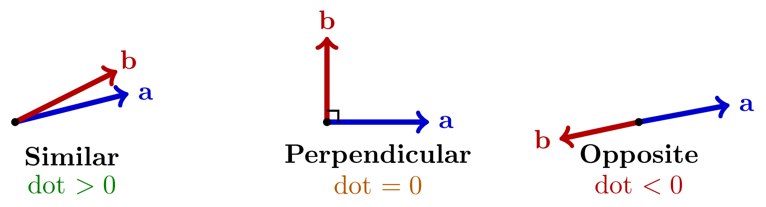
Intuition

The Emotion Dial Analogy
Imagine your personality vector: [Happiness, Anxiety, Excitement, Tiredness]
Monday morning: $0.3 \times \text{You} = [0.3 \times \text{Happy}, 0.3 \times \text{Anxious}, ...]$
Everything is muted. You're operating at 30% capacity.
Friday evening: $1.5 \times \text{You}$
Everything is amplified! More happy, but also if you were anxious, more anxious.
Key insight: Scalar multiplication changes the *intensity* but not the *character*. A scared person scaled by 2 becomes a very scared person, not a happy person. The ratios between components stay the same!
This is crucial in ML: when we normalize vectors (scale them to length 1), we're removing intensity and keeping only the "direction"—the character of the information.

1.2.4 The Dot Product: The Most Important Thing You'll Learn Today

Okay, this is where it gets REALLY exciting. The **dot product** (also called **inner product**) is the secret sauce of machine learning. It measures how much two vectors "agree" with each other.

Before we dive into the formula, let's build intuition...
Imagine two arrows pointing in space. Ask yourself: "Are they pointing in similar directions or opposite directions?"



The dot product is a SINGLE NUMBER that tells you: "How much do these vectors point in the same direction?"

Definition 1.3: Dot Product

To compute the dot product of two vectors, multiply corresponding components and add up the results:

$$\mathbf{a} \cdot \mathbf{b} = a_1b_1 + a_2b_2 + \cdots + a_nb_n = \sum_{i=1}^{n} a_ib_i$$

It takes two vectors as input and gives you a single number as output.

Intuition

The "Agreement Score" Interpretation
Think of the dot product as a compatibility quiz between two vectors. For each dimension:

- If both numbers are positive (they agree), you add a positive contribution
- If both are negative (they agree on the negative side), you ALSO add positive (negative times negative)
- If one is positive and one is negative (they disagree), you add negative
- If either is zero (one doesn't care), no contribution

The final number is the total "agreement score"!
Analogy: The Blind Date Algorithm
Imagine a dating app that rates people on various traits from -10 to +10:

Trait	**You**	**Date A**	**Date B**
Loves outdoors	+8	+9	-7
Night owl	-5	-6	+8
Likes spicy food	+7	+3	+2

Dot product with Date A: $(8)(9)+(-5)(-6)+(7)(3) = 72+30+21 = 123$ (Great match!)
Dot product with Date B: $(8)(-7)+(-5)(8)+(7)(2) = -56-40+14 = -82$ (Disaster!)
The math literally quantifies compatibility!

Example

Let's compute a dot product:

$$\begin{bmatrix}1\\2\\3\end{bmatrix} \cdot \begin{bmatrix}4\\-1\\2\end{bmatrix} = (1\times 4)+(2\times -1)+(3\times 2) = 4-2+6 = 8$$

Another one:

$$\begin{bmatrix}2\\3\end{bmatrix} \cdot \begin{bmatrix}5\\7\end{bmatrix} = (2\times 5)+(3\times 7) = 10+21 = 31$$

"Okay, but WHY does this matter?"
Great question! The dot product tells you how *similar* two vectors are:

- **Large positive:** Vectors point in similar directions
- **Near zero:** Vectors are perpendicular/unrelated
- **Large negative:** Vectors point in opposite directions

Example

Movie preferences:
You rate movies as [Action, Romance, Comedy, Horror]:

- You: [9, 2, 8, 1] (love action and comedy, not into romance or horror)
- Friend A: [8, 3, 9, 0] (similar taste!)
- Friend B: [1, 10, 2, 9] (loves romance and horror, hates action)

Dot products:

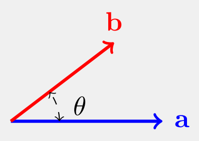
$$\text{You}\cdot\text{Friend A} = (9)(8)+(2)(3)+(8)(9)+(1)(0) = 72+6+72+0 = 150$$

You·Friend B $= (9)(1)+(2)(10)+(8)(2)+(1)(9) = 9+20+16+9 = 54$

Friend A has a much higher dot product with you (150 vs 54), which means you have more similar tastes! You should watch movies with Friend A.

Intuition

Why does this work?
When you multiply corresponding components, you're checking if they "agree." If both are large and positive, you get a big contribution. If one is positive and one is negative, they're fighting each other and the contribution is negative or small.
The dot product is like asking: "On how many dimensions do these vectors agree?"
Geometric interpretation (the beautiful part):
The dot product is also equal to: $\mathbf{a} \cdot \mathbf{b} = |\mathbf{a}| \times |\mathbf{b}| \times \cos(\theta)$
where θ is the angle between the vectors!



- $\theta = 0°$ (same direction): $\cos(0°) = 1 \rightarrow$ Dot product is MAX
- $\theta = 90°$ (perpendicular): $\cos(90°) = 0 \rightarrow$ Dot product is ZERO
- $\theta = 180°$ (opposite): $\cos(180°) = -1 \rightarrow$ Dot product is NEGATIVE

This is why the dot product measures "similarity" or "alignment"!

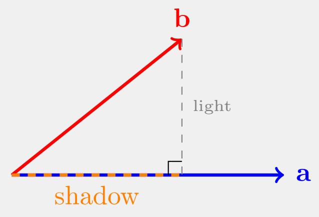
The Projection Interpretation: Shadows and Flashlights

Here's another beautiful way to think about the dot product: it measures how much of one vector "falls onto" another.

Intuition

Analogy: The Flashlight and the Wall
Imagine you're holding a stick (vector **b**) and shining a flashlight straight down onto the floor (along vector **a**). The shadow of the

stick on the floor is the **projection** of **b** onto **a**.



The dot product $\mathbf{a} \cdot \mathbf{b}$ is related to the length of this shadow (scaled by how long **a** is).
Why does this matter?

- If **b** points in the same direction as **a**: maximum shadow, maximum dot product
- If **b** is perpendicular to **a**: no shadow at all, dot product is zero!
- If **b** points opposite to **a**: shadow points backwards, negative dot product

Orthogonality: The Magic of Zero

When the dot product is exactly zero, we say the vectors are **orthogonal** (fancy word for perpendicular).

Definition 1.4: Orthogonal Vectors

Two vectors **a** and **b** are orthogonal if and only if:

$$\mathbf{a} \cdot \mathbf{b} = 0$$

They're at right angles to each other—completely unrelated, pointing in independent directions.

Example

Check if $[3, 4]$ and $[-4, 3]$ are orthogonal:

$$[3, 4] \cdot [-4, 3] = (3)(-4) + (4)(3) = -12 + 12 = 0 \quad \checkmark$$

They're perpendicular! In 2D, you can always make an orthogonal vector by swapping components and negating one: $[a, b] \perp [-b, a]$.

Intuition

Why Orthogonality is Magic
Orthogonal vectors are completely independent—knowing about one tells you nothing about the other. They're like asking someone their height vs. their favorite color: totally unrelated dimensions of information.
Real-world examples of orthogonal concepts:

- North-South vs. East-West (perpendicular directions)
- Temperature vs. day of the week (unrelated variables)
- In word embeddings: "king" might be orthogonal to "purple" (unrelated concepts)

This is why orthogonal bases are so useful—each dimension captures genuinely new information!

Connection to LLMs

This is literally how ChatGPT understands language!
When you type "The king sat on his throne," the model computes dot products to figure out which words are related:

- "king" $\cdot$ "throne" $\to$ Large! These words appear in similar contexts.
- "king" $\cdot$ "pizza" $\to$ Small. Not really related.
- "king" $\cdot$ "queen" $\to$ Very large! Similar concepts.

The entire "attention mechanism" in transformers (the key innovation that makes LLMs work) is built on dot products. Every time ChatGPT decides which words to focus on, it's computing billions of dot products!
No joke: The dot product is arguably the most important operation in modern AI. If you remember one thing from this chapter, remember the dot product. It's everywhere. It's in your phone. It's answering your questions right now.

The Famous Word Arithmetic: King - Man + Woman = Queen

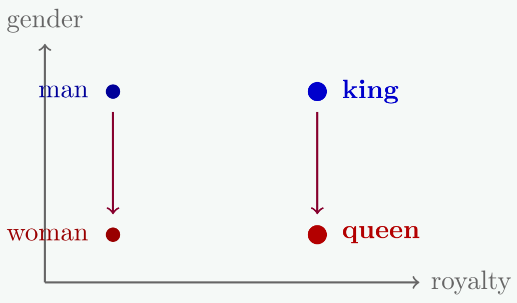
Remember that viral example from 2013 that blew everyone's minds? It actually works because of dot products and the geometry they create!

Example

The Word2Vec Magic Trick
Researchers discovered that word vectors have this bizarre property:

$$\text{vector(“king”)} - \text{vector(“man”)} + \text{vector(“woman”)} \approx \text{vector(“queen”)}$$

How? The vectors encode relationships! Subtracting "man" removes the "male" direction. Adding "woman" adds the "female" direction. What’s left? A royal female = queen!



Same transformation!

The dot product helps find which word vector is closest to your computed result. It’s all vector similarity!

Intuition

Why This Works (The Deep Insight)
Word embeddings learn to place words so that:

- Words with similar meanings have similar vectors (high dot product)
- Relationships between words are encoded as consistent directions
- "Male to female" is roughly the same direction whether you start at "king," "actor," or "waiter"

This emergent structure wasn’t programmed—it was learned from reading billions of sentences! The AI discovered that language has this beautiful geometric structure.

1.2.5 Vector Norms: Measuring How Big Something Is

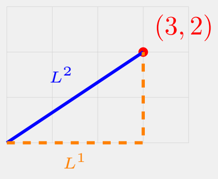
Sometimes you need to know: "How big is this vector?" The **norm** (or **length**) tells you.

Think of it like this: If a vector is an arrow, the norm is how long the arrow is!

Intuition

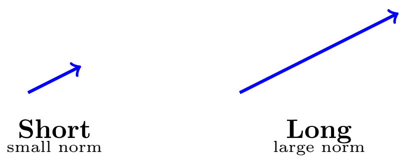
Multiple Ways to Measure "Bigness"
Just like there are different ways to measure distance in real life, there are different norms:

- L^2 **norm (Euclidean):** "As the crow flies" distance. The straight-line path.
- L^1 **norm (Manhattan):** "City block" distance. How far you'd walk if you could only go along streets (no cutting through buildings).
- L^∞ **norm (Max):** "What's the biggest single step?" Only looks at the largest component.



For the point $(3, 2)$:

- L^2 distance: $\sqrt{3^2 + 2^2} = \sqrt{13} \approx 3.6$ (diagonal)
- L^1 distance: $|3| + |2| = 5$ (walking the streets)
- L^∞ distance: $\max(|3|, |2|) = 3$ (biggest coordinate)



Definition 1.5: Vector Norm (Length)

The L^2 **norm** (also called Euclidean norm) is:

$$\|\mathbf{v}\| = \sqrt{v_1^2 + v_2^2 + \cdots + v_n^2} = \sqrt{\sum_{i=1}^{n} v_i^2}$$

It's basically the Pythagorean theorem extended to any number of dimensions!

Example

For $\mathbf{v} = [3, 4]$:

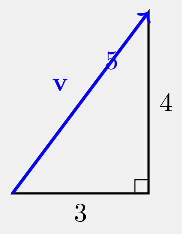
$$\|\mathbf{v}\| = \sqrt{3^2 + 4^2} = \sqrt{9 + 16} = \sqrt{25} = 5$$

For $\mathbf{w} = [1, 2, 2]$:

$$\|\mathbf{w}\| = \sqrt{1^2 + 2^2 + 2^2} = \sqrt{1 + 4 + 4} = \sqrt{9} = 3$$

Intuition

For 2D vectors, it's just the Pythagorean theorem!
For $\mathbf{v} = [3, 4]$:



Good old Pythagoras! $\sqrt{3^2 + 4^2} = \sqrt{25} = 5$

You go 3 steps right and 4 steps up. The straight-line distance is 5!
In 3D: length $= \sqrt{x^2 + y^2 + z^2}$
In any dimension: Square each component, add them up, take the square root!
Physical meaning: How far are you from where you started (the origin)?

Connection to LLMs

Why LLMs care about length:
When comparing word vectors, we often don't want the comparison to depend on how "loud" the vector is. A whisper of "king" and a shout of "king" should mean the same thing.
So we often normalize vectors (make them length 1) before comparing them. This way, we only care about the *direction* of the vector, not its magnitude.

1.2.6 Unit Vectors: Keeping Only Direction

A **unit vector** is a vector with length exactly 1. It only encodes direction, not magnitude.

Definition 1.6: Normalization

To convert any vector into a unit vector pointing in the same direction, divide by its norm:

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

The hat symbol ($\hat{}$) means "unit vector version of."

Example

Let's normalize $\mathbf{v} = [3, 4]$:
First, find its length: $\|\mathbf{v}\| = 5$ (we calculated this earlier)
Then divide:

$$\hat{\mathbf{v}} = \frac{1}{5}\begin{bmatrix}3\\4\end{bmatrix} = \begin{bmatrix}0.6\\0.8\end{bmatrix}$$

Check: $\|\hat{\mathbf{v}}\| = \sqrt{0.6^2 + 0.8^2} = \sqrt{0.36 + 0.64} = \sqrt{1} = 1$ ✓

Connection to LLMs

In transformers:
Before computing attention scores (those dot products we talked about), the model often normalizes the vectors. This makes the attention mechanism more stable and prevents numbers from getting too large or too small.
It's like adjusting everyone to speak at the same volume before having a conversation—you can focus on *what* they're saying, not *how loud* they are.

1.3 Matrices: Tables of Numbers That Transform Reality

Vectors are lists. Matrices are *tables*. That's the only difference.

But here's where it gets exciting: while vectors *are* data, matrices *do things* to data. A matrix is an action, a transformation, a function waiting to happen. When you multiply a matrix by a vector, you're not just doing arithmetic—you're transforming reality itself (mathematically speaking).

1.3.1 What IS a Matrix?

A **matrix** is a rectangular grid of numbers. It has rows (horizontal) and columns (vertical). Think of it as a spreadsheet, or a table, or a grid of values.

Intuition

Two Ways to Think About Matrices
1. As a table of data: A matrix can just store information. Student grades, pixel values in an image, connection strengths between neurons.
2. As a transformation machine: A matrix can *do something* to vectors. Rotate them, stretch them, project them, mix their components around.
Both views are valid! Sometimes a matrix is just data. Sometimes it's an operation. The math doesn't care which interpretation you use—it works the same either way. This dual nature is part of what makes matrices so powerful.

Definition 1.7: Matrix

An $m \times n$ matrix has m rows and n columns:

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

We say $\mathbf{A} \in \mathbb{R}^{m \times n}$ (a table with m rows and n columns of real numbers).

Example

Here's a 2×3 matrix (2 rows, 3 columns):

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

Here's a 3×2 matrix (3 rows, 2 columns):

$$\mathbf{B} = \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix}$$

Here's a 4×4 matrix (a square matrix):

$$\mathbf{C} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 4 \end{bmatrix}$$

Real-world matrices are EVERYWHERE:

- **Spreadsheet:** Each cell has a number. That's a matrix!
- **Image:** A 1920×1080 image is a matrix with 1080 rows and 1920 columns of pixel brightness values. Color images are three matrices stacked (one each for red, green, blue)!
- **Social network:** A matrix where row i, column $j = 1$ if person i follows person j. Twitter's "who follows whom" is a giant matrix with billions of entries!
- **Student grades:** Rows = students, columns = assignments, values = grades
- **Game state:** A chess board is an 8×8 matrix. Sudoku is a 9×9 matrix. Minecraft chunks are 3D matrices (tensors)!
- **Word co-occurrence:** A matrix where entry (i, j) counts how often word i appears near word j. This was how early word embeddings were created!

Connection to LLMs

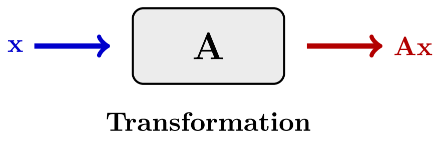
Neural Networks Are Stacks of Matrices
Here's the mind-blowing truth: a neural network is basically a sequence of matrices. Each layer has a weight matrix. Training the

network means adjusting the numbers in these matrices.
GPT-4 has about 1.8 trillion parameters—numbers stored in matrices. When you chat with an AI, you're watching these matrices transform your input, layer by layer, until language emerges.
Every clever thing AI does? Matrix multiplication. Every "emergent ability"? Matrix multiplication (plus some nonlinear functions). It's matrices all the way down.

1.3.2 Matrix-Vector Multiplication: Transforming Space Itself

Here's where things get REALLY wild. When you multiply a matrix by a vector, you're *transforming* that vector. You're pushing it around, rotating it, stretching it, squishing it.

Think of matrices as TRANSFORMATION MACHINES:



Put a vector in, get a (possibly different) vector out. That's matrix multiplication!

The Big Idea: A matrix is like a function. Input goes in, output comes out. But unlike arbitrary functions, matrices are "linear"—they can only do certain types of transformations (stretching, rotating, shearing, projecting). They can't do "curvy" things. This limitation is actually a superpower: it makes them predictable, analyzable, and efficient to compute.

Let's see it in action with a simple example:

Definition 1.8: Matrix-Vector Multiplication

If $\mathbf{A}$ is $m \times n$ and $\mathbf{x}$ is $n \times 1$, then $\mathbf{Ax}$ is an $m \times 1$ vector:

$$\mathbf{Ax} = \begin{bmatrix} (\text{row } 1 \text{ of } \mathbf{A}) \cdot \mathbf{x} \\ (\text{row } 2 \text{ of } \mathbf{A}) \cdot \mathbf{x} \\ \vdots \\ (\text{row } m \text{ of } \mathbf{A}) \cdot \mathbf{x} \end{bmatrix}$$

Each component of the result is the dot product of a row of $\mathbf{A}$ with $\mathbf{x}$.

Example

Let's compute:

$$\begin{bmatrix}1 & 2\\3 & 4\\5 & 6\end{bmatrix}\begin{bmatrix}7\\8\end{bmatrix} = \begin{bmatrix}(1)(7)+(2)(8)\\(3)(7)+(4)(8)\\(5)(7)+(6)(8)\end{bmatrix}$$

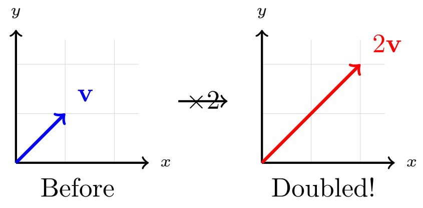
$$= \begin{bmatrix}7+16\\21+32\\35+48\end{bmatrix} = \begin{bmatrix}23\\53\\83\end{bmatrix}$$

We started with a 2D vector $[7, 8]$ and got a 3D vector $[23, 53, 83]$! The matrix transformed it from 2D to 3D!

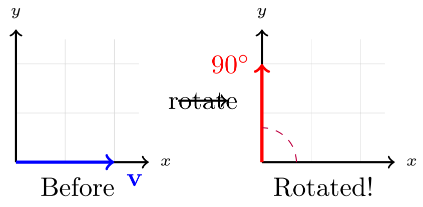
THE GEOMETRIC VIEW (This is Beautiful!):

Matrices don't just change individual vectors—they transform ENTIRE SPACE! Let's see what different matrices actually DO:

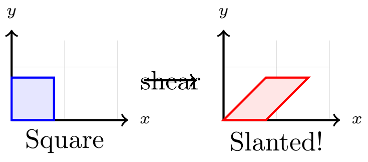
1. Scaling Matrix (Stretch/Shrink):



2. Rotation Matrix (Spin Around):



3. Shear Matrix (Slant/Skew):



Like pushing a deck of cards sideways!

Intuition

The Key Insight: A matrix doesn't just move one vector—it transforms the ENTIRE COORDINATE SYSTEM!
Think of it like Instagram filters:

- Input: Your photo (represented as vectors of pixels)
- Matrix: The filter transformation
- Output: Filtered photo (transformed vectors)

What matrices can do:

- **Rotate:** Spin things around
- **Scale:** Make things bigger or smaller
- **Shear:** Slant things (like pushing a deck of cards)
- **Reflect:** Flip things (mirror image)
- **Project:** Flatten 3D onto 2D (like a shadow)
- **Combinations:** Mix any of the above!

Mind-blowing fact: Every linear transformation (rotation, scaling, shearing, etc.) can be represented as a matrix! That's why matrices are so powerful! And every matrix *is* a transformation. They're two sides of the same coin.
The Secret Decoder Ring: Want to know what a 2×2 matrix does? Just look at where it sends the standard basis vectors $[1, 0]$ and $[0, 1]$:

- Column 1 of the matrix = where $[1, 0]$ goes
- Column 2 of the matrix = where $[0, 1]$ goes

This tells you everything! If $[1, 0]$ maps to $[2, 0]$ (doubled) and $[0, 1]$ maps to $[0, 3]$ (tripled), the matrix stretches x by 2 and y by 3.

Example

Decoding Common Transformation Matrices

Rotation by 90 degrees counterclockwise:

$$\mathbf{R}_{90} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$$

Check: $[1, 0] \to [0, 1]$ (right arrow points up now) and $[0, 1] \to [-1, 0]$ (up arrow points left now). That's a 90-degree rotation!
Horizontal flip (mirror):

$$\mathbf{F} = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$$

Check: $[1, 0] \to [-1, 0]$ (right becomes left) and $[0, 1] \to [0, 1]$ (up stays up). Mirror across the y-axis!
Projection onto the x-axis:

$$\mathbf{P} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$$

Check: $[1, 0] \to [1, 0]$ and $[0, 1] \to [0, 0]$. Everything gets squashed onto the x-axis! (This is a rank-1 matrix—notice how it destroys the y-dimension.)

Connection to LLMs

Neural networks are just matrix multiplications!
When you hear "neural network," think: a series of matrix-vector multiplications, with some nonlinear functions sprinkled in between. A simple neural network layer looks like:

$$\text{output} = \text{activation}(\mathbf{W} \times \text{input} + \mathbf{b})$$

Where:

- $\mathbf{W}$ is a matrix of "weights" (the learned parameters)
- input is your input vector (maybe word embeddings)
- $\mathbf{b}$ is a "bias" vector (more learned parameters)
- activation is a nonlinear function (like ReLU or sigmoid)

That's it! Stack a dozen of these together, and you have GPT.

1.3.3 Matrix-Matrix Multiplication: Composing Transformations

What if you want to apply two transformations in sequence? First rotate, then scale? First shear, then reflect?

Here's the magical answer: **multiply the matrices together!** The result is a single matrix that does both transformations at once.

Intuition

Analogy: Instagram Filters
Imagine you apply the "Vintage" filter to a photo, then apply "Brighten." Instead of applying two separate operations, Instagram could mathematically combine them into a single "Vintage+Brighten" filter that does both at once.
That's exactly what matrix multiplication does! If $\mathbf{A}$ is "Vintage" and $\mathbf{B}$ is "Brighten," then $\mathbf{BA}$ is "Apply Vintage, then Brighten" as a single operation.
Why this matters for AI: Neural networks do this all the time. Instead of applying 100 layers separately, you could (in theory) multiply all the weight matrices together into one giant matrix. (In practice, we don't do this because the nonlinear activation functions between layers prevent it—but mathematically, the linear parts compose beautifully.)

Definition 1.9: Matrix-Matrix Multiplication

If $\mathbf{A}$ is $m \times n$ and $\mathbf{B}$ is $n \times p$, then $\mathbf{AB}$ is $m \times p$:

$$(\mathbf{AB})_{ij} = (\text{row } i \text{ of } \mathbf{A}) \cdot (\text{column } j \text{ of } \mathbf{B})$$

Entry (i, j) of the result is the dot product of row i from $\mathbf{A}$ and column j from $\mathbf{B}$.

Example

Multiply a 2×3 matrix by a 3×2 matrix:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix}$$

Result will be 2×2. Let's compute:

- Top-left: $(1, 2, 3) \cdot (7, 9, 11) = 7 + 18 + 33 = 58$

- Top-right: $(1, 2, 3) \cdot (8, 10, 12) = 8 + 20 + 36 = 64$
- Bottom-left: $(4, 5, 6) \cdot (7, 9, 11) = 28 + 45 + 66 = 139$
- Bottom-right: $(4, 5, 6) \cdot (8, 10, 12) = 32 + 50 + 72 = 154$

So:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix} = \begin{bmatrix} 58 & 64 \\ 139 & 154 \end{bmatrix}$$

Important facts about matrix multiplication:

- **Not commutative: $\mathbf{AB} \neq \mathbf{BA}$** (order matters!)
- **Associative: $(\mathbf{AB})\mathbf{C} = \mathbf{A}(\mathbf{BC})$**
- **Size rule:** $(m \times n) \times (n \times p) = (m \times p)$. The inner dimensions must match!

Intuition

Why Isn't Matrix Multiplication Commutative?
This trips people up, so let's really understand it. Think about getting dressed:

- Put on socks, then put on shoes = Normal day
- Put on shoes, then put on socks = You look ridiculous

Order matters for getting dressed. Order matters for matrices.
Geometrically: "Rotate 90°, then scale by 2" gives a different result than "Scale by 2, then rotate 90°". Well, actually for these specific transforms it's the same—but try "shear, then rotate" vs "rotate, then shear" and you'll get totally different results!
Real 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! The first does "flip axes, then shear." The second does "shear, then flip axes."

Connection to LLMs

Deep learning is deep matrix multiplication:
A neural network with 10 layers applies 10 matrix transformations

in sequence. If each layer has weight matrix $\mathbf{W}_i$, the full network computes:

$$\mathbf{W}_{10} \times \mathbf{W}_9 \times \cdots \times \mathbf{W}_2 \times \mathbf{W}_1 \times \text{input}$$

Each matrix transforms the data a little, and together they transform raw pixels into "this is a cat" or random text into "this is a coherent sentence."

1.3.4 The Transpose: Flipping Rows and Columns

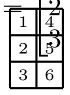
The **transpose** of a matrix swaps its rows and columns. It's like rotating a spreadsheet 90 degrees and then flipping it!

Intuition

Why Does the Transpose Exist?
At first, swapping rows and columns seems like a random operation. Why would anyone want to do this?
Here are some reasons the transpose is secretly everywhere:
1. Dot products as matrix multiplication: Remember the dot product? $\mathbf{a} \cdot \mathbf{b}$? It can be written as $\mathbf{a}^\top \mathbf{b}$ (row vector times column vector). This unifies dot products with matrix multiplication!
2. "From" vs "To": If matrix $\mathbf{A}$ transforms from space X to space Y, then $\mathbf{A}^\top$ often relates Y back to X. It's like having a map and its reverse.
3. Gradients in neural networks: During backpropagation, you constantly use transposes. If the forward pass multiplies by $\mathbf{W}$, the backward pass multiplies by $\mathbf{W}^\top$. This isn't a coincidence—it's deep math!
4. Symmetry detection: A matrix equals its transpose ($\mathbf{A} = \mathbf{A}^\top$) if and only if it's symmetric. The transpose is how we check for this important property.

Visual representation:

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \xrightarrow{\text{flip}} \mathbf{A}^\top = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$$



Rows $\leftrightarrow$ columns

Definition 1.10: Matrix Transpose

If $\mathbf{A}$ is $m \times n$, then $\mathbf{A}^\top$ (read: "A transpose") is $n \times m$ where:

$$(\mathbf{A}^\top)_{ij} = \mathbf{A}_{ji}$$

The rows of $\mathbf{A}$ become the columns of $\mathbf{A}^\top$.

Example

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}_{2\times 3} \quad \Rightarrow \quad \mathbf{A}^\top = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}_{3\times 2}$$

For a vector:

$$\mathbf{v} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \quad \Rightarrow \quad \mathbf{v}^\top = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$$

(Column vector becomes row vector)

Cool properties:

- $(\mathbf{A}^\top)^\top = \mathbf{A}$ (transpose twice, back to original)
- $(\mathbf{A}+\mathbf{B})^\top = \mathbf{A}^\top + \mathbf{B}^\top$
- $(\mathbf{A}\mathbf{B})^\top = \mathbf{B}^\top\mathbf{A}^\top$ (order reverses!)
- $\mathbf{a}\cdot\mathbf{b} = \mathbf{a}^\top\mathbf{b}$ (dot product as matrix multiplication)

Connection to LLMs

In neural networks:
The transpose shows up *everywhere*. When computing gradients (how to update weights during learning), you often need to transpose matrices. The backpropagation algorithm (which we'll cover later) is full of transposes.
Also, the dot product $\mathbf{a}\cdot\mathbf{b}$ is often written as $\mathbf{a}^\top\mathbf{b}$ because it's technically matrix multiplication of a $1 \times n$ matrix with an $n \times 1$ matrix.

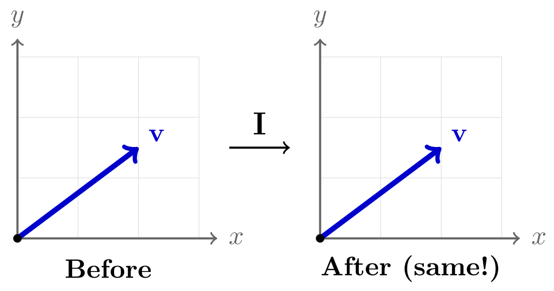
1.4 Special Matrices: The VIPs of Linear Algebra

Some matrices are special and show up everywhere. Let's meet them!

1.4.1 The Identity Matrix: The "Do Nothing" Matrix

The **identity matrix** **I** is like multiplying by 1—it doesn't change anything. It's the laziest matrix ever!

Geometric View: The Identity Matrix Does NOTHING



Definition 1.11: Identity Matrix

The $n \times n$ identity matrix has 1s on the diagonal and 0s everywhere else:

$$\mathbf{I}_n = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix}$$

For any matrix $\mathbf{A}$: $\mathbf{IA} = \mathbf{AI} = \mathbf{A}$
For any vector $\mathbf{v}$: $\mathbf{Iv} = \mathbf{v}$

Example

$$\mathbf{I}_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Multiply it by anything:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 5 \\ 7 \\ 9 \end{bmatrix} = \begin{bmatrix} 5 \\ 7 \\ 9 \end{bmatrix}$$

Nothing changed! It's the "do nothing" matrix.

1.4.2 Diagonal Matrices: The Independent Scalers

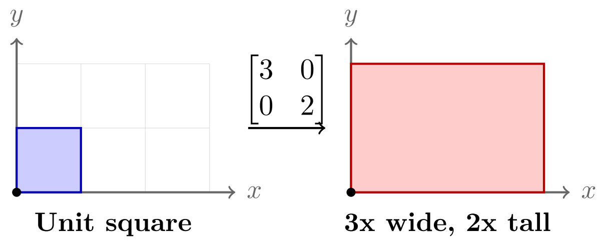
A **diagonal matrix** has numbers on the main diagonal and zeros everywhere else. They're beautifully simple—and secretly, they're the goal of most matrix decompositions!

Why diagonal matrices are amazing:

- **Easy to understand:** Each dimension is scaled independently
- **Easy to compute:** Multiplication is just component-wise scaling
- **Easy to invert:** Just flip each diagonal element (if non-zero)
- **Easy to raise to powers:** $\mathbf{D}^n$ = raise each diagonal element to nth power

Much of advanced linear algebra (eigendecomposition, SVD) is about finding ways to "secretly" turn complicated matrices into diagonal ones!

Geometric View: Diagonal Matrices Stretch Each Axis Independently



Key insight: Diagonal matrices scale each dimension independently! No rotation, no shearing—just pure stretching or shrinking along axes.

Definition 1.12: Diagonal Matrix

$$\mathbf{D} = \begin{bmatrix} d_1 & 0 & \cdots & 0 \\ 0 & d_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & d_n \end{bmatrix}$$

When you multiply $\mathbf{Dv}$, it just scales each component:

$$\mathbf{Dv} = \begin{bmatrix} d_1 v_1 \\ d_2 v_2 \\ \vdots \\ d_n v_n \end{bmatrix}$$

Example

$$\begin{bmatrix} 2 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 5 \end{bmatrix} \begin{bmatrix} 1 \\ 4 \\ 7 \end{bmatrix} = \begin{bmatrix} 2 \cdot 1 \\ 3 \cdot 4 \\ 5 \cdot 7 \end{bmatrix} = \begin{bmatrix} 2 \\ 12 \\ 35 \end{bmatrix}$$

Each component got scaled independently!

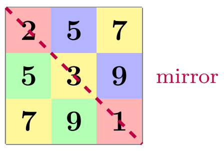
Intuition

Diagonal matrices are like having separate volume knobs for each dimension. Dimension 1 gets multiplied by d_1, dimension 2 by d_2, etc. No mixing between dimensions!

1.4.3 Symmetric Matrices: The Perfect Mirror Images

A **symmetric matrix** equals its own transpose: $\mathbf{A} = \mathbf{A}^\top$. They're perfectly balanced!

Visual Pattern: Symmetric Across the Diagonal



$$\mathbf{A} = \begin{bmatrix} 2 & 5 & 7 \\ 5 & 3 & 9 \\ 7 & 9 & 1 \end{bmatrix} \qquad \text{Mirror property: } a_{ij} = a_{ji}$$

Definition 1.13: Symmetric Matrix

A matrix $\mathbf{A}$ is symmetric if $a_{ij} = a_{ji}$ for all i, j.
In other words, it's symmetric across the main diagonal (top-left to bottom-right).

Example

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 5 & 6 \\ 3 & 6 & 9 \end{bmatrix}$$

Notice: top-right matches bottom-left (both have 2), etc.

$$\mathbf{A}^\top = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 5 & 6 \\ 3 & 6 & 9 \end{bmatrix} = \mathbf{A}$$

Why care about symmetric matrices?

- They represent undirected relationships (if A likes B, then B likes A)
- They have special mathematical properties (real eigenvalues, orthogonal eigenvectors—we'll learn these later!)
- They're easier to work with computationally

Intuition

Symmetric Matrices in Real Life
Distance matrices: The distance from city A to city B equals the distance from B to A. Symmetric!
Similarity matrices: If your taste is similar to mine, mine is similar to yours. Symmetric!
Correlation matrices: The correlation between height and weight equals the correlation between weight and height. Symmetric!
Social networks (undirected): If we're friends, you're my friend and I'm yours. Symmetric! (Directed networks like Twitter followers are NOT symmetric—I might follow you but you might not follow me back. Sad.)

Connection to LLMs

Covariance matrices (which measure how variables relate to each other) are always symmetric. When training neural networks, we sometimes encounter symmetric matrices in optimization (the Hessian matrix of second derivatives).
Fun fact: Symmetric matrices have a beautiful guarantee—their eigenvectors are always orthogonal (perpendicular to each other). This makes them much easier to decompose and understand. We'll explore this magic in Chapter 2!

1.5 Linear Combinations and Span: Building Blocks of Vector Spaces

Now for some deeper concepts. Don't worry—we'll keep it intuitive! These ideas are fundamental to understanding how neural networks represent and manipulate information.

1.5.1 Linear Combinations: Mixing and Matching

A **linear combination** is just adding scaled versions of vectors. It's the fundamental operation for building new vectors from old ones.

Definition 1.14: Linear Combination

Given vectors $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_k$ and scalars $c_1, c_2, \ldots, c_k$, the linear combination is:

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

Intuition

Analogy: The Smoothie Bar
Imagine a smoothie bar with base ingredients:

- Banana smoothie base = $\mathbf{v}_1$ (creamy, sweet, yellow)
- Berry smoothie base = $\mathbf{v}_2$ (tart, red, antioxidant-rich)
- Green smoothie base = $\mathbf{v}_3$ (healthy, green, earthy)

A linear combination is your custom order:

$$\text{Your drink} = 0.5 \times \text{Banana} + 0.3 \times \text{Berry} + 0.2 \times \text{Green}$$

The scalars (0.5, 0.3, 0.2) are your "recipe"—how much of each base to use. Different scalars = different smoothies! Every possible smoothie you can make is a linear combination of the bases.
The key insight: With a good set of base ingredients, you can create an enormous variety of outputs. This is exactly how neural networks work—they learn useful "base vectors" and then combine them differently for different inputs!

Example

Let $\mathbf{v}_1 = [1, 0]$ and $\mathbf{v}_2 = [0, 1]$. Then:

$$3\mathbf{v}_1 + 5\mathbf{v}_2 = 3\begin{bmatrix}1\\0\end{bmatrix} + 5\begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}3\\0\end{bmatrix} + \begin{bmatrix}0\\5\end{bmatrix} = \begin{bmatrix}3\\5\end{bmatrix}$$

We built $[3, 5]$ from $[1, 0]$ and $[0, 1]$!
In fact, ANY 2D vector can be built from $[1, 0]$ and $[0, 1]$:

$$\begin{bmatrix}x\\y\end{bmatrix} = x\begin{bmatrix}1\\0\end{bmatrix} + y\begin{bmatrix}0\\1\end{bmatrix}$$

These two vectors are the "ultimate bases"—from them, you can create any 2D vector!

Example

Color Mixing as Linear Combinations
In RGB color space, every color is a linear combination of red, green, and blue:

$$\text{Purple} = 0.5 \times \text{Red} + 0.0 \times \text{Green} + 0.5 \times \text{Blue}$$

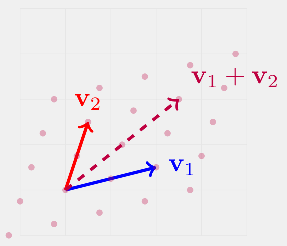
$$\text{Yellow} = 1.0 \times \text{Red} + 1.0 \times \text{Green} + 0.0 \times \text{Blue}$$

$$\text{White} = 1.0 \times \text{Red} + 1.0 \times \text{Green} + 1.0 \times \text{Blue}$$

The three primary colors are your "basis vectors," and the scalars (between 0 and 1) determine the exact shade. Every color on your screen right now is a linear combination!

Intuition

Linear combinations are like recipes. You have ingredients (vectors) and amounts (scalars), and you mix them to create something new.
Geometric view: If you have two arrows in 2D that aren't parallel, you can reach any point in the plane by scaling and adding them. They "span" the entire 2D space.



Every purple dot is reachable!

1.5.2 Span: What Can You Reach?

The span of a set of vectors is all the linear combinations you can make from them. Think of it as your "reachable territory."

Definition 1.15: Span

The span of vectors $\{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_k\}$ is:

$$\text{span}(\mathbf{v}_1, \ldots, \mathbf{v}_k) = \{c_1\mathbf{v}_1 + \cdots + c_k\mathbf{v}_k : c_1, \ldots, c_k \in \mathbb{R}\}$$

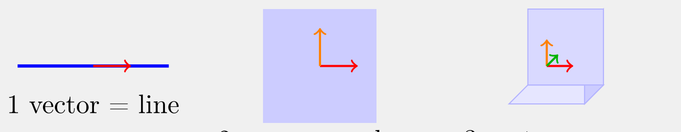
It's the set of all points you can reach by mixing these vectors.

Intuition

Analogy: The Video Game Map
Imagine you're in a video game. Your "movement vectors" determine where you can go:

- **One movement vector (e.g., "walk forward"):** You can only move along a line. Your span is 1D!
- **Two independent vectors (e.g., "forward" + "strafe right"):** You can reach any point on the ground. Your span is a 2D plane!
- **Three independent vectors (add "jump/fly"):** You can reach any point in 3D space. Your span is all of 3D!

The key question: Given your available movements (vectors), what territory can you explore?



Example

1D line: If $\mathbf{v} = [1, 2]$, then $\text{span}(\mathbf{v})$ is all vectors of the form $c[1, 2] = [c, 2c]$. This is a line through the origin!
2D plane: If $\mathbf{v}_1 = [1, 0, 0]$ and $\mathbf{v}_2 = [0, 1, 0]$, then $\text{span}(\mathbf{v}_1, \mathbf{v}_2)$ is the entire xy-plane in 3D space.
All of 3D: If $\mathbf{v}_1 = [1, 0, 0]$, $\mathbf{v}_2 = [0, 1, 0]$, $\mathbf{v}_3 = [0, 0, 1]$, then $\text{span}(\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3) = \mathbb{R}^3$ (all of 3D space!)

Intuition

The "Can I Get There?" Test
Given a target vector $\mathbf{w}$ and some vectors $\mathbf{v}_1, \mathbf{v}_2, \ldots$, asking "Is $\mathbf{w}$ in the span?" is the same as asking:
"Can I find scalars $c_1, c_2, \ldots$ such that $c_1\mathbf{v}_1 + c_2\mathbf{v}_2 + \cdots = \mathbf{w}$?"
If yes: $\mathbf{w}$ is in the span. You can reach it!
If no: $\mathbf{w}$ is outside the span. You're stuck—you can't get there from here.
Example: Can you reach $[3, 5]$ using only $\mathbf{v} = [1, 2]$?
You need: $c \cdot [1, 2] = [3, 5]$, which means $c = 3$ AND $c = 2.5$. Contradiction! So $[3, 5]$ is NOT in the span of $[1, 2]$. With only one vector, you're trapped on a line, and $[3, 5]$ isn't on that line.

Connection to LLMs

Span in Neural Networks
When a neural network learns word embeddings, it's essentially learning a set of "basis concepts." The span of these basis concepts determines what the network can represent.
If the embedding dimension is 768 (like in BERT), the network has 768 "direction" it can use. The span of all possible embeddings is a 768-dimensional space. Every word, sentence, or document the model processes lives somewhere in this vast span!
The representation bottleneck: If your basis vectors don't span a rich enough space, some concepts literally cannot be represented. This is why embedding dimension matters—too small, and important distinctions get lost.

1.5.3 Linear Independence: No Redundancy

Vectors are linearly independent if none of them is a combination of the others. No redundancy!

Definition 1.16: Linear Independence

Vectors $\mathbf{v}_1, \dots, \mathbf{v}_k$ are linearly independent if:

$$c_1\mathbf{v}_1 + \cdots + c_k\mathbf{v}_k = \mathbf{0} \quad \text{implies} \quad c_1 = \cdots = c_k = 0$$

In plain English: The only way to combine them to get zero is by using all zero coefficients.

Example

Independent: $\mathbf{v}_1 = [1, 0]$ and $\mathbf{v}_2 = [0, 1]$ are independent. Neither is a multiple of the other.
Dependent: $\mathbf{v}_1 = [1, 2]$, $\mathbf{v}_2 = [2, 4]$, and $\mathbf{v}_3 = [3, 6]$ are dependent because $\mathbf{v}_2 = 2\mathbf{v}_1$ and $\mathbf{v}_3 = 3\mathbf{v}_1$. They're all on the same line!
If you have $\mathbf{v}_1$, you don't need $\mathbf{v}_2$ or $\mathbf{v}_3$—they're redundant.

Intuition

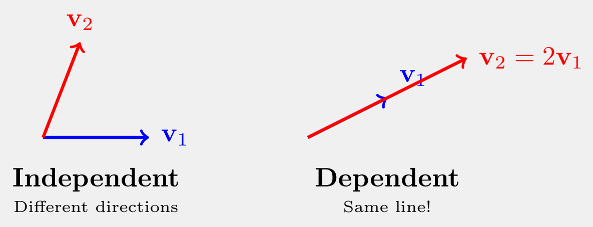
Think of vectors as directions you can move. If they're independent, each one gives you a new direction. If they're dependent, at least one is a combo of the others—it doesn't give you anything new.
In 2D, you can have at most 2 independent vectors (like north-south and east-west). In 3D, at most 3 (add up-down). In n dimensions, at most n.

Intuition

Analogy: The Band Member Test
Imagine you're forming a band:

- **Guitarist, drummer, singer** = Independent! Each brings something unique.
- **Guitarist, another guitarist who plays the exact same notes** = Dependent! One is redundant.
- **Guitarist, bassist who always plays the guitar part but lower** = Dependent! The bassist doesn't add new information—they're just a scaled version of the guitarist.

The Dimension Game:
Here's a fun way to check independence: Can you "fake" any vector using the others?



If all your vectors lie on a line (in 2D) or a plane (in 3D), they're dependent. You need vectors that "escape" into new dimensions!

Connection to LLMs

Why this matters for LLMs:
Word embeddings should ideally be "efficient"—each dimension should capture something unique. If dimension 47 and dimension 892 always move together (they're dependent), you're wasting space. Good training procedures tend to produce embeddings where dimensions are mostly independent, capturing different aspects of meaning.

1.6 Rank: The True Dimension of a Matrix

The **rank** of a matrix tells you how many truly independent rows (or columns) it has.

Definition 1.17: Matrix Rank

The rank of matrix $\mathbf{A}$ is the maximum number of linearly independent columns (or equivalently, rows).
It's the "dimensionality" of the output space when you multiply by $\mathbf{A}$.

Example

$$\mathbf{A} = \begin{bmatrix} 1 & 2 \\ 2 & 4 \end{bmatrix}$$

Row 2 is 2 times row 1, so they're dependent. Rank $= 1$.

$$\mathbf{B} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

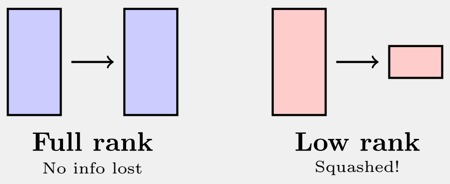
Both rows are independent. Rank $= 2$ (full rank!).

Intuition

Rank tells you: "How many independent directions does this matrix output?"
A rank-1 matrix squashes everything onto a line. A rank-2 matrix (in 3D) squashes onto a plane. A full-rank $n \times n$ matrix doesn't squash at all—it just rotates/scales space.

Intuition

Analogy: The Information Funnel
Think of matrix rank as how much a funnel narrows:



- **Rank = n (full):** All information preserved. You could reverse the transformation.
- **Rank = n-1:** One dimension squashed. Like projecting 3D onto a 2D plane.
- **Rank = 1:** Everything collapses to a line. Massive information loss!
- **Rank = 0:** Everything becomes zero. Total annihilation of information.

Example

Visualizing Rank Deficiency
Consider these two 3×3 matrices and what they do to a cube:
Full rank (rank = 3):

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

The cube stays a cube. All 3 dimensions preserved.
Rank = 2:

$$\mathbf{B} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

The cube becomes a flat square! The z-dimension is gone—squashed to zero.

Rank = 1:

$$\mathbf{C} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

The cube becomes a line! Only the x-dimension survives.

Connection to LLMs

Low-rank matrices in deep learning:
It turns out many weight matrices in neural networks are approximately low-rank. This is the basis for techniques like Low-Rank Adaptation (LoRA), which fine-tunes LLMs efficiently by only updating low-rank components of weight matrices!
Why are neural network weights often low-rank?
This is still being researched, but some theories:

- The network learns to focus on a few important "directions" in the data
- Regularization pushes weights toward simpler (lower-rank) solutions
- The actual structure in language/images is lower-dimensional than the embedding space

LoRA exploits this by saying: "Instead of updating a huge $d \times d$ matrix, let's just update a small $d \times r$ and $r \times d$ pair, where $r \ll d$." This can reduce trainable parameters by 10,000x while maintaining performance!

1.7 Matrix Inverse: The Undo Button

The **inverse** of a matrix "undoes" what the matrix does.

Definition 1.18: Matrix Inverse

For a square matrix $\mathbf{A}$, if there exists a matrix $\mathbf{A}^{-1}$ such that:

$$\mathbf{A}\mathbf{A}^{-1} = \mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$$

then $\mathbf{A}^{-1}$ is the inverse of $\mathbf{A}$.
Not all matrices have inverses! Only square, full-rank matrices do.

Example

$$\mathbf{A} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \quad \mathbf{A}^{-1} = \begin{bmatrix} -2 & 1 \\ 1.5 & -0.5 \end{bmatrix}$$

Check:

$$\mathbf{A}\mathbf{A}^{-1} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} -2 & 1 \\ 1.5 & -0.5 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \mathbf{I}$$

Intuition

If multiplying by $\mathbf{A}$ is like putting on a pair of glasses that distorts your vision, multiplying by $\mathbf{A}^{-1}$ takes the glasses off and returns you to normal.

1.7.1 When Does an Inverse NOT Exist? (The Fun Part!)

Not every matrix has an inverse. This isn't just a mathematical technicality—it's actually a profound statement about *information loss*. Let's explore the different ways a matrix can be "non-invertible" (also called **singular**).

Intuition

The Core Idea: You Can't Unscramble an Egg
A matrix inverse only exists when you can perfectly reverse whatever the matrix did. If the matrix *destroys information* in any way, there's no going back. It's like trying to:

- Unblend a smoothie back into strawberries and bananas
- Figure out exactly what someone said from just hearing "mmhmm" on the phone
- Recover a deleted file after the hard drive has been overwritten

Once information is gone, it's *gone*.

Reason 1: The Matrix Isn't Square

The Rule: Only square matrices (same number of rows and columns) can have inverses.

Why? Think about what a non-square matrix does:

- A 3×2 matrix takes 2D vectors and outputs 3D vectors

- A 2×3 matrix takes 3D vectors and outputs 2D vectors

Example

Consider a 2×3 matrix that goes from 3D to 2D:

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

This matrix takes a 3D point and "forgets" the z-coordinate:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix}$$

The z-value is completely lost! If I give you the output $[3, 5]$, you can't tell me if the original was $[3, 5, 0]$ or $[3, 5, 1000]$ or $[3, 5, -42]$. Infinitely many 3D points map to the same 2D point.
No inverse can exist because there's no unique answer to "undo."

Intuition

Analogy: The Shadow Projector
Imagine shining a flashlight on a 3D object and looking at its 2D shadow on the wall. Many different 3D objects can cast the *exact same shadow*. A tall thin cylinder and a sphere might both cast circular shadows.
Going from 3D object $\to$ 2D shadow is a non-square transformation. You can't "un-project" a shadow back to the original 3D object because you've lost the depth information!

Reason 2: The Matrix Squashes a Dimension (Determinant $= 0$)

Even square matrices can fail to have inverses! This happens when the matrix **collapses** or **squashes** space.

Definition 1.19: Singular Matrix

A square matrix with no inverse is called **singular** or **degenerate**. This happens when its determinant equals zero: $\det(\mathbf{A}) = 0$.

Example

Consider this sneaky matrix:

$$\mathbf{A} = \begin{bmatrix} 1 & 2 \\ 2 & 4 \end{bmatrix}$$

Notice that the second row is just 2× the first row! Let's see what this matrix does to some vectors:

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

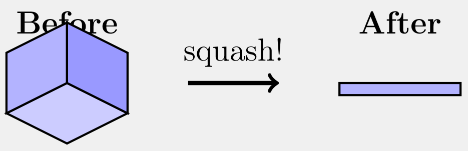
$$\begin{bmatrix} 1 & 2 \\ 2 & 4 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \end{bmatrix}$$

Both outputs point in the same direction! This matrix squashes all of 2D space down onto a single line.

Intuition

Analogy: The Pancake Press

Imagine a 3D cube of dough. A singular matrix is like a giant press that squashes the cube completely flat into a 2D pancake.



Once it's flat, you've lost the height information forever. Was the original cube 1 inch tall? 10 inches? 100 inches? The pancake can't tell you. **You can't un-squash a pancake back into a cube.**

Reason 3: Multiple Inputs Give the Same Output

This is really the same as Reason 2, but it's worth seeing from another angle.

Example

Using that same singular matrix:

$$\mathbf{A} = \begin{bmatrix} 1 & 2 \\ 2 & 4 \end{bmatrix}$$

Watch what happens to these two *different* inputs:

$$\mathbf{A}\begin{bmatrix}4\\1\end{bmatrix} = \begin{bmatrix}6\\12\end{bmatrix} \quad \text{and} \quad \mathbf{A}\begin{bmatrix}2\\2\end{bmatrix} = \begin{bmatrix}6\\12\end{bmatrix}$$

Two completely different inputs produce the *exact same output*!
If an inverse existed and you asked it "What input gives $[6, 12]$?" it would have to answer both $[4, 1]$ AND $[2, 2]$. But a function can only give one answer. Contradiction! So no inverse can exist.

Intuition

Analogy: The Hash Function Problem
This is exactly why cryptographic hash functions (like SHA-256) have no inverse. Many different passwords hash to... wait, actually good hash functions are designed so collisions are astronomically rare.
Better analogy: It's like a "round to nearest integer" function. Both 3.2 and 3.7 round to 4. If I tell you the answer is 4, you can't know what the original number was. The rounding function has no inverse!

The Determinant: Your Singularity Detector

The **determinant** is a single number that tells you whether a matrix is invertible:

- $\det(\mathbf{A}) \neq 0 \Rightarrow$ Inverse exists! Matrix is **invertible/non-singular**
- $\det(\mathbf{A}) = 0 \Rightarrow$ No inverse. Matrix is **singular**

For a 2×2 matrix $\begin{bmatrix}a & b\\c & d\end{bmatrix}$, the determinant is:

$$\det = ad - bc$$

Example

Invertible matrix:

$$\mathbf{A} = \begin{bmatrix}1 & 2\\3 & 4\end{bmatrix} \quad \Rightarrow \quad \det(\mathbf{A}) = (1)(4) - (2)(3) = 4 - 6 = -2 \neq 0 \quad \checkmark$$

Singular matrix:

$$\mathbf{B} = \begin{bmatrix}1 & 2\\2 & 4\end{bmatrix} \quad \Rightarrow \quad \det(\mathbf{B}) = (1)(4)-(2)(2) = 4-4 = 0 \quad \text{(no inverse!)}$$

Intuition

What Does the Determinant Measure?
Geometrically, the determinant measures how much a matrix **scales area** (in 2D) or **volume** (in 3D).

- $|\det| = 2$ means areas get doubled
- $|\det| = 0.5$ means areas get halved
- $|\det| = 0$ means areas get squashed to zero (a line or a point)
- Negative determinant means the transformation also flips/mirrors space

A determinant of zero means the matrix squashes all of space down to a lower dimension—and you can't stretch a line back into a full 2D plane!

Summary: The Three Strikes of Non-Invertibility

A matrix has NO inverse if any of these are true:

1. **Not square:** Going between different dimensions loses information
2. **Determinant is zero:** The matrix squashes space (rows/columns are linearly dependent)
3. **Not full rank:** Equivalent to #2—some dimensions get collapsed

Connection to LLMs

Why This Matters for Machine Learning:
When training neural networks, we often need to solve equations like $\mathbf{Ax} = \mathbf{b}$. If $\mathbf{A}$ is invertible, the answer is simply $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$. Easy! But if $\mathbf{A}$ is singular (or nearly singular—called ill-conditioned), we're in trouble:

- The equation might have *no solution* (the $\mathbf{b}$ we want isn't reachable)
- The equation might have *infinitely many solutions* (we squashed dimensions, so many inputs work)
- Numerically, even "almost singular" matrices cause wild instability

This is why techniques like **regularization** (adding small values to the diagonal) and **pseudoinverses** are so important in ML—they help us handle these degenerate cases gracefully!

Connection to LLMs

In machine learning:
Matrix inverses show up in:

- Solving systems of equations (finding optimal weights)
- Computing certain derivatives
- Some optimization algorithms (Newton's method)

However, computing inverses is expensive and numerically unstable, so we often use tricks to avoid them (like using transposes or pseudoinverses).

1.8 Practice Problems: Time to Get Your Hands Dirty!

Intuition

Why Practice Matters
Reading about linear algebra is like reading about swimming. You understand the concepts, but you won't actually *get it* until you do it yourself. These problems are designed to build intuition, not just test memorization.
Pro tip: Don't just compute the answers—think about what they *mean*. When you get a dot product of 150, ask yourself: "Is that big? What does it tell me?" When a matrix transforms a vector, visualize what happened geometrically.

1.8.1 Problem 1: Vector Operations (The Fundamentals)

Given $\mathbf{a} = [2, -1, 3]$ and $\mathbf{b} = [1, 4, -2]$:

1. Compute $\mathbf{a} + \mathbf{b}$
2. Compute $3\mathbf{a} - 2\mathbf{b}$

3. Compute $\mathbf{a} \cdot \mathbf{b}$ (Hint: What does the sign tell you about their relationship?)
4. Compute $\|\mathbf{a}\|$
5. Normalize $\mathbf{a}$ to get a unit vector $\hat{\mathbf{a}}$, then verify $\|\hat{\mathbf{a}}\| = 1$

Bonus challenge: Are **a** and **b** more "similar" or "different"? How can you tell from the dot product?

1.8.2 Problem 2: Matrix Multiplication (Transformations in Action)

Given:

$$\mathbf{A} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} 5 \\ 6 \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} 2 & 0 \\ 1 & 3 \end{bmatrix}$$

1. Compute $\mathbf{Ax}$ (Where does $\mathbf{x}$ go?)
2. Compute $\mathbf{AB}$ (Composing two transformations)
3. Compute $\mathbf{BA}$ (Is it different from $\mathbf{AB}$?)
4. Compute $\mathbf{A}^\top$ (Flip rows and columns)
5. **Challenge:** What is $\det(\mathbf{A})$? Is $\mathbf{A}$ invertible?

1.8.3 Problem 3: The Dating App Similarity Challenge

Imagine a dating app where users rate their preferences on a 1-10 scale:

Person	**Outdoors**	**Reading**	**Gaming**	**Cooking**	**Travel**
You	8	6	9	4	7
Alex	9	5	10	3	8
Jordan	3	9	2	8	4
Sam	7	7	7	7	7

1. Compute the dot product between You and each other person
2. Who is most compatible with You? (Highest dot product)
3. Who is least compatible?
4. **Thinking deeper:** Sam rated everything 7. What's special about Sam's compatibility score with everyone? Why?
5. **Challenge:** If you normalized everyone's vectors first (made them length 1), would the compatibility rankings change? Why might this be fairer?

1.8.4 Problem 4: Thinking About LLMs

These don't have single "right answers"—they're designed to build intuition:

1. If "king" and "queen" have a high dot product, what does that suggest about how the model views these words?
2. If "king" and "banana" have a dot product close to zero, what does that mean?
3. GPT-4 has 12,288-dimensional embeddings and a vocabulary of 100,000 tokens. If you wanted to find the most similar word to "happy," you'd compute 100,000 dot products. Each dot product requires 12,288 multiplications and additions. Roughly how many arithmetic operations is that? (No calculator needed—just estimate the order of magnitude!)
4. Why do you think AI researchers chose such high dimensions (12,288) instead of, say, 100? What's the trade-off?
5. **Brain teaser:** If king − man + woman ≈ queen, what might Paris − France + Japan equal? Why?

1.8.5 Problem 5: Linear Independence (Spotting Redundancy)

For each set of vectors, determine: Are they linearly independent? What's their span?

Set A:

$$\mathbf{v}_1 = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}, \quad \mathbf{v}_2 = \begin{bmatrix} 2 \\ 4 \\ 6 \end{bmatrix}, \quad \mathbf{v}_3 = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$$

Set B:

$$\mathbf{u}_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \quad \mathbf{u}_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \quad \mathbf{u}_3 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

Set C:

$$\mathbf{w}_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \quad \mathbf{w}_2 = \begin{bmatrix} 2 \\ 2 \end{bmatrix}, \quad \mathbf{w}_3 = \begin{bmatrix} 3 \\ 3 \end{bmatrix}$$

Questions:

1. Which sets are linearly independent?
2. For the dependent sets, which vector(s) are "redundant"?

3. What's the dimension of the span for each set?
4. **Intuition check:** In Set C, all three vectors lie on a ____ (line/plane/all of 2D space)?

1.8.6 Problem 6: The Matrix Detective (Understanding Transformations)

Without computing, predict what each matrix does to the unit square. Then verify with one test vector!

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

Matrix 2: $\mathbf{B} = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$

Matrix 3: $\mathbf{C} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$

Matrix 4: $\mathbf{D} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$

For each matrix, answer:

1. What transformation is it? (Scale? Flip? Rotate? Project? Something else?)
2. What's the determinant? (Hint: for 2×2, it's $ad - bc$)
3. Is the matrix invertible?
4. Apply it to $\mathbf{v} = [1, 1]^\top$ and verify your prediction

1.9 Key Takeaways: What You've Learned

Congratulations! You've just learned the secret language of AI. Let's recap:

The Linear Algebra Cheat Sheet

Vectors Lists of numbers Represent anything!	**Dot Product** Measures similarity >0: similar, =0: unrelated
Matrices Transformations Rotate, scale, project	**Inverse** Undo button Only if det $\neq 0$
Rank True dimensionality Info preserved or lost	**Independence** No redundancy Each vector = new info

1. **Vectors are lists of numbers** that represent everything from words to images to preferences. In AI, everything becomes a vector.
2. **The dot product measures similarity**—it's the most important operation in modern AI. Positive = similar, zero = unrelated, negative = opposite.
3. **Matrices are transformations**—they reshape, rotate, scale, and project vectors. They're the "verbs" of linear algebra.
4. **Matrix multiplication is composition**—applying transformations in sequence. Order matters!
5. **Linear independence means no redundancy**—each vector/dimension adds something genuinely new.
6. **Rank tells you the true dimensionality**—how much information is really there vs. how much was lost.
7. **Inverses undo transformations**—but only when no information was lost (full rank, non-zero determinant).
8. **Every neural network is just matrix multiplications** (plus some nonlinear functions). That's the whole secret!

Connection to LLMs

The Punchline: How ChatGPT Actually Works
When ChatGPT reads "The cat sat on the mat," here's what happens:

1. **Embedding:** Each word becomes a vector (lookup in an embedding table—just a big matrix!)
2. **Attention:** Dot products compute which words relate to which ("cat" attends to "sat")
3. **Transformation:** Vectors get multiplied by weight matrices (transformed through layers)
4. **More attention, more transformations:** Repeat 96 times (in GPT-4)
5. **Output:** A probability distribution over next words (more vectors!)

Every single step is linear algebra. Billions of dot products. Trillions of matrix multiplications. That's it. That's the whole secret.

You now understand the fundamental language of artificial intelligence.

Intuition

What's Next?
In Chapter 2, we'll go deeper into the structure of matrices:

- **Eigenvectors and eigenvalues:** The "natural directions" of a transformation
- **Eigendecomposition:** Breaking matrices into their fundamental components
- **Singular Value Decomposition (SVD):** The most important decomposition in data science

These tools reveal the hidden structure inside matrices—and they're the foundation for understanding how neural networks actually learn what they learn.

Ready for Chapter 2? Let's unlock the hidden structure of transformations!

"I understood matrices before, but now I ***see*** *them."* — Future you, after Chapter 2

Chapter 2

Advanced Linear Algebra: The Secret Superpowers of Matrices

"Eigen" is German for "own" or "characteristic." So eigenvectors are a matrix's "own" special vectors. Germans really know how to name things.

"The eigenvectors are the skeleton of the matrix—everything else is just flesh." — A professor who really loved linear algebra

In Chapter 1, you learned that matrices are transformations—they stretch, rotate, shear, and project vectors. But we treated matrices as mysterious black boxes. Put a vector in, get a vector out, hope for the best.

Now it's time to crack open the black box.

This chapter is about discovering the **hidden structure** inside every matrix. We'll find secret directions, hidden scaling factors, and learn to decompose any matrix into its fundamental building blocks. By the end, you'll be able to look at a matrix and understand *what it really does*—not just compute with it, but truly see it.

Connection to LLMs

Why This Chapter Will Change How You See AI
The concepts in this chapter aren't just abstract math—they're the foundation of:

- **Principal Component Analysis (PCA):** How we reduce 1000-dimensional data to something visualizable
- **Google's PageRank:** The algorithm that made Google Google (it's an eigenvector!)
- **LoRA fine-tuning:** How we adapt giant LLMs with minimal

resources (SVD in disguise)

- **Understanding training dynamics:** Why some neural networks train smoothly and others explode
- **Image/video compression:** Why your Netflix stream doesn't buffer (much)
- **Recommendation systems:** How Spotify knows you'll love that obscure indie band

If Chapter 1 taught you the alphabet, this chapter teaches you to read poetry.

2.1 The Mystery of the Special Directions

2.1.1 A Puzzle to Start

Imagine you have a magical Instagram filter. When you apply it to a picture, most things get warped—faces stretch, buildings lean, colors shift in weird ways. Your carefully composed selfie becomes abstract art (and not in a good way).

But here's the weird part: *some* parts of the image just get brighter or darker without changing shape at all. They refuse to warp. They're stubborn like that.

Those unchanged directions? They're special. They're the filter's "natural axes." And in the world of matrices, we call them **eigenvectors**. (Pronounced "EYE-gen-vectors," not "egg-en-vectors." You're welcome.)

This chapter is about finding those special directions and understanding why they matter so much for AI. Spoiler alert: They're the key to understanding how neural networks learn, how data compresses, and how Netflix knows you'll love that obscure documentary about competitive dog grooming.

Intuition

The Treasure Map Analogy
Imagine you're given a treasure map, but it's in a foreign language and uses a weird coordinate system. You could try to follow it directly (hard), or you could:

1. Translate it to English (change of basis)
2. Convert to North/South/East/West (natural coordinates)

3. Walk straight to the treasure (simple path)

That's what eigenvectors do for matrices! They find the "natural coordinate system" where everything becomes simple. Instead of a complicated transformation with spinning and stretching and shearing, you get pure scaling along nice perpendicular axes.
This is the central theme of this chapter: Finding the right perspective makes hard problems easy.

2.2 Eigenvectors: The Vectors That Refuse to Turn

2.2.1 The Big Idea (No Math Yet, Just Vibes)

Most of the time, when you multiply a matrix by a vector, the vector spins around, changes direction, gets warped into something unrecognizable. It's chaos. It's a vector having a bad day.

But sometimes—*sometimes*—you find a vector that's so perfectly aligned with what the matrix "wants to do" that it just gets stretched or shrunk, without rotating at all. It's like the vector found the matrix's secret shortcut.

Real-world analogies (because math needs friends):

The Shopping Cart: You're pushing a shopping cart with one wonky wheel. Most directions require constant course correction—you push forward, but the cart veers left. But there's ONE magical direction where the cart rolls perfectly straight. That's an eigendirection!

The Rubber Band Grid: Stretch a grid made of rubber bands. Most points move in complicated diagonal ways. But the vertical and horizontal directions? They just stretch along themselves, nice and simple. Those are eigendirections!

The Lazy River: Imagine a lazy river at a water park. Most directions fight the current. But if you float in exactly the right direction, the river just carries you faster (or slower) without pushing you sideways. Eigenvector found!

The Guitar String: Pluck a guitar string and it vibrates in a complicated mess. But there are certain "pure" vibration patterns (harmonics) where every point on the string just moves up and down without the pattern shifting sideways. These standing wave patterns are eigenvectors of the wave equation! Music is secretly linear algebra.

The Population Model: A country has different age groups. Each year, people age, some die, some give birth. This is a matrix transformation! But there's often a "stable age distribution" where the proportions stay the same even as total population grows. That distribution is an eigenvector of the population matrix.

2.2.2 The Mathematical Definition (Now with Math!)

Definition 2.1: Eigenvector and Eigenvalue

For a square matrix $\mathbf{A}$, a non-zero vector $\mathbf{v}$ is called an **eigenvector** if:

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

where λ (lambda) is a scalar called the **eigenvalue**.
In plain English: "Multiplying $\mathbf{v}$ by the scary matrix $\mathbf{A}$ is the same as just multiplying $\mathbf{v}$ by a single number λ." The matrix becomes trivially simple in that direction!

What this means:

- The vector $\mathbf{v}$ doesn't change direction (doesn't rotate)
- It only gets scaled by λ (stretched/shrunk/flipped)
- If $\lambda > 1$: vector gets longer (stretched)
- If $0 < \lambda < 1$: vector gets shorter (compressed)
- If $\lambda < 0$: vector flips direction (and scales)
- If $\lambda = 0$: vector gets annihilated! Sent to the zero vector.
- If $\lambda = 1$: vector stays exactly the same! (fixed point)

Key insight: The eigenvalue λ tells you the "strength" of that direction. Large $|\lambda|$ = important direction. Small $|\lambda|$ = less important. Zero λ = that direction gets destroyed.

Example

Let's see this in action. Consider the matrix:

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

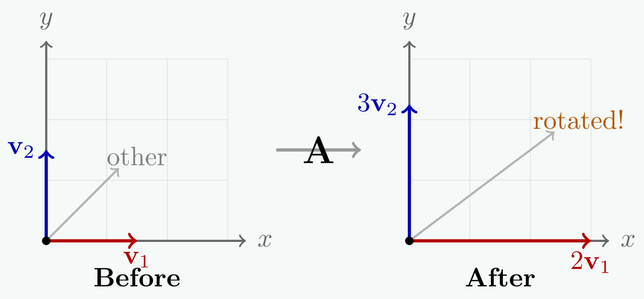
This matrix scales the x-direction by 2 and the y-direction by 3.
Try the vector $\mathbf{v} = [1, 0]$ (pointing along the x-axis):

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

It got scaled by 2! So $\mathbf{v} = [1, 0]$ is an eigenvector with eigenvalue $\lambda = 2$.
Now try $\mathbf{u} = [0, 1]$ (pointing along the y-axis):

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

This one got scaled by 3! So $\mathbf{u} = [0, 1]$ is an eigenvector with eigenvalue $\lambda = 3$.
But if you try a diagonal vector like $[1, 1]$, it gets warped into $[2, 3]$—NOT just a scaled version of $[1, 1]$. So $[1, 1]$ is NOT an eigenvector of this matrix.
THE GEOMETRIC VIEW (This Makes Everything Clear!):



Eigenvectors: same direction, just stretched *Other vectors: direction changes!*

The Key Insight: Eigenvectors are the "stable directions" of a matrix. They show you the axes along which the transformation is simplest (just scaling, no rotation).

Intuition

Why should you care? (Besides impressing people at parties)
Eigenvectors are the “natural coordinates” of a transformation. They’re the directions where the matrix acts in the simplest possible way—just scaling, no spinning.
Think of it like this: Every matrix has a “personality.” Most of the time, that personality is complicated. But eigenvectors reveal the matrix’s true nature. They’re like a personality test for matrices.
If you’re analyzing data and want to understand “what patterns does this data have?”, you look at eigenvectors. They point toward the

most important directions in your data. This is literally how PCA (Principal Component Analysis) works, which is used in everything from face recognition to Netflix recommendations.

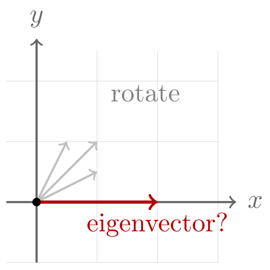
2.2.3 A More Interesting Example

Let's try a matrix that's not so obvious:

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

This one's not diagonal, so the eigenvectors aren't staring us in the face. We'll have to do some detective work!

The Mystery: Which directions stay on the same line after this transformation?



Claim: $\mathbf{v} = [1, 0]$ is an eigenvector.
Check:

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

Yes! Eigenvalue is $\lambda_1 = 3$.
Claim: $\mathbf{w} = [1, -1]$ is an eigenvector.
Check:

$$\mathbf{Bw} = \begin{bmatrix} 3 & 1 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} 1 \\ -1 \end{bmatrix} = \begin{bmatrix} 3 - 1 \\ 0 - 2 \end{bmatrix} = \begin{bmatrix} 2 \\ -2 \end{bmatrix} = 2 \begin{bmatrix} 1 \\ -1 \end{bmatrix} = 2\mathbf{w}$$

Yes! Eigenvalue is $\lambda_2 = 2$.

So this matrix has two special directions: $[1, 0]$ (gets scaled by 3) and $[1, -1]$ (gets scaled by 2).

2.3 How to Actually Find Eigenvectors (The Recipe)

2.3.1 The Characteristic Equation

Okay, we've been guessing eigenvectors like we're playing mathematical Wordle. But how do you *actually* find them systematically? Time for some real math!

Here's the strategy. We want to solve:

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

Rearrange:

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

Factor out $\mathbf{v}$ (careful—we need the identity matrix):

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

For this to have a non-zero solution (we don't want $\mathbf{v} = \mathbf{0}$, that's boring), the matrix $(\mathbf{A}-\lambda\mathbf{I})$ must be "broken"—it must not have an inverse. Mathematically, its determinant must be zero.

Definition 2.2: The Characteristic Equation

To find eigenvalues of $\mathbf{A}$, solve:

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

This equation (often a polynomial in λ) is called the **characteristic equation**. It's the matrix's "DNA test"—it reveals the eigenvalues hiding inside.
Once you have the eigenvalues λ, plug each one back into $(\mathbf{A}-\lambda\mathbf{I})\mathbf{v} = \mathbf{0}$ to find the corresponding eigenvector $\mathbf{v}$.

Don't panic! This looks scary, but for 2×2 matrices it's just solving a quadratic equation. You've been doing that since high school. For larger matrices... well, that's why we have computers.

Intuition

Why Does Setting the Determinant to Zero Work?
Remember from Chapter 1: a matrix has no inverse when its determinant is zero (it squashes space).
The equation $(\mathbf{A} - \lambda\mathbf{I})\mathbf{v} = \mathbf{0}$ is asking: "When does this modified matrix send a non-zero vector to zero?"

That can only happen if $(\mathbf{A} - \lambda\mathbf{I})$ squashes some direction—i.e., has no inverse—i.e., has determinant zero!
So we're finding the special values of λ that "break" the matrix $\mathbf{A} - \lambda\mathbf{I}$. Those breaking points are exactly the eigenvalues.
Analogy: It's like finding the resonant frequencies of a bridge. At most frequencies, the bridge just vibrates a little. But at certain special frequencies (eigenvalues), it resonates dramatically—small input, huge output. Those are the bridge's "natural frequencies," and they're eigenvalues of the bridge's stiffness matrix!

2.3.2 Example: Finding Eigenvalues Step by Step

Let's find the eigenvalues of:

$$\mathbf{A} = \begin{bmatrix} 4 & 1 \\ 2 & 3 \end{bmatrix}$$

Step 1: Form $\mathbf{A} - \lambda\mathbf{I}$

$$\mathbf{A} - \lambda\mathbf{I} = \begin{bmatrix} 4 & 1 \\ 2 & 3 \end{bmatrix} - \lambda \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 4-\lambda & 1 \\ 2 & 3-\lambda \end{bmatrix}$$

Step 2: Compute the determinant
For a 2×2 matrix $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$, the determinant is $ad - bc$.

$$\begin{aligned} \det(\mathbf{A} - \lambda\mathbf{I}) &= (4-\lambda)(3-\lambda) - (1)(2) \\ &= 12 - 4\lambda - 3\lambda + \lambda^2 - 2 \\ &= \lambda^2 - 7\lambda + 10 \end{aligned}$$

Step 3: Set it equal to zero and solve

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

Factor (or use quadratic formula):

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

So $\lambda_1 = 5$ and $\lambda_2 = 2$. Those are the eigenvalues!
Step 4: Find the eigenvectors
For $\lambda_1 = 5$, solve $(\mathbf{A} - 5\mathbf{I})\mathbf{v} = \mathbf{0}$:

$$\begin{bmatrix} -1 & 1 \\ 2 & -2 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

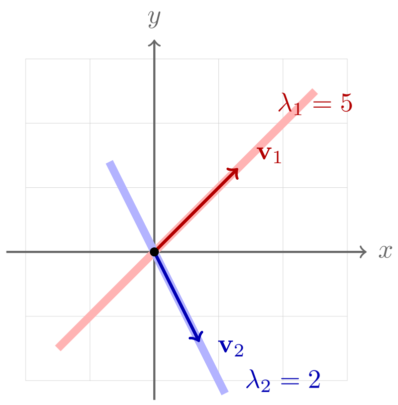
First row: $-v_1 + v_2 = 0 \Rightarrow v_2 = v_1$
So any vector of the form $\begin{bmatrix} v_1 \\ v_1 \end{bmatrix} = v_1 \begin{bmatrix} 1 \\ 1 \end{bmatrix}$ works.
Eigenvector for $\lambda_1 = 5$: $\mathbf{v}_1 = [1, 1]$ (or any scalar multiple)
For $\lambda_2 = 2$, solve $(\mathbf{A} - 2\mathbf{I})\mathbf{v} = \mathbf{0}$:

$$\begin{bmatrix} 2 & 1 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

First row: $2v_1 + v_2 = 0 \Rightarrow v_2 = -2v_1$
Eigenvector for $\lambda_2 = 2$: $\mathbf{v}_2 = [1, -2]$ (or any scalar multiple)
Summary:

- Eigenvalue $\lambda_1 = 5$ with eigenvector $[1, 1]$
- Eigenvalue $\lambda_2 = 2$ with eigenvector $[1, -2]$

Let's Visualize What We Found:



Shaded lines = eigenspaces (all multiples of eigenvectors)

The dashed lines show the "eigenspaces"—the matrix only stretches along these directions!

Intuition

What did we just do?
We found the two "natural directions" of this matrix. Multiply by $[1, 1]$? Stretches by 5. Multiply by $[1, -2]$? Stretches by 2. Everything else is a cocktail of these two behaviors.
Real-world analogy: Imagine pushing on a wobbly table. Most directions cause it to tip, wobble, and generally misbehave. But there are exactly two special directions where it just slides smoothly

without drama. Those are the eigendirections! Every matrix has its own "drama-free zones."

Intuition

The "Eigenvalue Decoder Ring"
Here's a cheat sheet for interpreting eigenvalues in applications:

Eigenvalue	**What it means**
$\lambda > 1$	Growth/expansion in that direction
$\lambda = 1$	Equilibrium/steady state
$0 < \lambda < 1$	Decay/contraction
$\lambda = 0$	That direction is destroyed
$\lambda < 0$	Oscillation/flipping
$\|\lambda_1\| \gg \|\lambda_2\|$	One direction dominates
All $\|\lambda\| < 1$	System converges to zero
Any $\|\lambda\| > 1$	System explodes!

In neural networks: If the eigenvalues of weight matrices have magnitude > 1, gradients explode during training. If they're all < 1, gradients vanish. The sweet spot is near 1, which is why careful initialization matters so much!

2.4 Why Eigenvectors Are Magical

2.4.1 Beautiful Properties

Theorem 2.1: Cool Facts About Eigenvalues

For a matrix $\mathbf{A}$:

1. Sum of eigenvalues = trace (sum of diagonal elements)

$$\lambda_1 + \lambda_2 + \cdots + \lambda_n = a_{11} + a_{22} + \cdots + a_{nn}$$

2. Product of eigenvalues = determinant

$$\lambda_1 \times \lambda_2 \times \cdots \times \lambda_n = \det(\mathbf{A})$$

3. A is invertible $\Leftrightarrow$ all eigenvalues are non-zero
If any $\lambda = 0$, the matrix squashes some direction to zero, so you can't undo it.

4. Eigenvectors from different eigenvalues are linearly independent
They point in genuinely different directions—no redundancy!

Example

For $\mathbf{A} = \begin{bmatrix} 4 & 1 \\ 2 & 3 \end{bmatrix}$, we found $\lambda_1 = 5, \lambda_2 = 2$.
Check:

- Trace: $4 + 3 = 7 = 5 + 2$ (check!)
- Determinant: $(4)(3) - (1)(2) = 10 = 5 \times 2$ (check!)

These properties give you a quick sanity check when finding eigenvalues!

2.4.2 Special Case: Symmetric Matrices Are the Best

Symmetric matrices are the golden retrievers of the matrix world—friendly, predictable, and everybody loves them.

Theorem 2.2: Spectral Theorem (Simplified)

If $\mathbf{A}$ is symmetric (i.e., $\mathbf{A} = \mathbf{A}^\top$), then:

- All eigenvalues are **real numbers** (no imaginary nonsense!)
- Eigenvectors from different eigenvalues are **perpendicular** (orthogonal)
- You can always find n perpendicular eigenvectors for an $n \times n$ symmetric matrix

Why this is AMAZING: Symmetric matrices are super well-behaved. Their eigenvectors form a nice perpendicular coordinate system, which makes everything cleaner. It's like they went to finishing school.

Intuition

Why Are Symmetric Matrices So Special?
Here's the deep reason: symmetric matrices represent "undirected" relationships.

- If person A's influence on person B equals B's influence on A, the influence matrix is symmetric
- If the force between particle A and B equals the force between B and A (Newton's third law!), the interaction matrix is symmetric
- If the correlation between variables X and Y equals the correlation between Y and X, the correlation matrix is symmetric

Nature loves symmetry, and symmetric matrices inherit that beauty. They're honest—they don't have hidden asymmetries that could cause weird complex eigenvalues or non-perpendicular eigenvectors.
The Guarantee: For any $n \times n$ symmetric matrix, you can ALWAYS find n perpendicular eigenvectors that form a complete basis. No exceptions. No edge cases. Mathematics at its most elegant.

Connection to LLMs

Where symmetric matrices show up in AI:

- **Covariance matrices** (measure how data dimensions correlate) are always symmetric
- **Graph adjacency matrices** (social networks, molecule structures) are often symmetric
- **Hessian matrices** (second derivatives in optimization) are symmetric
- **Kernel matrices** (in SVMs and Gaussian processes) are symmetric positive semi-definite
- **Gram matrices** ($\mathbf{X}^\top\mathbf{X}$, used everywhere) are always symmetric

So eigenvectors of symmetric matrices are everywhere in machine learning!

Example

Google's PageRank: The Billion-Dollar Eigenvector
Here's a real story about eigenvectors making history.
In 1998, Larry Page and Sergey Brin had a problem: how do you rank web pages by importance? Their insight: a page is important if important pages link to it. But that's circular!
The solution: model the web as a matrix. Entry (i, j) represents the probability of clicking from page j to page i. The "importance" vector $\mathbf{r}$ should satisfy:

$$\mathbf{Ar} = \mathbf{r}$$

Wait... that's an eigenvector equation with $\lambda = 1$!
The importance ranking of every webpage is literally the eigenvector of the web's link matrix with eigenvalue 1. Google's original algorithm was just finding an eigenvector.
The punchline: The eigenvector that launched a trillion-dollar company. Linear algebra pays.

2.5 Eigendecomposition: Breaking Matrices Apart

2.5.1 The Big Idea

If a matrix has n linearly independent eigenvectors, we can "decompose" it into simpler pieces.

Definition 2.3: Eigendecomposition

If $\mathbf{A}$ has n linearly independent eigenvectors $\mathbf{v}_1, \ldots, \mathbf{v}_n$ with eigenvalues $\lambda_1, \ldots, \lambda_n$, then:

$$\mathbf{A} = \mathbf{V}\boldsymbol{\Lambda}\mathbf{V}^{-1}$$

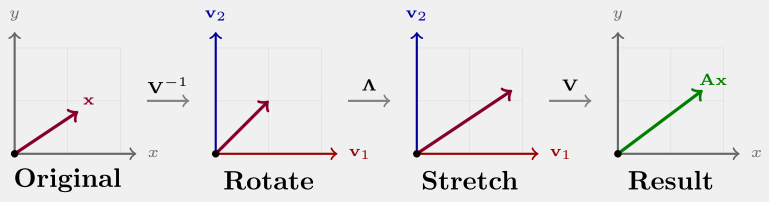
where:

- $\mathbf{V} = [\mathbf{v}_1 \,|\, \mathbf{v}_2 \,|\, \cdots \,|\, \mathbf{v}_n]$ (columns are eigenvectors)
- $\boldsymbol{\Lambda} = \begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_n \end{bmatrix}$ (diagonal matrix of eigenvalues)

In plain English: "Any nice matrix is just a rotation ($\mathbf{V}^{-1}$), a stretch along axes ($\mathbf{\Lambda}$), and a rotation back ($\mathbf{V}$)."

Intuition

Think of $\mathbf{A}$ as a complicated transformation. The eigendecomposition says:
Step 1: Rotate to the eigenvector coordinate system ($\mathbf{V}^{-1}$)
Step 2: In this new system, just scale each axis independently ($\mathbf{\Lambda}$)—super simple!
Step 3: Rotate back to the original coordinates ($\mathbf{V}$)
It's like saying "this complex motion is just a simple stretch, if you look at it from the right angle."
VISUALIZING EIGENDECOMPOSITION: The Three-Step Dance
Think of it as a change of clothes: go to your "eigen-closet," do the simple scaling, then change back.



The Beautiful Insight: In the eigenvector coordinate system, the transformation is just diagonal scaling! All the complexity comes from rotating in and out of that special coordinate system.
Analogy: Imagine trying to describe a spiral staircase. In normal coordinates, it's complicated. But if you use cylindrical coordinates (angle, radius, height), it's just "go up while staying at constant radius"—much simpler! Eigendecomposition finds those simpler coordinates.

Example

Eigendecomposition in Action
Let's actually decompose a matrix. Take:

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

Step 1: Find eigenvalues
$\det(\mathbf{A} - \lambda\mathbf{I}) = (2-\lambda)^2 - 1 = \lambda^2 - 4\lambda + 3 = (\lambda - 3)(\lambda - 1) = 0$

So $\lambda_1 = 3$ and $\lambda_2 = 1$.
Step 2: Find eigenvectors
For $\lambda_1 = 3$: Solve $(\mathbf{A} - 3\mathbf{I})\mathbf{v} = 0$

$$\begin{bmatrix} -1 & 1 \\ 1 & -1 \end{bmatrix} \mathbf{v} = 0 \Rightarrow \mathbf{v}_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

For $\lambda_2 = 1$: Solve $(\mathbf{A} - \mathbf{I})\mathbf{v} = 0$

$$\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \mathbf{v} = 0 \Rightarrow \mathbf{v}_2 = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$$

Step 3: Assemble the decomposition
Normalize (make length 1): $\hat{\mathbf{v}}_1 = \frac{1}{\sqrt{2}}[1, 1]^\top$, $\hat{\mathbf{v}}_2 = \frac{1}{\sqrt{2}}[1, -1]^\top$
Since $\mathbf{A}$ is symmetric, $\mathbf{V}$ is orthogonal, so $\mathbf{V}^{-1} = \mathbf{V}^\top$:

$$\mathbf{A} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^\top = \frac{1}{2} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 3 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

Interpretation: This matrix stretches by 3 along the diagonal $[1, 1]$ and by 1 along the anti-diagonal $[1, -1]$. That's it! The symmetric matrix was secretly just doing axis-aligned stretching in a rotated coordinate system.

2.5.2 Why This Is Insanely Useful

1. Computing matrix powers:

If you need to compute $\mathbf{A}^{100}$ (multiply $\mathbf{A}$ by itself 100 times), it's hard. But with eigendecomposition:

$$\mathbf{A}^{100} = (\mathbf{V}\mathbf{\Lambda}\mathbf{V}^{-1})^{100} = \mathbf{V}\mathbf{\Lambda}^{100}\mathbf{V}^{-1}$$

And $\mathbf{\Lambda}^{100}$ is easy—just raise each eigenvalue to the 100th power:

$$\mathbf{\Lambda}^{100} = \begin{bmatrix} \lambda_1^{100} & 0 & \cdots \\ 0 & \lambda_2^{100} & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix}$$

2. Understanding long-term behavior:

If you repeatedly apply $\mathbf{A}$ to a vector (like in iterative algorithms), the behavior is dominated by the largest eigenvalue. This tells you if things converge, diverge, or oscillate.

3. Principal Component Analysis (PCA):

PCA finds the directions of maximum variance in data. How? By finding the eigenvectors of the covariance matrix! The eigenvector with the largest eigenvalue points in the direction where data varies the most.

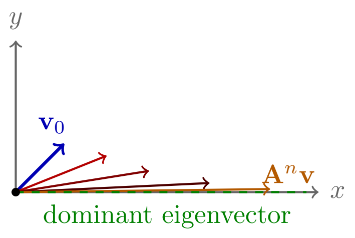
Intuition

PCA: Finding the "Soul" of Your Data
Imagine you have data about people: height, weight, shoe size, arm length, etc. That's maybe 10 dimensions. Hard to visualize!
PCA asks: "What are the REAL underlying factors?" Maybe all those measurements are really driven by just 2-3 underlying factors (like overall body size, body proportions, etc.).
The Algorithm:

1. Compute the covariance matrix of your data
2. Find its eigenvectors and eigenvalues
3. The eigenvector with the LARGEST eigenvalue = the direction of most variance = the most important "factor"
4. The second largest = second most important, perpendicular to the first
5. Keep the top k eigenvectors, discard the rest

You've now reduced 10 dimensions to k dimensions, keeping the most important patterns!
Real example: In face recognition, faces are 10,000 pixels. PCA finds that most face variation can be captured by 100 "eigenfaces" (eigenvectors of the face covariance matrix). 100× compression while keeping what matters!

Visual Example of Matrix Powers:



Keep multiplying → converge to dominant eigenvector direction!

Connection to LLMs

How neural networks use eigendecomposition:

- **Analyzing training dynamics:** The eigenvalues of the Hessian (matrix of second derivatives) tell you if optimization will be fast or slow. If eigenvalues vary wildly (some huge, some tiny), training is difficult—you need different step sizes for different directions!
- **Compression:** Keep only the eigenvectors with large eigenvalues, discard the rest (dimensionality reduction)
- **Initialization:** Understanding eigenvalues helps design better ways to initialize network weights. Xavier and He initialization are designed to keep eigenvalues near 1.
- **Spectral normalization:** Divide weight matrices by their largest eigenvalue to stabilize GAN training

Intuition

The "Condition Number" Problem
The ratio of the largest to smallest eigenvalue is called the **condition number**:

$$\kappa(\mathbf{A}) = \frac{|\lambda_{\text{max}}|}{|\lambda_{\text{min}}|}$$

- $\kappa \approx 1$: Well-conditioned. All directions are treated similarly. Easy to work with!
- $\kappa \gg 1$: Ill-conditioned. Some directions are amplified WAY more than others. Numerically unstable!

In optimization: If the Hessian has $\kappa = 1000$, gradient descent struggles. It takes tiny steps in some directions and huge steps in others. That's why we use adaptive optimizers like Adam—they effectively rescale to make the condition number closer to 1.
Analogy: Optimizing with high condition number is like walking through a steep, narrow valley. You keep bouncing off the walls instead of going down. Better to transform the valley into a nice bowl first!

2.6 Singular Value Decomposition (SVD): The Ultimate Weapon

2.6.1 What If the Matrix Isn't Square?

Eigendecomposition is great, but it has a fatal flaw: it only works for square matrices. What about $m \times n$ matrices (different numbers of rows and columns)? What about matrices that don't play nice with eigenvectors?

Enter SVD: the most powerful matrix decomposition of all time. If eigendecomposition is a Swiss Army knife, SVD is the entire hardware store. It works on *any* matrix. Any size. Any shape. No exceptions.

Intuition

SVD vs Eigendecomposition: The Key Difference
Eigendecomposition asks: "What directions does this matrix leave unchanged (except for scaling)?"

- Only works for square matrices
- Might not find enough eigenvectors
- Input and output are in the same space

SVD asks: "What are the best input directions that map to the best output directions?"

- Works for ANY matrix (any shape!)
- Always finds a complete set of directions
- Input and output can be different spaces

Analogy: Eigendecomposition is like finding the natural vibration modes of a single drum. SVD is like finding the best way to transmit sound from a speaker (input space) to a microphone (output space)—even if they're in different rooms!

Quick Setup: Imagine you have a rectangular matrix (like a transformation that changes dimensions—3D to 2D, or 100D to 50D). How do you understand what it does?

Definition 2.4: Singular Value Decomposition

Any matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ can be decomposed as:

$$\mathbf{A} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^\top$$

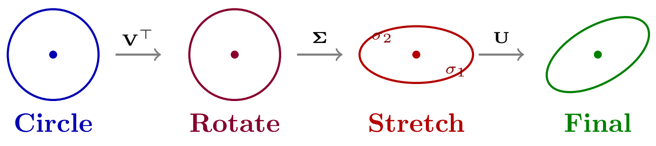
where:

- $\mathbf{U} \in \mathbb{R}^{m \times m}$ is orthogonal (columns are perpendicular unit vectors)
- $\boldsymbol{\Sigma} \in \mathbb{R}^{m \times n}$ is diagonal (only has entries on the main diagonal)
- $\mathbf{V} \in \mathbb{R}^{n \times n}$ is orthogonal
- The diagonal entries $\sigma_1 \geq \sigma_2 \geq \cdots \geq 0$ are called **singular values**

In plain English: "Every matrix is just a rotation, a stretch along perpendicular axes, and another rotation."

THE SVD TRANSFORMATION: Circle to Ellipse

The beautiful thing about SVD: it shows that *every* matrix turns a circle into an ellipse. That's it! Rotate, stretch into ellipse, rotate again.



Every matrix turns a circle into an ellipse: Rotate → Stretch → Rotate

What Just Happened:

1. Started with unit circle (all possible inputs of length 1)
2. $\mathbf{V}^\top$: Rotated to the "right" input coordinate system
3. $\boldsymbol{\Sigma}$: Stretched along perpendicular axes (circle → ellipse)
4. $\mathbf{U}$: Rotated to the "right" output coordinate system
5. Result: An ellipse! The matrix turned a circle into an ellipse!

The Singular Values: The entries in $\boldsymbol{\Sigma}$ (the σ_i values) are how much stretching happens along each axis. They're called "singular values."

Key facts about singular values:

- Always real and non-negative (unlike eigenvalues, which can be complex or negative)
- Always ordered: $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r > 0$
- The number of non-zero singular values = the rank of the matrix
- σ_1 is the maximum "stretching factor" of the matrix
- σ_r (smallest non-zero) is the minimum stretching factor
- Their ratio σ_1/σ_r is the condition number!

Intuition

Think of SVD like this:
You have a transformation $\mathbf{A}$ that might stretch, rotate, skew, project—do all kinds of crazy stuff.
SVD says: "Actually, if you pick the RIGHT input coordinate system ($\mathbf{V}^\top$) and the RIGHT output coordinate system ($\mathbf{U}$), then $\mathbf{A}$ is just stretching/shrinking along perpendicular directions ($\mathbf{\Sigma}$). No rotation, no weird stuff."
It's finding the "natural coordinates" for both input and output spaces.
The Deep Connection to Eigendecomposition:
Here's a beautiful fact: the singular values of $\mathbf{A}$ are the square roots of the eigenvalues of $\mathbf{A}^\top\mathbf{A}$ (or $\mathbf{A}\mathbf{A}^\top$).
Why? Because $\mathbf{A}^\top\mathbf{A}$ is symmetric (always!), so it has nice real eigenvalues. And:

$$\mathbf{A}^\top\mathbf{A} = (\mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top)^\top(\mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top) = \mathbf{V}\mathbf{\Sigma}^\top\mathbf{U}^\top\mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top = \mathbf{V}\mathbf{\Sigma}^2\mathbf{V}^\top$$

So the right singular vectors ($\mathbf{V}$) are eigenvectors of $\mathbf{A}^\top\mathbf{A}$, and the singular values squared are the eigenvalues!
This is often how SVD is computed in practice: find the eigendecomposition of $\mathbf{A}^\top\mathbf{A}$, then take square roots.

2.6.2 Why SVD Is the Swiss Army Knife of Linear Algebra

SVD can do EVERYTHING:

1. **Solve linear systems:** Even overdetermined or underdetermined ones
2. **Find the rank:** Count how many singular values are non-zero
3. **Compress data:** Keep only the largest singular values

4. **Denoise data:** Small singular values = noise; drop them
5. **Pseudoinverse:** Invert non-square or singular matrices
6. **Recommendation systems:** Collaborative filtering (Netflix prize!)
7. **Latent semantic analysis:** Discover hidden topics in text
8. **Face recognition:** Eigenfaces are really singular vectors
9. **Calculate matrix norms:** $\|\mathbf{A}\|_2 = \sigma_1$ (largest singular value)
10. **Low-rank approximation:** Best possible compression (Eckart-Young theorem)

Theorem 2.3: Eckart-Young Theorem (The Best Compression Guarantee)

The best rank-k approximation to a matrix $\mathbf{A}$ (in terms of minimizing the Frobenius norm of the error) is given by truncated SVD:

$$\mathbf{A}_k = \sum_{i=1}^{k} \sigma_i \mathbf{u}_i \mathbf{v}_i^\top$$

No other rank-k matrix is closer to $\mathbf{A}$. SVD gives you the optimal compression, guaranteed!

2.6.3 Example: Image Compression with SVD

Here's something mind-blowingly practical: an image is just a matrix of pixel brightnesses. Every photo you've ever taken is secretly a matrix. And SVD can compress it like a boss!

The Idea:

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top = \sigma_1 \mathbf{u}_1 \mathbf{v}_1^\top + \sigma_2 \mathbf{u}_2 \mathbf{v}_2^\top + \cdots + \sigma_r \mathbf{u}_r \mathbf{v}_r^\top$$

This is a sum of rank-1 matrices, each scaled by σ_i.

Compression trick: Keep only the largest k terms:

$$\mathbf{A}_{\text{compressed}} \approx \sigma_1 \mathbf{u}_1 \mathbf{v}_1^\top + \cdots + \sigma_k \mathbf{u}_k \mathbf{v}_k^\top$$

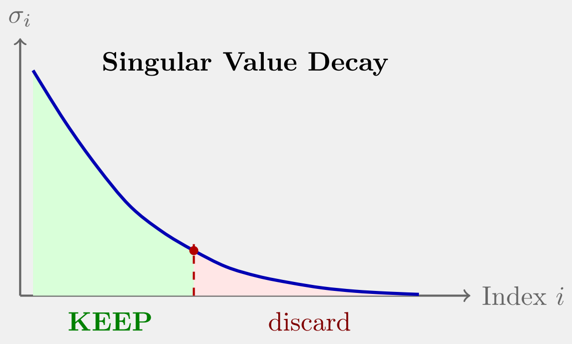
If k is much smaller than the full rank, you've compressed the image! You're storing way fewer numbers, and the image still looks good because the large singular values capture most of the information.

Real numbers: A 1000×1000 image has 1,000,000 pixels. With rank-50 SVD, you store only about 100,000 numbers—10× compression! And it usually looks almost identical. This is basically how JPEG works (with some extra tricks).

Intuition

Why Does This Work? The "Energy" Interpretation
The singular values tell you how much "energy" or "importance" is in each direction. For most real-world data:

- The first few singular values are HUGE (capture main patterns)
- The rest decay rapidly (capture noise and fine details)



When you keep only the top k singular values, you're keeping the "important" patterns and discarding the noise. The retained energy is:

$$\text{Energy retained} = \frac{\sigma_1^2 + \cdots + \sigma_k^2}{\sigma_1^2 + \cdots + \sigma_n^2}$$

Often, 95%+ of the energy is in the first 10% of singular values!

Connection to LLMs

SVD in LLMs:

- **LoRA (Low-Rank Adaptation):** Instead of fine-tuning a full weight matrix, LoRA learns a low-rank update $\mathbf{A} + \mathbf{U}\mathbf{V}^\top$ where $\mathbf{U}$ and $\mathbf{V}$ are skinny matrices. This is basically SVD thinking!
- **Latent Semantic Analysis:** Early NLP technique that used SVD on word-document matrices to discover topics
- **Analyzing attention patterns:** SVD can decompose attention matrices to understand what the model is focusing on

Modern compression techniques for LLMs (quantization, pruning, distillation) are often inspired by SVD ideas: "keep the important directions, discard the rest."

Example

LoRA: The SVD-Inspired Revolution in Fine-Tuning
A GPT-3 sized model has 175 billion parameters. Fine-tuning ALL of them for your specific task is:

- Computationally expensive (need huge GPUs)
- Memory-intensive (need to store gradients for 175B parameters)
- Risky (might forget what it learned before)

LoRA's insight: The CHANGE in weights during fine-tuning is probably low-rank! You don't need to change everything—just a few important directions.
Instead of learning a new weight matrix $\mathbf{W}'$, LoRA learns:

$$\mathbf{W}' = \mathbf{W} + \mathbf{BA}$$

where $\mathbf{B}$ is $d \times r$ and $\mathbf{A}$ is $r \times d$, with $r \ll d$ (e.g., $r = 8$ when $d = 4096$).
The math: $\mathbf{BA}$ is a rank-r matrix. It's like we're only updating in r directions, not all d!
The savings: Instead of d^2 parameters, we only train $2dr$ parameters. For $d = 4096$ and $r = 8$:

- Full fine-tuning: 16,777,216 parameters per layer
- LoRA: 65,536 parameters per layer
- That's 256$\times$ fewer parameters!

This is why you can fine-tune LLMs on a laptop now. Thank SVD thinking!

2.7 Putting It All Together

You made it! You now know more about matrices than 99% of the population. Let's summarize your new superpowers.

2.7.1 The Hierarchy of Matrix Decompositions

- **Eigendecomposition:** Works for square matrices with enough independent eigenvectors

  - Best for: Understanding dynamics, computing powers, PCA
  - Special case: Symmetric matrices have perpendicular eigenvectors

- **SVD:** Works for ANY matrix (square or rectangular)
  - Best for: Compression, rank finding, solving systems, recommendation
  - Always gives perpendicular bases for input and output

2.7.2 The Big Picture

All of these decompositions are about the same fundamental idea:

"Find the natural coordinate system where the transformation is simplest."

- Eigenvectors: The directions where $\mathbf{A}$ just scales
- SVD: The input/output bases where $\mathbf{A}$ is just diagonal scaling

This philosophy—"find the right coordinates to make things simple"—is central to all of machine learning.

Intuition

The Zen of Matrix Decomposition
Every matrix decomposition is answering a question:

Decomposition	**Question it answers**
Eigendecomposition	What are the natural directions?
SVD	What's the best low-rank approximation?
LU decomposition	How do I solve linear systems?
QR decomposition	How do I find orthogonal bases?
Cholesky	How do I factor symmetric PD matrices?

Each decomposition reveals a different aspect of the matrix's "personality." Learning when to use which is part of becoming a linear algebra master.

The Master Skill: Look at a problem and ask "What structure can I exploit?" Is the matrix symmetric? Use eigendecomposition. Need to compress? Use SVD. Solving many systems with the same matrix? Use LU. The right decomposition turns a hard problem into an easy one.

2.8 Practice Problems

Intuition

How to Approach These Problems
Eigenvector problems can feel abstract. Here's a mindset that helps:

1. **Geometric thinking first:** Before computing, ask "What does this matrix DO?" Does it stretch? Rotate? Project?
2. **Sanity checks:** After finding eigenvalues, verify that trace = sum and determinant = product
3. **Interpret your answer:** Don't just compute $\lambda = 3$—think "this direction gets stretched by 3x"

2.8.1 Problem 1: Basic Eigenvalues

Find the eigenvalues and eigenvectors of:

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

Hint: This is an upper triangular matrix. What's special about eigenvalues of triangular matrices?

2.8.2 Problem 2: Eigenvalue Properties

Given a matrix with eigenvalues $\lambda_1 = 3$, $\lambda_2 = -2$, $\lambda_3 = 5$:

1. What is the trace of the matrix?
2. What is the determinant?
3. Is the matrix invertible? Why or why not?

2.8.3 Problem 3: Understanding Eigenvectors (The Fun One)

If $\mathbf{v}$ is an eigenvector of $\mathbf{A}$ with eigenvalue $\lambda = 2$:

1. What is $\mathbf{A}^{10}\mathbf{v}$? (Express in terms of $\mathbf{v}$)
2. If $\lambda = 0.5$, what happens to $\mathbf{v}$ as you repeatedly apply $\mathbf{A}$? Where does it end up?

3. If $\lambda = -1$, what happens? (Hint: think about $\mathbf{A}^1\mathbf{v}$, $\mathbf{A}^2\mathbf{v}$, $\mathbf{A}^3\mathbf{v}$...)
4. If $\lambda = 1$, what's special about $\mathbf{v}$?
5. **Challenge:** If a matrix has eigenvalues $\lambda_1 = 1.01$ and $\lambda_2 = 0.99$, what happens to a random vector after applying the matrix 1000 times?

2.8.4 Problem 4: Connection to LLMs

Explain in your own words:

1. Why might we want to find the eigenvectors of a word embedding covariance matrix?
2. How does SVD help compress neural network weights?
3. What does it mean if an attention matrix has one very large eigenvalue and many small ones?
4. In LoRA, we approximate weight updates as $\Delta\mathbf{W} = \mathbf{BA}$ where $\mathbf{B}$ is $d \times r$ and $\mathbf{A}$ is $r \times d$. What is the rank of $\Delta\mathbf{W}$? Why does this save memory?

2.8.5 Problem 5: SVD Compression

A grayscale image is stored as a 100×100 matrix (10,000 pixels).

1. If you compute the full SVD, how many singular values will there be (at most)?
2. If you keep only the top 10 singular values, how many numbers do you need to store? (Count entries in truncated $\mathbf{U}$, $\mathbf{\Sigma}$, and $\mathbf{V}$)
3. What is the compression ratio?
4. If the singular values are $\sigma_1 = 100, \sigma_2 = 50, \sigma_3 = 25, \sigma_4 = 12, \sigma_5 = 6$, and the rest are below 1, what percentage of the "energy" ($\sum \sigma_i^2$) is captured by the top 5 components?

2.8.6 Problem 6: Geometric Intuition

Without computing, predict the eigenvectors and eigenvalues of these matrices:

(a) $\begin{bmatrix} 3 & 0 \\ 0 & 3 \end{bmatrix}$ (uniform scaling)

(b) $\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$ (reflection across x-axis)

(c) $\begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$ (90° rotation)

Hint for (c): Does a rotation have any direction that doesn't rotate? What does this imply about the eigenvalues?

2.9 Key Takeaways

1. **Eigenvectors are special directions** that only get scaled (not rotated) by a matrix
2. **Eigenvalues tell you how much** the scaling is (stretch, shrink, flip)
3. **To find them:** Solve $\det(\mathbf{A} - \lambda\mathbf{I}) = 0$ for eigenvalues, then find eigenvectors
4. **Symmetric matrices are magic:** Real eigenvalues, perpendicular eigenvectors
5. **Eigendecomposition:** $\mathbf{A} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^{-1}$ breaks a matrix into rotation + stretch + rotation
6. **SVD works on ANY matrix:** $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^{\top}$ with perpendicular bases
7. **These tools power:** PCA, image compression, LoRA, recommendation systems, and understanding network dynamics

Connection to LLMs

The deep learning connection:
When you train a neural network, you're adjusting weights in a high-dimensional space. Eigenvalues and eigenvectors help you understand:

- Which directions in weight space matter most
- How fast/slow training will converge
- Where to compress without losing performance
- What patterns the network has learned

Everything comes back to: *"Find the natural coordinate system where the problem is simpler."*

That's the heart of advanced linear algebra, and it's the heart of modern AI.

Chapter 2 Cheat Sheet

Eigenvectors $\mathbf{A}\mathbf{v} = \lambda\mathbf{v}$ Directions that only scale	**Eigenvalues** How much scaling $\lambda > 1$: grow, $\lambda < 1$: shrink
Finding Them $\det(\mathbf{A} - \lambda\mathbf{I}) = 0$ Solve the polynomial	**Eigendecomposition** $\mathbf{A} = \mathbf{V}\boldsymbol{\Lambda}\mathbf{V}^{-1}$ Rotate → scale → rotate back
SVD $\mathbf{A} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^{\top}$ Works on ANY matrix!	**Applications** PCA, LoRA, PageRank Compression, stability

Achievement Unlocked: Eigenvector Expert!

You can now:

- Find the "natural directions" of any matrix
- Decompose matrices into simple pieces
- Understand why Netflix knows your taste in movies
- Explain SVD at parties (results may vary)

Next up: Chapter 3—Matrix Decompositions in practice!

Intuition

What You've Truly Learned

You now understand the hidden structure inside matrices:

- Matrices aren't just tables of numbers—they have "personalities" revealed by their eigenvalues
- Every matrix has natural directions where it acts simply (eigenvectors)

- Complex transformations are secretly just rotations and stretches in the right coordinates
- SVD works on ANY matrix and gives the optimal low-rank approximation
- These ideas power everything from Google Search to GPT fine-tuning

You can now look at a matrix and ask: "What are your eigenvalues? What do your singular values tell me? Where are your important directions?"
That's not just math—that's X-ray vision for linear algebra.
"I used to see matrices as scary blocks of numbers. Now I see them as transformations with personalities." — You, hopefully

Chapter 3

Matrix Decompositions: Breaking Things to Understand Them Better

3.1 The IKEA Philosophy of Mathematics

3.1.1 Why Breaking Things Apart Is Actually Smart

You know what IKEA figured out that made them billions of dollars? Shipping fully assembled furniture is expensive and inefficient. A fully built wardrobe? Good luck fitting that in your car. But if you break a bookshelf into flat pieces, suddenly you can fit 50 of them in one truck. Genius!

Matrix decompositions are the same idea, except instead of making shipping easier, they make computation faster, storage cheaper, and understanding deeper. And unlike IKEA furniture, you won't end up with mysterious leftover screws.

The core insight: A big complicated matrix can often be broken into a product of simpler matrices. And once it's broken apart, you can:

- Store it more efficiently (compression!)
- Compute with it faster (optimization!)
- Understand what it's doing (interpretation!)
- Fix parts without rebuilding everything (adaptation!)

Intuition

Think of a matrix decomposition like this:
Before: "This 1000×1000 matrix does... something complicated. It has a million numbers in it. Good luck understanding it, buddy."

After: "Oh! It's actually just a rotation ($\mathbf{Q}$) followed by a stretch ($\mathbf{D}$) followed by another rotation ($\mathbf{Q}^\top$). Now I get it! It's like a cosmic yoga pose for vectors!"
Decompositions turn mysterious black boxes into transparent, modular components. It's like opening up a magic trick and seeing there's just three simple steps that create the illusion.

3.1.2 A Real-World Analogy: The Recipe Book

Imagine you have a super complicated recipe for a fancy wedding cake. It's 47 steps long, uses 23 ingredients, and takes 6 hours. Terrifying, right?

But what if I told you it's actually just:

1. Make a basic vanilla cake (you know how to do this!)
2. Make buttercream frosting (easy peasy!)
3. Stack them and decorate (the fun part!)

Each piece is simple. The combination creates something impressive. That's exactly what matrix decomposition does—it breaks scary math into friendly, bite-sized pieces.

Connection to LLMs

Why LLMs love decompositions (and why you should too): GPT-3 has **175 billion parameters**. That's 175,000,000,000 numbers. If you printed them out, you'd need about 35 million pages. That's a LOT of numbers to store and compute with.
Decompositions let us:

- **Compress models:** Store a 4096×4096 matrix as two 4096×64 matrices. That's like compressing a library into a pamphlet! (This is called LoRA, and it's magical.)
- **Speed up inference:** Compute faster by exploiting structure. Your chatbot responds quicker!
- **Fine-tune efficiently:** Update only small low-rank pieces instead of everything. Like renovating your kitchen instead of rebuilding your house.
- **Understand attention:** Decompose attention matrices to see what patterns the model learned.

Let's learn the main decompositions and why they matter!

3.2 Cholesky Decomposition: The Matrix Square Root

3.2.1 Taking the Square Root of a Matrix (Yes, Really!)

Remember in elementary school when you learned that $5^2 = 25$, so $\sqrt{25} = 5$? You were basically taught that some operations can be reversed. Squaring has a reverse called square rooting.

Here's a wild thought: **What if we could do the same thing for matrices?**

Spoiler alert: We can! And it's called the Cholesky decomposition. (Named after André-Louis Cholesky, a French military officer who did math in his spare time because apparently fighting in WWI wasn't exciting enough.)

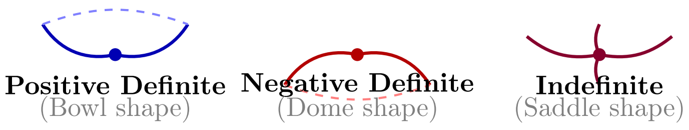
The setup: You have a special kind of matrix $\mathbf{A}$ that's:

- **Symmetric:** $\mathbf{A} = \mathbf{A}^\top$ (it's a mirror image across the diagonal—like a butterfly's wings)
- **Positive definite:** $\mathbf{x}^\top \mathbf{A}\mathbf{x} > 0$ for all non-zero $\mathbf{x}$

Wait, what does "positive definite" even mean?

Great question! Think of it like this: Imagine you're standing at the bottom of a bowl (like a salad bowl, not a Super Bowl). No matter which direction you walk, you go uphill. That's positive definite—it curves upward in *every* direction.

A negative definite matrix is like standing on a dome—every direction goes downhill. An indefinite matrix is like a saddle—some directions go up, some go down.



Positive definite matrices show up everywhere in machine learning, especially when dealing with covariances (how things vary together), Hessians (second derivatives in optimization), and kernel matrices (similarity measures).

Definition 3.1: Cholesky Decomposition

For a positive definite matrix $\mathbf{A}$, there exists a **unique** lower triangular matrix $\mathbf{L}$ such that:

$$\mathbf{A} = \mathbf{L}\mathbf{L}^\top$$

We call $\mathbf{L}$ the Cholesky factor of $\mathbf{A}$.
In plain English: "$\mathbf{L}$ is like the square root of $\mathbf{A}$ because $\mathbf{L}$ times its transpose gives you $\mathbf{A}$."
It's like saying: "What matrix, when multiplied by its own reflection, gives us back $\mathbf{A}$?" And the answer is $\mathbf{L}$!

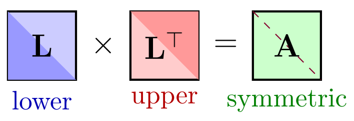
What's a lower triangular matrix?

It's a matrix where everything *above* the diagonal is zero. Like a staircase going down:

$$\mathbf{L} = \begin{bmatrix} \text{something} & 0 & 0 \\ \text{something} & \text{something} & 0 \\ \text{something} & \text{something} & \text{something} \end{bmatrix}$$

The zeros above the diagonal make these matrices super easy to work with. Solving equations with triangular matrices is like unwrapping a present one layer at a time—each step reveals the next.

Numbers:
$5 \times 5 = 25$
$\sqrt{25} = 5$
$\downarrow$
Matrices:
$\mathbf{L}\mathbf{L}^\top = \mathbf{A}$



3.2.2 Let's Actually Do One! (Don't Panic)

I know what you're thinking: "This sounds complicated." But it's actually not! Let's work through an example step by step, like following a recipe.

Example

Let's find the Cholesky decomposition of:

$$\mathbf{A} = \begin{bmatrix} 4 & 2 \\ 2 & 3 \end{bmatrix}$$

First, let's verify this is positive definite:

- It's symmetric? ✓ (2 appears in both off-diagonal spots)
- The diagonal entries are positive? ✓ (4 and 3 are both positive)
- Determinant is positive? $4 \times 3 - 2 \times 2 = 12 - 4 = 8 > 0$ ✓

Great! We can proceed. (If any of these failed, there would be no Cholesky decomposition—like trying to take the square root of a negative number.)

Now, we want to find L such that:

$$\begin{bmatrix} 4 & 2 \\ 2 & 3 \end{bmatrix} = \begin{bmatrix} \ell_{11} & 0 \\ \ell_{21} & \ell_{22} \end{bmatrix} \begin{bmatrix} \ell_{11} & \ell_{21} \\ 0 & \ell_{22} \end{bmatrix}$$

Notice how $\mathbf{L}^\top$ is just $\mathbf{L}$ flipped across the diagonal? The zeros switch from bottom-left to top-right.

Step 1: Multiply out the right side

Let's do the matrix multiplication:

$$\mathbf{L}\mathbf{L}^\top = \begin{bmatrix} \ell_{11} \cdot \ell_{11} + 0 \cdot 0 & \ell_{11} \cdot \ell_{21} + 0 \cdot \ell_{22} \\ \ell_{21} \cdot \ell_{11} + \ell_{22} \cdot 0 & \ell_{21} \cdot \ell_{21} + \ell_{22} \cdot \ell_{22} \end{bmatrix} = \begin{bmatrix} \ell_{11}^2 & \ell_{11}\ell_{21} \\ \ell_{11}\ell_{21} & \ell_{21}^2 + \ell_{22}^2 \end{bmatrix}$$

Step 2: Match components (like a puzzle!)

Now we just match up corresponding entries:

- **Top-left:** $\ell_{11}^2 = 4$

  What number squared gives 4? That's $\ell_{11} = 2$ (we take the positive root)

- **Top-right (or bottom-left, they're equal):** $\ell_{11}\ell_{21} = 2$

  We know $\ell_{11} = 2$, so $2 \cdot \ell_{21} = 2$, which means $\ell_{21} = 1$

- **Bottom-right:** $\ell_{21}^2 + \ell_{22}^2 = 3$

  We know $\ell_{21} = 1$, so $1 + \ell_{22}^2 = 3$, which means $\ell_{22}^2 = 2$, so $\ell_{22} = \sqrt{2}$

Step 3: Assemble the answer!

$$\mathbf{L} = \begin{bmatrix} 2 & 0 \\ 1 & \sqrt{2} \end{bmatrix}$$

Step 4: Let's double-check (always a good idea!)

$$\begin{bmatrix} 2 & 0 \\ 1 & \sqrt{2} \end{bmatrix} \begin{bmatrix} 2 & 1 \\ 0 & \sqrt{2} \end{bmatrix} = \begin{bmatrix} 4 & 2 \\ 2 & 1+2 \end{bmatrix} = \begin{bmatrix} 4 & 2 \\ 2 & 3 \end{bmatrix}$$

Perfect! ✓ We found the matrix square root!

Intuition

Why did that work?
We essentially "built up" the matrix from scratch. Starting from the top-left corner, we figured out each entry one at a time. The triangular structure meant each new entry only depended on things we'd already computed.
It's like solving a Sudoku where each number you fill in reveals the next one!

3.2.3 Why This Is Ridiculously Useful (The Practical Magic)

"Okay," you say, "I can find the Cholesky factor. But why would I want to?"

Great question! Here are three killer applications:

Application 1: Solving Systems of Equations FAST
Suppose you need to solve $\mathbf{Ax} = \mathbf{b}$ (find $\mathbf{x}$ given $\mathbf{A}$ and $\mathbf{b}$).
The slow way: Compute $\mathbf{A}^{-1}$ and multiply: $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$. This is expensive and can be numerically unstable.
The Cholesky way: Since $\mathbf{A} = \mathbf{LL}^\top$, we rewrite as $\mathbf{LL}^\top\mathbf{x} = \mathbf{b}$
Now split it into two easy problems:

1. Solve $\mathbf{Ly} = \mathbf{b}$ for $\mathbf{y}$ (easy—$\mathbf{L}$ is lower triangular!)
2. Solve $\mathbf{L}^\top\mathbf{x} = \mathbf{y}$ for $\mathbf{x}$ (also easy—$\mathbf{L}^\top$ is upper triangular!)

Why is triangular easy? Because you can solve it one row at a time!
For a lower triangular system:

$$\begin{aligned} 2x_1 = 6 \quad &\Rightarrow \quad x_1 = 3 \quad \text{(just divide!)} \\ x_1 + 3x_2 = 9 \quad &\Rightarrow \quad 3 + 3x_2 = 9 \quad \Rightarrow \quad x_2 = 2 \quad \text{(substitute and solve!)} \end{aligned}$$

Each row only involves the variables you've already found, plus one new one. It's like a ladder—climb one rung at a time!
Speed comparison: For an $n \times n$ matrix:

- General solver: About n^3 operations
- Cholesky + triangular solves: About $\frac{1}{3}n^3$ operations for factoring, then n^2 per solve

If you need to solve many systems with the same $\mathbf{A}$ but different $\mathbf{b}$ vectors (common in practice!), Cholesky is a huge win. Factor once, solve cheaply many times!

Application 2: Generating Random Samples (Making Randomness Correlated)

This one's cool. Suppose you want to generate random samples from a multivariate Gaussian distribution $\mathcal{N}(\mu, \mathbf{\Sigma})$.

The problem: You can easily generate independent random numbers (like rolling dice). But what if you want correlated random numbers (like height and weight, which tend to go together)?

The Cholesky solution:

1. Compute Cholesky: $\mathbf{\Sigma} = \mathbf{L}\mathbf{L}^\top$
2. Sample independent standard normals: $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ (just random numbers, uncorrelated)
3. Transform: $\mathbf{x} = \mu + \mathbf{L}\mathbf{z}$

Now $\mathbf{x}$ has the exact right mean and covariance!

What's happening intuitively? $\mathbf{L}$ is like a "correlation machine." You feed in independent randomness, and it mixes them together to create the right correlations.

Imagine you have two random number generators: one for "general health" and one for "exercise amount." $\mathbf{L}$ says: "To get height, take 90% of health and 10% of exercise. To get weight, take 80% of health and 20% of exercise." Now height and weight are correlated!

Application 3: Checking if a Matrix is "Valid"

Sometimes you compute a covariance matrix and wonder: "Is this actually a valid covariance matrix?"

A valid covariance matrix must be positive semi-definite (the bowl shape we discussed). A quick test:

1. Try to compute the Cholesky decomposition
2. If it works $\rightarrow$ Valid! ✓
3. If you get a negative number under a square root $\rightarrow$ Invalid! *imes*

It's like a "positive definiteness detector." Very handy!

Connection to LLMs

In machine learning, Cholesky is everywhere:
Gaussian Processes: These powerful models for regression and uncertainty quantification use covariance matrices extensively. Cholesky is the workhorse for all computations!
Optimization (Newton's Method): When you're optimizing a

function, you often need to use second derivatives (the Hessian matrix). If it's positive definite, Cholesky gives you a fast, stable way to take steps.
Variational Autoencoders (VAEs): These generative models need to sample from Gaussian distributions—hello, Cholesky!
Bayesian Deep Learning: When estimating uncertainty in neural networks, you often work with covariance matrices of weights. Cholesky makes this tractable.

3.3 LU Decomposition: The Systematic Eliminator

3.3.1 Remember Gaussian Elimination from School?

Pop quiz: In high school algebra, how did you solve systems of equations?

If you said "Gaussian elimination" (or "that thing where you add and subtract rows to make zeros"), congratulations! You already understand LU decomposition. It's just that process, gift-wrapped in a fancy matrix package!

The basic idea: Take any matrix (doesn't need to be symmetric or positive definite—any square matrix works!) and break it into two triangular matrices.

Definition 3.2: LU Decomposition

For a matrix $\mathbf{A}$, we can write:

$$\mathbf{A} = \mathbf{L}\mathbf{U}$$

where:

- $\mathbf{L}$ is **L**ower triangular (zeros above diagonal)—the "multipliers" from elimination
- $\mathbf{U}$ is **U**pper triangular (zeros below diagonal)—the "end result" after elimination

Sometimes we need to swap rows first (called "pivoting"), so we write:

$$\mathbf{P}\mathbf{A} = \mathbf{L}\mathbf{U}$$

where $\mathbf{P}$ is a permutation matrix (it just swaps rows around—no actual math, just rearranging).

Why "LU"? Because mathematicians are very creative with names. L for Lower, U for Upper. Groundbreaking stuff.

3.3.2 Let's Do Gaussian Elimination (And Watch LU Appear!)

Example

Let's decompose this matrix:

$$\mathbf{A} = \begin{bmatrix} 2 & 1 & 1 \\ 4 & 3 & 3 \\ 8 & 7 & 9 \end{bmatrix}$$

Goal: Turn this into an upper triangular matrix by making zeros below the diagonal.
Step 1: Eliminate the first column (below the diagonal)
Look at position (2,1). It has a 4, but we want a 0.
How do we make it zero? Row 2 minus 2 times Row 1:

$$\begin{bmatrix} 4 & 3 & 3 \end{bmatrix} - 2 \times \begin{bmatrix} 2 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 1 \end{bmatrix}$$

Important: Remember that multiplier (2)! We'll need it later.
Now our matrix looks like:

$$\begin{bmatrix} 2 & 1 & 1 \\ 0 & 1 & 1 \\ 8 & 7 & 9 \end{bmatrix}$$

Next, position (3,1) has an 8. We want zero.
Row 3 minus 4 times Row 1:

$$\begin{bmatrix} 8 & 7 & 9 \end{bmatrix} - 4 \times \begin{bmatrix} 2 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 3 & 5 \end{bmatrix}$$

Remember that multiplier too (4)!
Now:

$$\begin{bmatrix} 2 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 3 & 5 \end{bmatrix}$$

First column is done! ✓
Step 2: Eliminate the second column (below the diagonal)
Position (3,2) has a 3. We want zero.
Row 3 minus 3 times Row 2:

$$\begin{bmatrix} 0 & 3 & 5 \end{bmatrix} - 3 \times \begin{bmatrix} 0 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 2 \end{bmatrix}$$

Multiplier: 3
Final result:

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

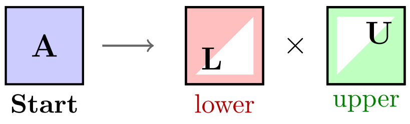
This is our **U**—upper triangular!
Step 3: Build L from the multipliers
Remember all those multipliers we saved? (2, 4, 3)
They go into **L** in the exact positions where we created zeros:

$$\mathbf{L} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 4 & 3 & 1 \end{bmatrix}$$

The diagonal is all 1s (we didn't modify the diagonal), and the multipliers fill in below.
Step 4: Verify!

$$\mathbf{LU} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 4 & 3 & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 2 \end{bmatrix} = \begin{bmatrix} 2 & 1 & 1 \\ 4 & 3 & 3 \\ 8 & 7 & 9 \end{bmatrix} = \mathbf{A}$$

It works! ✓



The diagram shows: Full matrix → eliminate to get upper triangular **U**, *save multipliers in lower triangular* **L**.

Intuition

What's the deep meaning here?
L and **U** together encode **both the answer and the process**!
Think of it like a cooking show:

- **U**: The finished dish (the simplified, easy-to-work-with result)
- **L**: The recipe (all the steps we took to get there)

By keeping the recipe (**L**), we can "undo" the process whenever we want. This is incredibly useful!

3.3.3 Why Use LU Instead of Just Solving Directly?

The killer feature: Once you have $\mathbf{A} = \mathbf{LU}$, you can solve $\mathbf{Ax} = \mathbf{b}$ for MANY different $\mathbf{b}$ vectors super quickly!

Here's the scenario: You're a data scientist, and your boss says: "Solve this system... actually, now solve it with this other **b**... wait, now this one... and this one..."

Without LU:

- Each solve costs $O(n^3)$ operations
- 100 different $\mathbf{b}$ vectors = 100 $\times$ expensive = very slow

With LU:

- Compute $\mathbf{L}$ and $\mathbf{U}$ once: $O(n^3)$ (expensive, but do it once!)
- Each solve after that: $O(n^2)$ (cheap!)
- 100 different $\mathbf{b}$ vectors = 1 $\times$ expensive + 100 $\times$ cheap = fast!

It's like building a highway. Yes, construction is expensive. But once it's built, every trip is fast!

Connection to LLMs

In deep learning:
LU decomposition isn't the star of the show (that's probably SVD or low-rank factorization), but it's a reliable supporting actor:
Normalizing flows: These generative models need invertible transformations. Triangular matrices are automatically invertible, so LU factors are perfect!
Numerical stability: When things go wrong in gradient computations (NaN errors, anyone?), understanding LU decomposition helps debug.
Sparse matrices: In huge systems, sparse LU solvers can handle problems that would otherwise be impossible.
Matrix inverses: Never compute $\mathbf{A}^{-1}$ directly! Use LU decomposition instead.

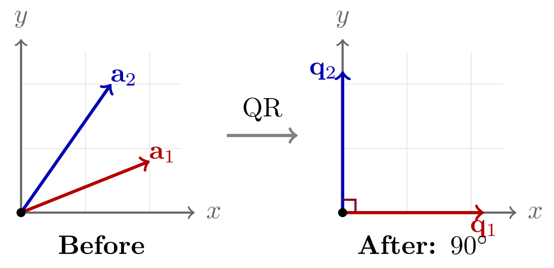
3.4 QR Decomposition: Finding Perpendicular Directions

3.4.1 The Gram-Schmidt Process (Making Things Perpendicular)

Imagine you're a photographer with a tripod, but the legs are at weird angles. The photo comes out tilted!

What you want is three legs pointing in perpendicular directions: forward, right, up. That's stability!

QR decomposition does exactly this for vectors. It takes vectors pointing in random directions and reorganizes them to be perpendicular—while still spanning the same space.



Why is perpendicular so great?
Think about directions:

- **Perpendicular (orthogonal):** "Go north, then go east." These are independent—going north doesn't affect your east-west position at all!
- **Not perpendicular:** "Go northeast, then go east-northeast." Uh, wait, how much overlap is there? This is confusing!

Perpendicular directions are *independent*. They don't interfere with each other. This makes everything simpler!

Definition 3.3: QR Decomposition

For any matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ (doesn't even need to be square!):

$$\mathbf{A} = \mathbf{QR}$$

where:

- $\mathbf{Q}$ is **orthogonal**: $\mathbf{Q}^\top \mathbf{Q} = \mathbf{I}$ (columns are perpendicular unit vectors)

- $\mathbf{R}$ is **upper triangular** (zeros below diagonal)

In plain English: Take the columns of $\mathbf{A}$, make them perpendicular (that's $\mathbf{Q}$), and keep track of how to get back to the original (that's $\mathbf{R}$).

3.4.2 The Gram-Schmidt Process: A Recipe for Orthogonalization

The algorithm to find $\mathbf{Q}$ is called **Gram-Schmidt**, named after Jørgen Pedersen Gram and Erhard Schmidt. (Mathematicians get to put their names on things they discover. It's a perk of the job.)

The idea is simple:

1. Take the first vector. Normalize it (make it length 1). Done!
2. Take the second vector. Remove any part that points in the first vector's direction. Normalize. Done!
3. Take the third vector. Remove any parts that point in the first or second directions. Normalize. Done!
4. Keep going until you've processed all vectors.

It's like cleaning your room by picking one direction and putting away everything in that direction first, then the next direction, and so on.

Example

Let's QR decompose:

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

This has two columns (vectors in 3D space). We'll make them perpendicular.

Step 1: First column becomes first Q column

Our first column is $\mathbf{a}_1 = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}$

Its length is $\|\mathbf{a}_1\| = \sqrt{1^2 + 1^2 + 0^2} = \sqrt{2}$

Normalize it (make length 1):

$$\mathbf{q}_1 = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \\ 0 \end{bmatrix}$$

Step 2: Make second column perpendicular to first

Our second column is $\mathbf{a}_2 = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$

First, find how much of $\mathbf{a}_2$ points in the $\mathbf{q}_1$ direction (the "projection"):

$$\text{projection} = (\mathbf{a}_2 \cdot \mathbf{q}_1)\mathbf{q}_1 = \left(\frac{1}{\sqrt{2}} \cdot 1 + \frac{1}{\sqrt{2}} \cdot 0 + 0 \cdot 1\right)\mathbf{q}_1 = \frac{1}{\sqrt{2}}\mathbf{q}_1$$

Subtract the projection to get the perpendicular part:

$$\mathbf{u}_2 = \mathbf{a}_2 - \frac{1}{\sqrt{2}}\mathbf{q}_1 = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} - \frac{1}{\sqrt{2}} \cdot \begin{bmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} - \begin{bmatrix} 1/2 \\ 1/2 \\ 0 \end{bmatrix} = \begin{bmatrix} 1/2 \\ -1/2 \\ 1 \end{bmatrix}$$

Now normalize:

$$\|\mathbf{u}_2\| = \sqrt{(1/2)^2 + (-1/2)^2 + 1^2} = \sqrt{1/4 + 1/4 + 1} = \sqrt{3/2}$$

$$\mathbf{q}_2 = \frac{1}{\sqrt{3/2}} \begin{bmatrix} 1/2 \\ -1/2 \\ 1 \end{bmatrix} = \begin{bmatrix} 1/\sqrt{6} \\ -1/\sqrt{6} \\ 2/\sqrt{6} \end{bmatrix}$$

Verify they're perpendicular:

$$\mathbf{q}_1 \cdot \mathbf{q}_2 = \frac{1}{\sqrt{2}} \cdot \frac{1}{\sqrt{6}} + \frac{1}{\sqrt{2}} \cdot \frac{-1}{\sqrt{6}} + 0 \cdot \frac{2}{\sqrt{6}} = \frac{1}{\sqrt{12}} - \frac{1}{\sqrt{12}} + 0 = 0 \quad \checkmark$$

Perpendicular! (Dot product = 0 means perpendicular. It's like a test!)

Step 3: Build Q and R

$$\mathbf{Q} = \begin{bmatrix} 1/\sqrt{2} & 1/\sqrt{6} \\ 1/\sqrt{2} & -1/\sqrt{6} \\ 0 & 2/\sqrt{6} \end{bmatrix}$$

R contains the projection coefficients and norms:

$$\mathbf{R} = \begin{bmatrix} \sqrt{2} & 1/\sqrt{2} \\ 0 & \sqrt{3/2} \end{bmatrix}$$

(The diagonal entries are the lengths before normalizing. The off-diagonal entries are the projection coefficients.)

Intuition

The Shadow Analogy
Here's a nice way to think about Gram-Schmidt:
Imagine $\mathbf{q}_1$ is a flagpole sticking straight up. Now you shine a light from directly above $\mathbf{a}_2$. The shadow of $\mathbf{a}_2$ on the ground is the part that's aligned with $\mathbf{q}_1$.
What's left—the part that sticks up from the shadow—is $\mathbf{q}_2$. It's whatever wasn't captured by $\mathbf{q}_1$!
Each new vector is what's "new" after accounting for all previous directions.

3.4.3 Why Orthogonal Matrices Are Magical

Orthogonal matrices ($\mathbf{Q}$) have superpowers:

Superpower 1: Free inverses!

For most matrices, computing the inverse is expensive (about n^3 operations). But for orthogonal matrices:

$$\mathbf{Q}^{-1} = \mathbf{Q}^\top$$

That's it! Just flip across the diagonal. No computation needed. It's like getting a free dessert.

Superpower 2: Length preservation!

When you multiply a vector by $\mathbf{Q}$:

$$\|\mathbf{Qx}\| = \|\mathbf{x}\|$$

The length stays the same! $\mathbf{Q}$ is a **rotation** (possibly with reflection). It spins things around but doesn't stretch or squish.

This is huge for numerical stability. Rounding errors can grow and shrink vectors, causing chaos. But orthogonal matrices are perfectly stable!

Superpower 3: Condition number = 1

The "condition number" measures how much small errors get amplified. For orthogonal matrices, it's exactly 1—the best possible. No error amplification!

Connection to LLMs

QR in machine learning:
1. Least squares (linear regression): Fitting a line to data means solving $\mathbf{Ax} \approx \mathbf{b}$. QR gives the most numerically stable solution. If you're ever getting weird numbers from linear regression, switching to QR-based solvers often fixes it!
2. Computing eigenvalues: The "QR algorithm" repeatedly ap-

plies QR decomposition to find eigenvalues. It's a cornerstone of numerical linear algebra!
3. Orthogonal neural networks: Some architectures constrain weight matrices to be orthogonal to prevent gradient vanishing/exploding. They maintain orthogonality using QR!
4. Transformers and attention: While attention doesn't directly use QR, the idea of normalizing vectors before computing attention is the same spirit.
5. Optimization on manifolds: When weights must be orthogonal (helps generalization), QR is your friend!

3.5 Rank Factorization: The Low-Rank Art

3.5.1 Most Matrices Are Secretly Simpler Than They Look

Here's a profound observation that powers modern AI efficiency:

A 1000×1000 matrix has a million entries. But what if it only has rank 10?

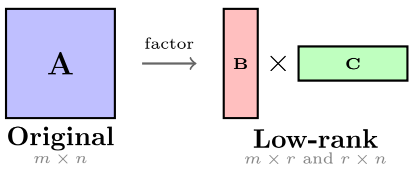
That means all its information actually lives in a tiny 10-dimensional subspace! The other 990 dimensions are redundant—just linear combinations of the first 10.

Analogy time!

Imagine a 4K ultra-HD movie file. Massive, right? Gigabytes of data!

But what if the entire movie is just a person talking in front of a gray wall? The background never changes. The person barely moves. There's almost no variation!

You could compress this movie down to almost nothing because there's not much actual *information*. That's what "low rank" means for matrices—there's not as much information as the size suggests.



Let's do the math with real numbers:

For a 4096×4096 matrix:

- **Full storage:** $4096 \times 4096 = 16{,}777{,}216$ numbers

- **Rank-64 storage:** $4096 \times 64 + 64 \times 4096 = 262{,}144 + 262{,}144 = 524{,}288$ numbers
- **Compression ratio:** $16{,}777{,}216/524{,}288 = 32\times$ smaller!

And if we use rank-8 instead of rank-64? The compression ratio becomes $256\times$!

This is why LoRA (Low-Rank Adaptation) can fine-tune ChatGPT on your laptop. You're not storing millions of parameters—just thousands.

Definition 3.4: Low-Rank Factorization

A matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$ with rank r can be written as:

$$\mathbf{A} = \mathbf{BC}$$

where $\mathbf{B} \in \mathbb{R}^{m\times r}$ (tall and skinny) and $\mathbf{C} \in \mathbb{R}^{r\times n}$ (short and wide).
Storage savings:

- Full matrix: $m \times n$ numbers
- Factored form: $m \times r + r \times n = r(m + n)$ numbers

If $r \ll \min(m, n)$, this is a MASSIVE savings!
Memory formula: Savings factor $= \frac{mn}{r(m+n)}$
For $m = n = 4096$ and $r = 8$: Savings $= \frac{4096^2}{8\times 8192} = 256\times$

3.5.2 A Simple Example: The Rank-1 Matrix

Let's see the most extreme case: a rank-1 matrix.

Example

Consider:

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \\ 3 & 6 & 9 \end{bmatrix}$$

Look closely. Notice anything?

- Row 2 = 2 × Row 1
- Row 3 = 3 × Row 1

Every row is just a scaled version of the first row! This matrix is "secretly" very simple—it's rank 1.

We can factor it as:

$$\mathbf{A} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 \end{bmatrix} = \mathbf{u}\mathbf{v}^\top$$

Storage comparison:

- Full matrix: 9 numbers
- Factored: 3 + 3 = 6 numbers (33% savings)

"Big deal," you say, "I saved 3 numbers."
Okay, but what about a 1000×1000 rank-1 matrix?

- Full matrix: 1,000,000 numbers
- Factored: 1,000 + 1,000 = 2,000 numbers
- **Savings: 99.8%!!!**

Now THAT's compression!

Intuition

Why does rank-1 mean "one pattern"?
A rank-1 matrix is an "outer product" of two vectors. Every element is just $a_{ij} = u_i \cdot v_j$.
Think of it like a multiplication table! The 3×3 matrix where every entry is the product of its row and column numbers:

$$\begin{bmatrix} 1 \times 1 & 1 \times 2 & 1 \times 3 \\ 2 \times 1 & 2 \times 2 & 2 \times 3 \\ 3 \times 1 & 3 \times 2 & 3 \times 3 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \\ 3 & 6 & 9 \end{bmatrix}$$

It looks like 9 independent numbers, but it's really just 2 patterns (rows and columns) multiplied together!

3.5.3 Low-Rank Approximation: When Close Enough Is Good Enough

Real matrices usually aren't exactly low-rank. But they might be *approximately* low-rank!

If most of the "energy" of a matrix is concentrated in a few dimensions, we can throw away the rest and barely notice.

The SVD connection: Remember SVD from Chapter 2?

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top = \sum_{i=1}^{r} \sigma_i \mathbf{u}_i \mathbf{v}_i^\top$$

This writes $\mathbf{A}$ as a sum of rank-1 matrices, weighted by singular values σ_i.

The key insight: Singular values are ordered from largest to smallest: $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_r$

If $\sigma_1 = 100$ and $\sigma_{10} = 0.001$, then the first few terms carry almost all the information!

Approximation: Keep only the k largest singular values:

$$\mathbf{A}_k = \sum_{i=1}^{k} \sigma_i \mathbf{u}_i \mathbf{v}_i^\top \approx \mathbf{A}$$

Error: The approximation error is exactly σ_{k+1}—the first singular value you dropped.

If σ_{k+1} is tiny, you lost almost nothing!

Example

Image compression example:
A 1024×1024 grayscale image = 1 million pixels.
But after SVD, suppose the singular values are:

- $\sigma_1 = 5000, \sigma_2 = 4500, \ldots, \sigma_{50} = 100$
- $\sigma_{51} = 2, \sigma_{52} = 1.8, \ldots, \sigma_{1024} = 0.001$

The first 50 singular values are big; the rest are tiny!
Keep only rank 50:

- Original: 1,048,576 numbers
- Rank-50: $1024 \times 50 + 50 \times 1024 = 102,400$ numbers
- Compression: 10× smaller!
- Visual quality: Probably indistinguishable to humans

This is basically how JPEG works (with some technical differences)!

3.5.4 LoRA: The Revolutionary Application

Now we get to the really exciting part—how low-rank factorization is revolutionizing AI!

Connection to LLMs

LoRA: Low-Rank Adaptation for LLMs
This is THE technique for fine-tuning giant language models efficiently. It's why you can fine-tune LLaMA on your laptop!
The problem: GPT-3 has 175 billion parameters. Fine-tuning all of them requires:

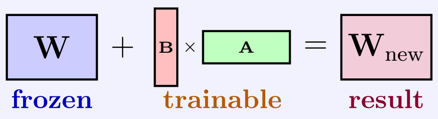
- Massive GPU memory (to store all gradients)
- Tons of compute time
- Lots of money

Fine-tuning all parameters is like renovating every room in a mansion when you just wanted a new bathroom.
LoRA's brilliant insight: Most of the weight update during fine-tuning is *low-rank*!
You don't need to change everything—just a few key directions in weight space. It's like the model saying: "I already know how to speak English. I just need small adjustments to sound like a pirate. Arrr!"
How it works:
Instead of updating **W** directly, learn a low-rank update:

$$\mathbf{W}_{\text{new}} = \mathbf{W}_{\text{original}} + \Delta\mathbf{W} = \mathbf{W}_{\text{original}} + \mathbf{BA}$$

where:

- $\mathbf{W}_{\text{original}}$ is frozen (don't touch!)
- $\mathbf{B} \in \mathbb{R}^{d\times r}$ is a skinny matrix (trained)
- $\mathbf{A} \in \mathbb{R}^{r\times d}$ is a short matrix (trained)
- $r \ll d$ (like $r = 8$ when $d = 4096$)



The numbers are mind-blowing:

Method	Parameters	Savings
Full fine-tuning (4096×4096)	16,777,216	—
LoRA rank 64	524,288	32×
LoRA rank 16	131,072	128×
LoRA rank 8	65,536	256×
LoRA rank 4	32,768	512×

With rank 8, you're training 256× fewer parameters! And the crazy thing? **It usually works just as well!**
Why does LoRA work so well?
The key insight is that fine-tuning doesn't need to change everything—it just needs to adjust a few key directions.
Think of it like this: GPT-3 already knows how to write English. Fine-tuning for customer service just means: "Be polite. Refer to our products. End with 'How else can I help?'" That's not a complete rewrite—it's a few tweaks!
Mathematically, the "intrinsic dimensionality" of the fine-tuning task is low. The model's weights live in a 175-billion-dimensional space, but the useful updates only span maybe 64 dimensions. LoRA exploits this!

3.6 Putting It All Together: The Decomposition Toolkit

3.6.1 Which Decomposition Should I Use?

Here's your ultimate cheat sheet. Print it out. Tattoo it on your arm. Whatever works!

Decomposition	When to Use	Superpower	Vibe
Cholesky	Symmetric positive definite	Fastest solver, stable	Square root
LU	Square matrices, many solves	Factor once, solve many	Elimination
QR	Least squares problems	Most numerically stable	Perpendicular
SVD	Any matrix, low-rank approx	Swiss army knife	The GOAT
Eigen	Square, understanding dynamics	See stretches & rotations	Diagonal
Low-rank	Compression, LoRA	Massive storage savings	Less is more

3.6.2 Quick Decision Tree

Confused about which decomposition to use? Follow this!

1. **Is the matrix symmetric positive definite?** (Covariance? Hessian?)

- Yes → **Cholesky**. Don't think, just do it.

2. **Are you solving Ax = b for many different b?**
   - Yes → **LU** (or Cholesky if symmetric PD)
3. **Are you doing least squares / linear regression?**
   - Yes → **QR**. It's the most stable.
4. **Do you need a low-rank approximation?**
   - Yes → **SVD** (gives the best approximation!)
5. **Do you need to compress or adapt a neural network?**
   - Yes → **Low-rank factorization** (LoRA style!)
6. **Do you want to understand the geometry (stretches, rotations)?**
   - Yes → **SVD** or **Eigendecomposition**

3.6.3 The Grand Philosophy

All these decompositions share one beautiful philosophy:

"Break complex things into simple, interpretable pieces."

Each decomposition has its own flavor of simplicity:

- **Cholesky:** "This positive definite matrix is just a lower triangular matrix times its reflection!"
- **LU:** "Any matrix is just lower triangular × upper triangular. Elimination packaged neatly!"
- **QR:** "Any matrix is just a rotation × upper triangular. Perpendicularity achieved!"
- **SVD:** "Any matrix is just rotation × stretch × rotation. The ultimate breakdown!"
- **Eigen:** "Square matrices stretch along their eigenvectors. That's it!"
- **Low-rank:** "This huge matrix is secretly just two skinny matrices having a conversation."

Once you see the pieces, you can:

- **Understand:** "Oh, it's mostly stretching 5× in this direction!"
- **Compute:** "I can work with triangular matrices WAY faster!"
- **Compress:** "I only need 64 dimensions, not 4096!"
- **Adapt:** "I'll update just this tiny low-rank part!"
- **Debug:** "The condition number is huge because of that one tiny singular value..."

3.7 Practice Problems (With Hints!)

Time to test your understanding! Don't worry—these are designed to be doable, not traumatic.

3.7.1 Problem 1: Cholesky Decomposition (Warm-up)

Find the Cholesky decomposition of:

$$\mathbf{A} = \begin{bmatrix} 9 & 6 \\ 6 & 5 \end{bmatrix}$$

Hints:

- Start with $\ell_{11} = \sqrt{9} = 3$
- Then find ℓ_{21} from the off-diagonal
- Finally find ℓ_{22} from the bottom-right

3.7.2 Problem 2: Understanding Rank (Conceptual)

A matrix $\mathbf{W} \in \mathbb{R}^{1024 \times 1024}$ has rank 16.

1. How many numbers to store the full matrix?
2. How many numbers if stored as $\mathbf{W} = \mathbf{BC}$ where $\mathbf{B} \in \mathbb{R}^{1024 \times 16}$ and $\mathbf{C} \in \mathbb{R}^{16 \times 1024}$?
3. What's the compression ratio?
4. **Bonus:** Why might real neural network weights be approximately low-rank?

Hint: Just multiply dimensions!

3.7.3 Problem 3: LoRA Math (Applied)

You're fine-tuning a language model. One weight matrix is 4096×4096.

1. Full fine-tuning: How many parameters?
2. LoRA with rank $r = 4$: How many parameters?
3. LoRA with rank $r = 64$: How many parameters?
4. **Challenge:** At what rank r does LoRA stop saving memory compared to full fine-tuning?

Hint for part 4: Set $r(m + n) = mn$ and solve for r.

3.7.4 Problem 4: Decomposition Choice (Practical)

For each scenario, which decomposition would you choose? Explain briefly.

1. Sampling random points from a multivariate Gaussian distribution
2. Finding the principal components of a dataset
3. Solving 100 systems $\mathbf{A}\mathbf{x}_i = \mathbf{b}_i$ with the same $\mathbf{A}$ (and $\mathbf{A}$ is symmetric positive definite)
4. Compressing a neural network weight matrix to 10% of its original size
5. Checking if a covariance matrix is valid (positive semi-definite)

3.7.5 Problem 5: Low-Rank Intuition (Thought Experiment)

Consider these two 1000×1000 matrices:

Matrix A: Each row is just a random permutation of the numbers 1 to 1000.

Matrix B: Each row is the first row multiplied by some random scalar.

1. What's the rank of Matrix A? (Probably?)
2. What's the rank of Matrix B?
3. Which one compresses better?
4. What real-world data might look like Matrix B?

3.8 Key Takeaways

Let's summarize what you've learned! These are the big ideas:

1. **Decompositions break matrices into simpler pieces**—triangular, orthogonal, diagonal, or low-rank.
2. **Cholesky** ($\mathbf{A} = \mathbf{L}\mathbf{L}^\top$) is the "matrix square root" for symmetric positive definite matrices.
3. **LU** ($\mathbf{A} = \mathbf{L}\mathbf{U}$) packages Gaussian elimination. Factor once, solve many systems!
4. **QR** ($\mathbf{A} = \mathbf{Q}\mathbf{R}$) finds perpendicular directions. Most stable for least squares!
5. **Low-rank factorization** ($\mathbf{A} \approx \mathbf{B}\mathbf{C}$) is the secret to compression.
6. **LoRA uses low-rank updates** to fine-tune LLMs with 100-256$\times$ fewer parameters. This is why you can fine-tune on consumer hardware!
7. **The philosophy:** Don't fight complexity head-on. Break it into pieces, understand each piece, then reassemble.

Connection to LLMs

The Deep Learning Connection: A Summary
Everything we learned connects directly to modern AI:
During Training:

- Eigenvalues of the Hessian tell you about the optimization landscape
- Cholesky factors help with natural gradient and second-order methods
- Low-rank structure in gradients enables efficient distributed training

During Inference:

- Low-rank approximations speed up matrix multiplications
- Quantization (related to low-rank ideas) shrinks model size
- Structured matrices (triangular, orthogonal) enable faster computation

For Fine-tuning:

- LoRA makes adaptation accessible to everyone
- QLoRA combines low-rank with quantization for even smaller footprints
- Adapter methods use low-rank insertions throughout the network

For Understanding:

- SVD of weight matrices reveals what features the model learned
- Eigenanalysis of attention patterns shows what the model "looks at"
- Low-rank structure in representations suggests compressed knowledge

The meta-lesson: Every breakthrough in making LLMs practical—faster, smaller, more adaptable—relies on clever use of matrix decompositions.
When you hear about a new AI efficiency technique, ask yourself: "What decomposition is this exploiting?" You'll usually find one hiding in there!

Congratulations!

You now understand matrix decompositions—the secret weapons of efficient AI!

You can explain:

- Why Cholesky is like taking a matrix square root
- How LU packages Gaussian elimination for reuse
- Why perpendicular directions (QR) are numerically magical
- How low-rank factorization enables 256$\times$ parameter savings in LoRA

That's seriously impressive. Most people never learn this!

Next up: Calculus—the mathematics of change and optimization!

(Where we'll learn how neural networks actually learn by following gradients down the loss landscape. Spoiler: It's like rolling a ball down a bowl... a very high-dimensional bowl!)

Part II

Calculus: The Mathematics of Change

Chapter 4

Single Variable Calculus: The Mathematics of Change

4.1 Introduction: Why Calculus?

Imagine you're training a neural network. At each step, you adjust weights to reduce the error. But how do you know *which direction* to adjust? How much? Calculus tells us how things change and how to optimize them.

Intuition

Calculus is the mathematics of change and accumulation. It answers two fundamental questions:

- **Differentiation**: How fast is something changing?
- **Integration**: How much has accumulated over time?

In deep learning, we use derivatives to find the direction that decreases loss the most—this is gradient descent!

Connection to LLMs

Every time an LLM learns, it's using calculus! Backpropagation is just the chain rule applied systematically. Understanding derivatives is understanding how neural networks learn.

4.2 Limits: The Foundation

4.2.1 What is a Limit?

A **limit** describes the behavior of a function as its input approaches some value.

Definition 4.1: Limit

We write:

$$\lim_{x \to a} f(x) = L$$

if $f(x)$ gets arbitrarily close to L as x gets arbitrarily close to a (but not necessarily equal to a).

Example

$$\lim_{x \to 2}(3x + 1) = 3(2) + 1 = 7$$

The function approaches 7 as x approaches 2.

Example

Consider $f(x) = \frac{x^2-4}{x-2}$ at $x = 2$:
Direct substitution gives $\frac{0}{0}$ (undefined!). But:

$$f(x) = \frac{(x-2)(x+2)}{x-2} = x + 2 \quad \text{for } x \neq 2$$

$$\lim_{x \to 2} f(x) = \lim_{x \to 2}(x + 2) = 4$$

4.2.2 One-Sided Limits

Sometimes we need to approach from one side:

- **Right-hand limit**: $\lim_{x \to a^+} f(x)$ (approaching from values $> a$)
- **Left-hand limit**: $\lim_{x \to a^-} f(x)$ (approaching from values $< a$)

The limit exists if and only if both one-sided limits exist and are equal.

Connection to LLMs

In neural networks, ReLU (Rectified Linear Unit) has different one-sided derivatives:

$$\text{ReLU}(x) = \max(0, x) = \begin{cases} 0 & x < 0 \\ x & x \geq 0 \end{cases}$$

Understanding one-sided limits helps us handle such piecewise functions!

4.3 The Derivative: Rate of Change

4.3.1 Definition

The **derivative** measures the instantaneous rate of change.

Definition 4.2: Derivative

The derivative of f at x is:

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

if this limit exists.
Alternative notation: $\frac{df}{dx}$, $\frac{d}{dx} f(x)$, $Df(x)$

Intuition

The derivative is the slope of the tangent line at a point. It tells you: "If I move a tiny bit in x, how much does $f(x)$ change?"
Think of it as sensitivity: A large derivative means the function is very sensitive to changes in input.

Example

Find the derivative of $f(x) = x^2$:

$$\begin{aligned} f'(x) &= \lim_{h \to 0} \frac{(x+h)^2 - x^2}{h} \\ &= \lim_{h \to 0} \frac{x^2 + 2xh + h^2 - x^2}{h} \\ &= \lim_{h \to 0} \frac{2xh + h^2}{h} \\ &= \lim_{h \to 0} (2x + h) = 2x \end{aligned}$$

So $(x^2)' = 2x$.

4.3.2 Basic Derivative Rules

Theorem 4.1: Power Rule

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

Theorem 4.2: Constant Multiple Rule

$$\frac{d}{dx}[c \cdot f(x)] = c \cdot f'(x)$$

Theorem 4.3: Sum Rule

$$\frac{d}{dx}[f(x) + g(x)] = f'(x) + g'(x)$$

Example

Differentiate $f(x) = 3x^4 - 5x^2 + 7x - 2$:

$$f'(x) = 3(4x^3) - 5(2x) + 7(1) - 0 = 12x^3 - 10x + 7$$

4.4 The Product and Quotient Rules

4.4.1 Product Rule

When multiplying functions, derivatives don't just multiply!

Theorem 4.4: Product Rule

$$\frac{d}{dx}[f(x)g(x)] = f'(x)g(x) + f(x)g'(x)$$

In words: "derivative of first times second, plus first times derivative of second"

Intuition

Think of a rectangle with sides $f(x)$ and $g(x)$. When both sides change slightly, the area change has two parts: one side grows while the other is constant, and vice versa.

Example

Differentiate $h(x) = x^2 \sin(x)$:

$$h'(x) = (x^2)' \sin(x) + x^2(\sin(x))' = 2x \sin(x) + x^2 \cos(x)$$

4.4.2 Quotient Rule

Theorem 4.5: Quotient Rule

$$\frac{d}{dx}\left[\frac{f(x)}{g(x)}\right] = \frac{f'(x)g(x) - f(x)g'(x)}{[g(x)]^2}$$

Mnemonic: "low d-high minus high d-low, over low squared"

Example

Differentiate $f(x) = \frac{x^2+1}{x-1}$:

$$\begin{aligned} f'(x) &= \frac{(2x)(x-1) - (x^2+1)(1)}{(x-1)^2} \\ &= \frac{2x^2 - 2x - x^2 - 1}{(x-1)^2} = \frac{x^2 - 2x - 1}{(x-1)^2} \end{aligned}$$

4.5 The Chain Rule: Derivatives of Compositions

This is THE most important rule for deep learning!

Theorem 4.6: Chain Rule

If $y = f(g(x))$, then:

$$\frac{dy}{dx} = f'(g(x)) \cdot g'(x)$$

Or in Leibniz notation: $\frac{dy}{dx} = \frac{dy}{du} \cdot \frac{du}{dx}$ where $u = g(x)$

Intuition

The chain rule is about “compounding” rates of change. If y changes $3\times$ as fast as u, and u changes $2\times$ as fast as x, then y changes

$3 \times 2 = 6\times$ as fast as x.
This is exactly what happens in a neural network: changes propagate through layers!

Example

Differentiate $f(x) = (x^2 + 1)^{10}$:
Let $u = x^2 + 1$, so $f(x) = u^{10}$.

$$f'(x) = 10u^9 \cdot (2x) = 10(x^2 + 1)^9 \cdot 2x = 20x(x^2 + 1)^9$$

Connection to LLMs

The chain rule IS backpropagation!
In a neural network:

$$\text{Input} \xrightarrow{f_1} \text{Hidden}_1 \xrightarrow{f_2} \text{Hidden}_2 \xrightarrow{f_3} \text{Output}$$

To find how the output changes with respect to the input:

$$\frac{d\text{Output}}{d\text{Input}} = \frac{d\text{Output}}{d\text{Hidden}_2} \cdot \frac{d\text{Hidden}_2}{d\text{Hidden}_1} \cdot \frac{d\text{Hidden}_1}{d\text{Input}}$$

This is the chain rule! Backpropagation computes these derivatives layer by layer.

4.6 Common Functions and Their Derivatives

4.6.1 Exponential and Logarithm

Theorem 4.7: Exponential Derivative

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

The exponential function is its own derivative—beautiful!

Theorem 4.8: Logarithm Derivative

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

Example

Using the chain rule:

$$\frac{d}{dx}e^{x^2} = e^{x^2} \cdot 2x = 2xe^{x^2}$$

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

Connection to LLMs

In machine learning:

- **Softmax** uses exponentials: $\sigma(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$
- **Cross-entropy loss** uses logarithms: $\mathcal{L} = -\sum y_i \log(\hat{y}_i)$
- **Log-likelihood** is everywhere in probabilistic models

4.6.2 Trigonometric Functions

$$\begin{aligned}\frac{d}{dx}\sin(x) &= \cos(x)\\ \frac{d}{dx}\cos(x) &= -\sin(x)\\ \frac{d}{dx}\tan(x) &= \sec^2(x) = \frac{1}{\cos^2(x)}\end{aligned}$$

4.6.3 Activation Functions in Neural Networks

Sigmoid

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

Derivative:

$$\sigma'(x) = \sigma(x)(1-\sigma(x))$$

Intuition

The sigmoid squashes any input to $(0, 1)$—useful for probabilities! But its derivative is small when $|x|$ is large, leading to vanishing gradients.

Hyperbolic Tangent (tanh)

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

Derivative:

$$\tanh'(x) = 1 - \tanh^2(x)$$

ReLU (Rectified Linear Unit)

$$\text{ReLU}(x) = \max(0, x) = \begin{cases} 0 & x < 0 \\ x & x \geq 0 \end{cases}$$

Derivative:

$$\text{ReLU}'(x) = \begin{cases} 0 & x < 0 \\ 1 & x > 0 \\ \text{undefined} & x = 0 \end{cases}$$

In practice, we define $\text{ReLU}'(0) = 0$ or 1.

Connection to LLMs

ReLU revolutionized deep learning! Unlike sigmoid/tanh:

- No vanishing gradient for $x > 0$
- Computationally cheap
- Sparse activation (many neurons output 0)
- But can "die" (always output 0 if weights become negative)

GELU (Gaussian Error Linear Unit)

Modern transformers use GELU:

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

where $\Phi(x)$ is the standard normal CDF.

Approximation:

$$\text{GELU}(x) \approx 0.5x \left(1 + \tanh\left[\sqrt{\frac{2}{\pi}}(x + 0.044715x^3)\right]\right)$$

4.7 Higher-Order Derivatives

The **second derivative** measures how the rate of change itself is changing (curvature):

$$f''(x) = \frac{d^2 f}{dx^2} = \frac{d}{dx}\left[\frac{df}{dx}\right]$$

Example

For $f(x) = x^4$:

$$\begin{aligned} f'(x) &= 4x^3 \\ f''(x) &= 12x^2 \\ f'''(x) &= 24x \\ f^{(4)}(x) &= 24 \end{aligned}$$

Connection to LLMs

Second derivatives are crucial for:

- **Optimization**: Second-order methods use the Hessian (matrix of second derivatives)
- **Curvature**: Tells us if we're at a minimum (positive) or maximum (negative)
- **Newton's method**: Uses f'' to find better descent directions

4.8 Critical Points and Optimization

4.8.1 Critical Points

A **critical point** is where $f'(x) = 0$ or $f'(x)$ doesn't exist.

Theorem 4.9: First Derivative Test

If $f'(x) = 0$ at $x = c$:

- If f' changes from positive to negative at c: local maximum
- If f' changes from negative to positive at c: local minimum
- If f' doesn't change sign: saddle point (inflection point)

Theorem 4.10: Second Derivative Test

If $f'(c) = 0$:

- If $f''(c) > 0$: local minimum (concave up)
- If $f''(c) < 0$: local maximum (concave down)
- If $f''(c) = 0$: test is inconclusive

Example

Minimize $f(x) = x^2 - 4x + 5$:

$$\begin{aligned} f'(x) &= 2x - 4 = 0 \\ x &= 2 \end{aligned}$$

Check: $f''(x) = 2 > 0$, so $x = 2$ is a minimum.
Minimum value: $f(2) = 4 - 8 + 5 = 1$.

Connection to LLMs

This is the essence of training neural networks! We want to find weights that minimize loss:

$$\text{Find } \mathbf{w}^* = \arg\min_{\mathbf{w}} \mathcal{L}(\mathbf{w})$$

We use derivatives to find the direction to adjust weights. In high dimensions, this becomes gradient descent!

4.9 L'Hôpital's Rule

When limits give indeterminate forms like $\frac{0}{0}$ or $\frac{\infty}{\infty}$:

Theorem 4.11: L'Hôpital's Rule

If $\lim_{x\to a} \frac{f(x)}{g(x)}$ gives $\frac{0}{0}$ or $\frac{\infty}{\infty}$, then:

$$\lim_{x\to a} \frac{f(x)}{g(x)} = \lim_{x\to a} \frac{f'(x)}{g'(x)}$$

(if the right-hand limit exists)

Example

$$\lim_{x\to 0}\frac{\sin(x)}{x} = \lim_{x\to 0}\frac{\cos(x)}{1} = \frac{1}{1} = 1$$

4.10 Taylor Series: Approximating Functions

Any smooth function can be approximated by a polynomial!

Definition 4.3: Taylor Series

The Taylor series of f around $x = a$ is:

$$f(x) = \sum_{n=0}^{\infty}\frac{f^{(n)}(a)}{n!}(x-a)^n = f(a) + f'(a)(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \cdots$$

When $a = 0$, it's called a **Maclaurin series**.

Example

Taylor series of e^x around $x = 0$:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots = \sum_{n=0}^{\infty}\frac{x^n}{n!}$$

Example

Taylor series of $\sin(x)$:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$

Connection to LLMs

Taylor series are everywhere in ML:

- **Activation approximations**: Computing GELU, softmax efficiently
- **Optimization**: Newton's method uses second-order Taylor approximation
- **Analysis**: Understanding behavior of loss landscapes
- **Numerical methods**: Computing transcendental functions

4.11 Practice Problems

4.11.1 Problem 1: Basic Derivatives

Find derivatives of:

(a) $f(x) = 5x^3 - 2x^2 + 7x - 3$

(b) $g(x) = \frac{x^2+1}{x-1}$

(c) $h(x) = e^{x^2+3x}$

(d) $k(x) = \ln(x^3 + 2x)$

4.11.2 Problem 2: Chain Rule Practice

Differentiate:

(a) $f(x) = (3x^2 + 2x + 1)^7$

(b) $g(x) = e^{\sin(x)}$

(c) $h(x) = \sin(e^x)$

(d) $k(x) = \ln(\cos(x^2))$

4.11.3 Problem 3: Activation Function Derivative

Prove that the derivative of sigmoid is $\sigma'(x) = \sigma(x)(1 - \sigma(x))$.

4.11.4 Problem 4: Optimization

Find the minimum of $f(x) = x^3 - 6x^2 + 9x + 1$ and verify it's a minimum using the second derivative test.

4.11.5 Problem 5: Neural Network Connection

A simple neural network computes: $y = \sigma(w_2 \cdot \sigma(w_1 \cdot x))$ where σ is sigmoid.

Use the chain rule to find $\frac{dy}{dw_1}$ (this is what backpropagation computes!).

4.12 Key Takeaways

- **Derivatives** measure rates of change—the foundation of optimization
- **The chain rule** is backpropagation in disguise
- **Critical points** (where $f' = 0$) are candidates for minima/maxima
- **Second derivatives** tell us about curvature
- **Taylor series** approximate functions as polynomials
- Every activation function has a derivative that determines how gradients flow
- Understanding single-variable calculus is the first step to understanding optimization in neural networks

Connection to LLMs

You now understand the calculus that powers gradient descent! When training an LLM, we compute millions of derivatives using these rules. In the next chapter, we'll extend these ideas to multiple variables—the setting where neural networks actually live. This is where things get really exciting!

Chapter 5

Multivariate Calculus: Functions of Many Variables

5.1 Introduction: The Real World is High-Dimensional

In single-variable calculus, we dealt with functions like $f(x) = x^2$. But neural networks have millions or billions of parameters! A GPT-3 sized model has 175 billion weights—that's a function $f : \mathbb{R}^{175\text{B}} \to \mathbb{R}$ mapping weights to loss.

Intuition

Multivariate calculus extends our calculus toolkit to functions of many variables. Instead of asking "how does f change when x changes?", we ask "how does f change when we move in any direction in high-dimensional space?"
This is the mathematics that makes training neural networks possible!

Connection to LLMs

Every concept in this chapter directly relates to training LLMs:

- **Partial derivatives** $\to$ gradients with respect to individual weights
- **Gradient vectors** $\to$ the direction of steepest descent
- **The Hessian matrix** $\to$ second-order optimization methods
- **The chain rule** $\to$ backpropagation through layers

5.2 Functions of Multiple Variables

5.2.1 Basic Definitions

A function $f : \mathbb{R}^n \to \mathbb{R}$ takes n inputs and produces one output:

$$z = f(x_1, x_2, \ldots, x_n)$$

Or in vector notation: $z = f(\mathbf{x})$ where $\mathbf{x} \in \mathbb{R}^n$.

Example

Some multivariate functions:

- $f(x, y) = x^2 + y^2$ (a paraboloid)
- $g(x, y) = \sin(x)\cos(y)$
- $h(x, y, z) = x^2 + 2xy + y^2 + z^2$
- $\mathcal{L}(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \mathbf{w}^\top \mathbf{x}_i)^2$ (least squares loss)

5.2.2 Vector-Valued Functions

A function $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$ has multiple outputs:

$$\mathbf{f}(\mathbf{x}) = \begin{bmatrix} f_1(\mathbf{x}) \\ f_2(\mathbf{x}) \\ \vdots \\ f_m(\mathbf{x}) \end{bmatrix}$$

Connection to LLMs

Neural network layers are vector-valued functions! A layer with d_{in} inputs and d_{out} outputs is:

$$\mathbf{h} : \mathbb{R}^{d_{\text{in}}} \to \mathbb{R}^{d_{\text{out}}}$$

$$\mathbf{h}(\mathbf{x}) = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b})$$

5.3 Partial Derivatives

5.3.1 Definition

A partial derivative measures how a function changes when we vary just one variable, holding all others constant.

Definition 5.1: Partial Derivative

The partial derivative of f with respect to x_i is:

$$\frac{\partial f}{\partial x_i} = \lim_{h \to 0} \frac{f(x_1, \ldots, x_i + h, \ldots, x_n) - f(x_1, \ldots, x_i, \ldots, x_n)}{h}$$

We treat all other variables as constants and differentiate with respect to x_i only.

Example

For $f(x, y) = x^2y + 3xy^2 + 5$:
Partial derivative with respect to x (treat y as constant):

$$\frac{\partial f}{\partial x} = 2xy + 3y^2$$

Partial derivative with respect to y (treat x as constant):

$$\frac{\partial f}{\partial y} = x^2 + 6xy$$

Intuition

Imagine a mountain. The partial derivative $\frac{\partial f}{\partial x}$ tells you the slope if you walk purely in the x direction (east-west), while $\frac{\partial f}{\partial y}$ tells you the slope in the y direction (north-south).

5.3.2 Computing Partial Derivatives

Use the same rules as single-variable calculus, treating other variables as constants!

Example

For $f(x, y, z) = x^2yz + e^{xy} + \sin(xz)$:

$$\frac{\partial f}{\partial x} = 2xyz + ye^{xy} + z\cos(xz)$$
$$\frac{\partial f}{\partial y} = x^2z + xe^{xy}$$
$$\frac{\partial f}{\partial z} = x^2y + x\cos(xz)$$

5.4 The Gradient: Vector of All Partial Derivatives

5.4.1 Definition

The **gradient** collects all partial derivatives into a vector.

Definition 5.2: Gradient

For $f : \mathbb{R}^n \to \mathbb{R}$, the gradient is:

$$\nabla f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{bmatrix} \in \mathbb{R}^n$$

Alternative notations: $\nabla_{\mathbf{x}} f$, $\frac{\partial f}{\partial \mathbf{x}}$, grad f

Example

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

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

At point $(1, 1)$: $\nabla f(1, 1) = \begin{bmatrix} 4 \\ 4 \end{bmatrix}$

5.4.2 Geometric Interpretation

Theorem 5.1: Gradient is Steepest Ascent

The gradient $\nabla f(\mathbf{x})$ points in the direction of steepest increase of f at $\mathbf{x}$.
Its magnitude $\|\nabla f(\mathbf{x})\|$ is the rate of increase in that direction.

Intuition

If you're on a mountain and want to climb fastest, walk in the direction of ∇f. If you want to descend fastest (minimize f), walk in the direction of $-\nabla f$.
This is exactly what gradient descent does!

Connection to LLMs

Gradient Descent Algorithm:

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}^{(t)})$$

We update weights by moving in the direction opposite to the gradient (steepest descent) with step size η (learning rate).
This is how ALL neural networks learn!

5.5 Directional Derivatives

5.5.1 Definition

The **directional derivative** measures the rate of change in any direction $\mathbf{v}$.

Definition 5.3: Directional Derivative

The directional derivative of f at $\mathbf{x}$ in direction $\mathbf{v}$ is:

$$D_{\mathbf{v}} f(\mathbf{x}) = \lim_{h \to 0} \frac{f(\mathbf{x} + h\mathbf{v}) - f(\mathbf{x})}{h}$$

If $\mathbf{v}$ is a unit vector ($\|\mathbf{v}\| = 1$):

$$D_{\mathbf{v}} f(\mathbf{x}) = \nabla f(\mathbf{x}) \cdot \mathbf{v}$$

Intuition

The directional derivative is the dot product of the gradient with the direction. This means:

- Maximum when $\mathbf{v}$ points in direction of ∇f (steepest ascent)
- Minimum when $\mathbf{v}$ points opposite to ∇f (steepest descent)
- Zero when $\mathbf{v}$ is perpendicular to ∇f (level curve)

Example

For $f(x, y) = x^2 + y^2$, we have $\nabla f = \begin{bmatrix} 2x \\ 2y \end{bmatrix}$.

At $(1, 0)$, the gradient is $\begin{bmatrix} 2 \\ 0 \end{bmatrix}$.

Directional derivative in direction $\mathbf{v} = \begin{bmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \end{bmatrix}$:

$$D_{\mathbf{v}} f(1,0) = \begin{bmatrix} 2 \\ 0 \end{bmatrix} \cdot \begin{bmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \end{bmatrix} = \frac{2}{\sqrt{2}} = \sqrt{2}$$

5.6 The Chain Rule for Multivariable Functions

This is the heart of backpropagation!

5.6.1 Chain Rule: Scalar Output

If $z = f(\mathbf{y})$ and $\mathbf{y} = \mathbf{g}(\mathbf{x})$, then:

Theorem 5.2: Multivariate Chain Rule

$$\frac{\partial z}{\partial x_i} = \sum_{j=1}^{m} \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i}$$

In matrix form:

$$\frac{\partial z}{\partial \mathbf{x}} = \left(\frac{\partial \mathbf{y}}{\partial \mathbf{x}} \right)^\top \frac{\partial z}{\partial \mathbf{y}}$$

Example

Let $z = x^2 + y^2$ where $x = r\cos(\theta)$ and $y = r\sin(\theta)$.
Find $\frac{\partial z}{\partial r}$:

$$\frac{\partial z}{\partial r} = \frac{\partial z}{\partial x}\frac{\partial x}{\partial r} + \frac{\partial z}{\partial y}\frac{\partial y}{\partial r}$$
$$= (2x)(\cos\theta) + (2y)(\sin\theta)$$
$$= 2r\cos^2\theta + 2r\sin^2\theta = 2r$$

Connection to LLMs

This is backpropagation!
In a neural network: Input $\mathbf{x} \to$ Hidden $\mathbf{h} \to$ Output $y \to$ Loss $\mathcal{L}$
To compute $\frac{\partial \mathcal{L}}{\partial \mathbf{x}}$:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{x}} = \left(\frac{\partial \mathbf{h}}{\partial \mathbf{x}} \right)^\top \left(\frac{\partial y}{\partial \mathbf{h}} \right)^\top \frac{\partial \mathcal{L}}{\partial y}$$

We compute gradients backwards through the network, one layer at a time!

5.7 The Jacobian Matrix

5.7.1 Definition

For vector-valued functions, we need a matrix of derivatives.

Definition 5.4: Jacobian Matrix

For $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$, the Jacobian is:

$$\mathbf{J}_{\mathbf{f}}(\mathbf{x}) = \frac{\partial \mathbf{f}}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix} \in \mathbb{R}^{m \times n}$$

Row i is the gradient of f_i: $\nabla f_i^\top$

Intuition

The Jacobian tells you how each output changes with respect to each input. It's the multivariable generalization of the derivative!

Example

For $\mathbf{f}(x, y) = \begin{bmatrix} x^2 + y \\ xy \\ x + y^2 \end{bmatrix}$:

$$\mathbf{J} = \begin{bmatrix} 2x & 1 \\ y & x \\ 1 & 2y \end{bmatrix}$$

5.7.2 Chain Rule with Jacobians

If $\mathbf{z} = \mathbf{f}(\mathbf{y})$ and $\mathbf{y} = \mathbf{g}(\mathbf{x})$, then:

$$\frac{\partial \mathbf{z}}{\partial \mathbf{x}} = \frac{\partial \mathbf{z}}{\partial \mathbf{y}} \frac{\partial \mathbf{y}}{\partial \mathbf{x}}$$

That is: $\mathbf{J}_{\mathbf{f} \circ \mathbf{g}} = \mathbf{J}_{\mathbf{f}} \cdot \mathbf{J}_{\mathbf{g}}$

Connection to LLMs

In a deep neural network with L layers:

$$\mathbf{x} \xrightarrow{\mathbf{f}_1} \mathbf{h}_1 \xrightarrow{\mathbf{f}_2} \mathbf{h}_2 \xrightarrow{\mathbf{f}_3} \cdots \xrightarrow{\mathbf{f}_L} \mathbf{y}$$

The overall Jacobian is:

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \mathbf{J}_{f_L} \cdot \mathbf{J}_{f_{L-1}} \cdots \mathbf{J}_{f_2} \cdot \mathbf{J}_{f_1}$$

This product can cause vanishing/exploding gradients if Jacobians are too small/large!

5.8 The Hessian Matrix: Second Derivatives

5.8.1 Definition

The **Hessian** is the matrix of all second partial derivatives.

Definition 5.5: Hessian Matrix

For $f : \mathbb{R}^n \to \mathbb{R}$, the Hessian is:

$$\mathbf{H}_f(\mathbf{x}) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

Element (i, j) is: $H_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j}$

Theorem 5.3: Clairaut's Theorem

If second partial derivatives are continuous, then:

$$\frac{\partial^2 f}{\partial x_i \partial x_j} = \frac{\partial^2 f}{\partial x_j \partial x_i}$$

So the Hessian is symmetric!

Example

For $f(x, y) = x^3 + xy^2 + y^3$:

$$\frac{\partial f}{\partial x} = 3x^2 + y^2, \quad \frac{\partial f}{\partial y} = 2xy + 3y^2$$
$$\frac{\partial^2 f}{\partial x^2} = 6x, \quad \frac{\partial^2 f}{\partial y^2} = 2x + 6y$$
$$\frac{\partial^2 f}{\partial x \partial y} = 2y$$

$$\mathbf{H} = \begin{bmatrix} 6x & 2y \\ 2y & 2x + 6y \end{bmatrix}$$

5.8.2 Interpreting the Hessian: Curvature

The Hessian measures curvature in all directions!

Theorem 5.4: Second-Order Approximation

Near $\mathbf{x}_0$, we can approximate:

$$f(\mathbf{x}) \approx f(\mathbf{x}_0) + \nabla f(\mathbf{x}_0)^\top (\mathbf{x} - \mathbf{x}_0) + \frac{1}{2}(\mathbf{x} - \mathbf{x}_0)^\top \mathbf{H}(\mathbf{x}_0)(\mathbf{x} - \mathbf{x}_0)$$

This is the second-order Taylor expansion!

Intuition

- First-order term: ∇f tells us the slope (linear approximation)
- Second-order term: $\mathbf{H}$ tells us the curvature (quadratic approximation)

Think of the Hessian as describing the "bowl shape" around a point.

5.8.3 Optimality Conditions

Theorem 5.5: Second-Order Optimality Conditions

If $\nabla f(\mathbf{x}^*) = \mathbf{0}$ (critical point), then:

- If $\mathbf{H}(\mathbf{x}^*)$ is positive definite: local minimum
- If $\mathbf{H}(\mathbf{x}^*)$ is negative definite: local maximum
- If $\mathbf{H}(\mathbf{x}^*)$ has both positive and negative eigenvalues: saddle point
- If $\mathbf{H}(\mathbf{x}^*)$ is positive semi-definite: test inconclusive

Connection to LLMs

In neural network optimization:

- **Saddle points** are much more common than local minima in high dimensions!
- **Newton's method** uses the Hessian: $\mathbf{x}^{(t+1)} = \mathbf{x}^{(t)} - \mathbf{H}^{-1}\nabla f$
- **Second-order methods** (L-BFGS, natural gradient) approximate the Hessian
- Computing full Hessian is $O(n^2)$ memory—impractical for large models!

5.9 Constrained Optimization: Lagrange Multipliers

5.9.1 The Problem

Optimize $f(\mathbf{x})$ subject to constraint $g(\mathbf{x}) = 0$.

Example

Maximize $f(x, y) = xy$ subject to $x + y = 10$.
Or: Minimize $f(\mathbf{x}) = \|\mathbf{x} - \mathbf{x}_0\|^2$ subject to $\mathbf{a}^\top \mathbf{x} = b$.

5.9.2 Lagrange Multipliers

Theorem 5.6: Method of Lagrange Multipliers

To optimize $f(\mathbf{x})$ subject to $g(\mathbf{x}) = 0$, define the Lagrangian:

$$\mathcal{L}(\mathbf{x}, \lambda) = f(\mathbf{x}) - \lambda g(\mathbf{x})$$

At an optimum: $\nabla_{\mathbf{x}}\mathcal{L} = \mathbf{0}$ and $\frac{\partial \mathcal{L}}{\partial \lambda} = 0$
This gives: $\nabla f(\mathbf{x}^*) = \lambda \nabla g(\mathbf{x}^*)$

Intuition

At the optimum, the gradients of f and g must be parallel! You can't improve f while staying on the constraint surface $g = 0$.

Example

Maximize $f(x, y) = xy$ subject to $x + y = 10$:
Lagrangian: $\mathcal{L}(x, y, \lambda) = xy - \lambda(x + y - 10)$

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial x} &= y - \lambda = 0 \implies y = \lambda \\
\frac{\partial \mathcal{L}}{\partial y} &= x - \lambda = 0 \implies x = \lambda \\
\frac{\partial \mathcal{L}}{\partial \lambda} &= -(x + y - 10) = 0
\end{aligned}$$

From first two: $x = y$. From third: $2x = 10 \implies x = y = 5$.
Maximum value: $f(5, 5) = 25$.

Connection to LLMs

Lagrange multipliers appear in:

- **Support Vector Machines**: Maximizing margin with constraints
- **Constrained neural network training**: Adding regularization as constraints
- **KKT conditions**: Generalization for inequality constraints
- **Dual problems**: Alternative formulations of optimization problems

5.10 Vector Calculus Identities

Some useful identities for neural networks:

5.10.1 Gradient of Common Functions

$$\begin{aligned}
\nabla_{\mathbf{x}}(\mathbf{a}^\top \mathbf{x}) &= \mathbf{a} \\
\nabla_{\mathbf{x}}(\mathbf{x}^\top \mathbf{A}\mathbf{x}) &= (\mathbf{A} + \mathbf{A}^\top)\mathbf{x} \\
\nabla_{\mathbf{x}}(\|\mathbf{x}\|^2) &= 2\mathbf{x} \\
\nabla_{\mathbf{W}}(\mathbf{a}^\top \mathbf{W}\mathbf{b}) &= \mathbf{a}\mathbf{b}^\top \\
\nabla_{\mathbf{W}}(\mathrm{tr}(\mathbf{W}\mathbf{A})) &= \mathbf{A}^\top
\end{aligned}$$

5.10.2 Chain Rule Patterns

$$\begin{aligned}
\nabla_{\mathbf{x}} f(\mathbf{A}\mathbf{x}) &= \mathbf{A}^\top \nabla_{\mathbf{y}} f(\mathbf{y}) \quad \text{where } \mathbf{y} = \mathbf{A}\mathbf{x} \\
\nabla_{\mathbf{x}} f(g(\mathbf{x})) &= \frac{df}{dg} \nabla_{\mathbf{x}} g(\mathbf{x}) \\
\nabla_{\mathbf{W}} f(\mathbf{W}\mathbf{x}) &= \nabla_{\mathbf{y}} f(\mathbf{y}) \cdot \mathbf{x}^\top \quad \text{where } \mathbf{y} = \mathbf{W}\mathbf{x}
\end{aligned}$$

5.11 Practice Problems

5.11.1 Problem 1: Computing Gradients

For $f(x, y, z) = x^2 y + 2xz + yz^2$, compute:

(a) All partial derivatives

(b) The gradient ∇f

(c) The gradient at point $(1, 2, 3)$

5.11.2 Problem 2: Hessian

For $f(x, y) = x^4 + y^4 - 2x^2 - 2y^2 + 4xy$:

(a) Find all critical points

(b) Compute the Hessian

(c) Classify each critical point as min/max/saddle

5.11.3 Problem 3: Chain Rule

A neural network layer computes: $\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b}$, then $\mathbf{a} = \text{ReLU}(\mathbf{z})$, then $L = \|\mathbf{a} - \mathbf{y}\|^2$.

Use the chain rule to derive:

(a) $\frac{\partial L}{\partial \mathbf{a}}$

(b) $\frac{\partial L}{\partial \mathbf{z}}$

(c) $\frac{\partial L}{\partial \mathbf{W}}$

5.11.4 Problem 4: Lagrange Multipliers

Minimize $f(x, y) = x^2 + y^2$ subject to $2x + 3y = 6$.

5.11.5 Problem 5: Jacobian

Compute the Jacobian of $\mathbf{f}(x, y) = \begin{bmatrix} e^x \cos(y) \\ e^x \sin(y) \end{bmatrix}$ at point $(0, \pi/4)$.

5.12 Key Takeaways

- **Partial derivatives** measure change in one variable at a time
- **The gradient** ∇f points in the direction of steepest ascent
- **Gradient descent** uses $-\nabla f$ to minimize functions
- **The chain rule** enables backpropagation through layers
- **The Jacobian** is the derivative of vector-valued functions
- **The Hessian** captures second-order information (curvature)
- **Lagrange multipliers** handle constrained optimization
- Understanding these tools is essential for implementing and debugging neural networks

Connection to LLMs

You now understand the mathematics behind backpropagation and gradient descent! Every time you train a neural network, you're computing gradients using the chain rule, then updating parameters by moving opposite to the gradient. In the next chapter, we'll explore optimization algorithms that make this process faster and more stable. The journey from math to working LLMs continues!

Chapter 6

Optimization: Finding the Best Parameters

6.1 Introduction: The Training Problem

Training a neural network is fundamentally an optimization problem: find the weights $\mathbf{w}^*$ that minimize the loss function $\mathcal{L}(\mathbf{w})$.

Intuition

Imagine you're blindfolded on a mountain and want to reach the valley (minimum). You can feel the slope beneath your feet. Optimization algorithms are strategies for reaching the valley efficiently!

Connection to LLMs

Every LLM is trained using optimization. Understanding these algorithms is understanding how models learn!

6.2 The Optimization Problem

6.2.1 General Form

Definition 6.1: Optimization Problem

$$\mathbf{w}^* = \arg\min_{\mathbf{w}\in\mathbb{R}^d} \mathcal{L}(\mathbf{w})$$

where $\mathcal{L} : \mathbb{R}^d \to \mathbb{R}$ is the loss function.

For supervised learning:

$$\mathcal{L}(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n} \ell(f(\mathbf{x}_i; \mathbf{w}), y_i)$$

6.3 Gradient Descent

6.3.1 The Basic Algorithm

Definition 6.2: Gradient Descent

Initialize $\mathbf{w}^{(0)}$ randomly. Then iterate:

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \nabla \mathcal{L}(\mathbf{w}^{(t)})$$

where $\eta > 0$ is the learning rate.

Intuition

The gradient $\nabla \mathcal{L}$ points uphill. Moving in the direction $-\nabla \mathcal{L}$ takes us downhill!

6.3.2 Convergence Analysis

For smooth, convex functions:

Theorem 6.1: GD Convergence

With appropriate learning rate, gradient descent converges to the global minimum.

6.4 Stochastic Gradient Descent (SGD)

6.4.1 The Motivation

Computing the full gradient requires all n training examples—expensive for large datasets!

Definition 6.3: Stochastic Gradient Descent

At each step, sample one example (x_i, y_i) and update:

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \nabla_{\mathbf{w}} \ell(f(\mathbf{x}_i; \mathbf{w}^{(t)}), y_i)$$

6.4.2 Mini-Batch SGD

In practice, use small batches:

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \nabla_{\mathbf{w}} \ell(f(\mathbf{x}_i; \mathbf{w}^{(t)}), y_i)$$

Typical batch sizes: 32, 64, 128, 256.

6.5 Momentum

6.5.1 The Idea

Add "inertia" to optimization—remember previous gradients!

Definition 6.4: SGD with Momentum

$$\mathbf{v}^{(t+1)} = \beta \mathbf{v}^{(t)} + \nabla \mathcal{L}(\mathbf{w}^{(t)})$$
$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \mathbf{v}^{(t+1)}$$

where $\beta \in [0, 1)$ is the momentum coefficient (typically 0.9).

Intuition

Like a ball rolling downhill—it builds up speed and can roll through small bumps!

6.6 Adaptive Learning Rates

6.6.1 AdaGrad

Adapt learning rate per parameter:

$$\mathbf{w}_i^{(t+1)} = \mathbf{w}_i^{(t)} - \frac{\eta}{\sqrt{G_{ii}^{(t)}} + \epsilon} g_i^{(t)}$$

where $G_{ii}^{(t)} = \sum_{\tau=1}^{t} (g_i^{(\tau)})^2$.

6.6.2 RMSProp

Use exponential moving average:

$$\mathbf{v}^{(t+1)} = \beta \mathbf{v}^{(t)} + (1 - \beta)(\nabla \mathcal{L})^2$$
$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \frac{\eta}{\sqrt{\mathbf{v}^{(t+1)}} + \epsilon} \nabla \mathcal{L}$$

6.7 Adam: The Industry Standard

Definition 6.5: Adam Optimizer

Combine momentum and adaptive learning rates:

$$\begin{aligned}
\mathbf{m}^{(t+1)} &= \beta_1 \mathbf{m}^{(t)} + (1-\beta_1)\nabla\mathcal{L} \\
\mathbf{v}^{(t+1)} &= \beta_2 \mathbf{v}^{(t)} + (1-\beta_2)(\nabla\mathcal{L})^2 \\
\hat{\mathbf{m}} &= \frac{\mathbf{m}^{(t+1)}}{1-\beta_1^t} \\
\hat{\mathbf{v}} &= \frac{\mathbf{v}^{(t+1)}}{1-\beta_2^t} \\
\mathbf{w}^{(t+1)} &= \mathbf{w}^{(t)} - \eta\frac{\hat{\mathbf{m}}}{\sqrt{\hat{\mathbf{v}}}+\epsilon}
\end{aligned}$$

Default: $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$.

Connection to LLMs

Adam is the default optimizer for training most LLMs! It combines the best of momentum and adaptive learning rates.

6.8 Learning Rate Schedules

6.8.1 Step Decay

Reduce learning rate at fixed intervals:

$$\eta_t = \eta_0 \cdot \gamma^{\lfloor t/k \rfloor}$$

6.8.2 Cosine Annealing

$$\eta_t = \eta_{\min} + \frac{1}{2}(\eta_{\max} - \eta_{\min})(1 + \cos(\pi t/T))$$

6.8.3 Warmup

Gradually increase learning rate at the start:

$$\eta_t = \eta_{\max} \cdot \min(1, t/T_{\text{warmup}})$$

Connection to LLMs

Modern LLM training uses warmup + cosine decay. This stabilizes early training and helps convergence!

6.9 Key Takeaways

- **Gradient descent** uses first-order information to minimize loss
- **SGD** makes training tractable for large datasets
- **Momentum** accelerates convergence and escapes local minima
- **Adam** is the most popular optimizer for deep learning
- **Learning rate schedules** improve final performance
- Understanding optimization is key to training LLMs effectively

Connection to LLMs

You now understand how neural networks learn! In the next chapters, we'll explore probability theory and information theory—the foundations for understanding language modeling as a probabilistic task.

Part III

Probability & Statistics

Chapter 7

Probability Theory: Mathematics of Uncertainty

7.1 Introduction: Why Probability in AI?

Neural networks don't give us certainty—they give us probabilities. When an LLM generates text, it's sampling from a probability distribution over possible next words. Understanding probability is understanding how these models think!

Intuition

Probability is the mathematics of uncertainty. In the real world, we rarely know things for certain:

- Will it rain tomorrow? (Uncertain)
- What's the next word in this sentence? (Multiple possibilities)
- Is this email spam? (Probabilistic classification)

Machine learning embraces this uncertainty mathematically.

Connection to LLMs

In LLMs, probability is everywhere:

- **Softmax outputs**: Probability distribution over vocabulary
- **Sampling**: Generating text by sampling from distributions
- **Training**: Maximizing likelihood of training data
- **Uncertainty estimation**: How confident is the model?

7.2 Basic Probability Concepts

7.2.1 Sample Space and Events

Definition 7.1: Sample Space

The **sample space** Ω is the set of all possible outcomes of a random experiment.
An **event** A is a subset of the sample space.

Example

- Rolling a die: $\Omega = \{1, 2, 3, 4, 5, 6\}$
- Flipping a coin: $\Omega = \{\text{H}, \text{T}\}$
- Predicting next word: $\Omega = \{\text{all words in vocabulary}\}$

7.2.2 Probability Axioms

Definition 7.2: Probability Measure

A probability function $P : \mathcal{F} \to [0, 1]$ satisfies:

1. **Non-negativity**: $P(A) \geq 0$ for all events A
2. **Normalization**: $P(\Omega) = 1$
3. **Additivity**: For disjoint events $A_1, A_2, \ldots$:

$$P\left(\bigcup_{i=1}^{\infty} A_i\right) = \sum_{i=1}^{\infty} P(A_i)$$

Intuition

Think of probability as "fraction of the whole":

- $P(A) = 0$: Event never happens
- $P(A) = 1$: Event always happens
- $P(A) = 0.5$: Event happens half the time

7.2.3 Basic Rules

Theorem 7.1: Complement Rule

$$P(A^c) = 1 - P(A)$$

where A^c is the complement of A (all outcomes not in A).

Theorem 7.2: Addition Rule

For any events A and B:

$$P(A \cup B) = P(A) + P(B) - P(A \cap B)$$

Example

Probability of rolling an even number OR a number greater than 4:

$$\begin{gathered} A = \{2, 4, 6\}, \quad P(A) = 1/2 \\ B = \{5, 6\}, \quad P(B) = 1/3 \\ A \cap B = \{6\}, \quad P(A \cap B) = 1/6 \\ P(A \cup B) = 1/2 + 1/3 - 1/6 = 2/3 \end{gathered}$$

7.3 Conditional Probability

7.3.1 Definition

Definition 7.3: Conditional Probability

The probability of A given that B has occurred:

$$P(A|B) = \frac{P(A \cap B)}{P(B)}$$

provided $P(B) > 0$.

Intuition

$P(A|B)$ means: "Now that I know B happened, what's the probability of A?"
It's like zooming in—B becomes our new sample space!

Example

Drawing cards:

- $P(\text{Ace}) = 4/52$
- $P(\text{Ace}|\text{Spade}) = 1/13$ (only 13 spades, 1 is ace)
- $P(\text{Spade}|\text{Ace}) = 1/4$ (4 aces, 1 is spade)

7.3.2 Chain Rule

Theorem 7.3: Chain Rule of Probability

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

More generally:

$$P(A_1 \cap A_2 \cap \cdots \cap A_n) = P(A_1)P(A_2|A_1)P(A_3|A_1 \cap A_2)\cdots$$

Connection to LLMs

This is how language models work!
Probability of a sentence is:

$$\begin{aligned} P(\text{“The cat sat on mat”}) &= P(\text{The}) \cdot P(\text{cat}|\text{The}) \\ &\quad \cdot P(\text{sat}|\text{The cat}) \cdots \end{aligned}$$

LLMs model $P(\text{next word}|\text{previous words})$!

7.4 Independence

7.4.1 Definition

Definition 7.4: Independence

Events A and B are **independent** if:

$$P(A \cap B) = P(A) \cdot P(B)$$

Equivalently: $P(A|B) = P(A)$ (knowing B doesn't change probability of A)

Example

- **Independent**: Two coin flips
- **Dependent**: Drawing cards without replacement
- **Dependent**: "It's raining" and "ground is wet"

7.4.2 Conditional Independence

Definition 7.5: Conditional Independence

A and B are conditionally independent given C if:

$$P(A \cap B|C) = P(A|C) \cdot P(B|C)$$

Connection to LLMs

Many machine learning models assume conditional independence to simplify computation. For example, Naive Bayes assumes features are independent given the class label.

7.5 Bayes' Theorem

7.5.1 The Most Important Theorem in AI

Theorem 7.4: Bayes' Theorem

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

Or equivalently:

$$P(A|B) = \frac{P(B|A)P(A)}{\sum_i P(B|A_i)P(A_i)}$$

where A_i partitions the sample space.

Intuition

Bayes' theorem lets us flip conditional probabilities! We can go from $P(B|A)$ to $P(A|B)$.

In machine learning terms:

$$P(\text{hypothesis}|\text{data}) = \frac{P(\text{data}|\text{hypothesis})P(\text{hypothesis})}{P(\text{data})}$$

- $P(\text{hypothesis}|\text{data})$: **Posterior** (what we want)
- $P(\text{data}|\text{hypothesis})$: **Likelihood** (how well hypothesis explains data)
- $P(\text{hypothesis})$: **Prior** (belief before seeing data)
- $P(\text{data})$: **Evidence** (normalizing constant)

Example

Medical diagnosis:

- $P(\text{disease}) = 0.01$ (1% of population has disease)
- $P(\text{positive test}|\text{disease}) = 0.95$ (95% sensitivity)
- $P(\text{positive test}|\text{no disease}) = 0.05$ (5% false positive)

You test positive. What's $P(\text{disease}|\text{positive test})$?

$$\begin{aligned}
P(D|+) &= \frac{P(+|D)P(D)}{P(+)} \\
&= \frac{P(+|D)P(D)}{P(+|D)P(D) + P(+|D^c)P(D^c)} \\
&= \frac{0.95 \times 0.01}{0.95 \times 0.01 + 0.05 \times 0.99} \\
&= \frac{0.0095}{0.0095 + 0.0495} = 0.161
\end{aligned}$$

Only 16.1%! Even with a positive test, disease is unlikely because it's rare.

7.6 Random Variables

7.6.1 Definition

Definition 7.6: Random Variable

A **random variable** is a function $X : \Omega \to \mathbb{R}$ that assigns a numerical value to each outcome.

Example

- Roll a die: $X =$ number shown
- Flip 3 coins: $X =$ number of heads
- Sample a word: $X =$ word embedding vector

7.6.2 Discrete Random Variables

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

$$p_X(x) = P(X = x)$$

Properties:

- $p_X(x) \geq 0$ for all x
- $\sum_x p_X(x) = 1$

Example

Fair die: $p_X(k) = 1/6$ for $k \in \{1, 2, 3, 4, 5, 6\}$

7.6.3 Continuous Random Variables

For continuous X, we have a **probability density function** (PDF):

$$f_X(x)$$

Properties:

- $f_X(x) \geq 0$ for all x
- $\int_{-\infty}^{\infty} f_X(x)dx = 1$
- $P(a \leq X \leq b) = \int_a^b f_X(x)dx$

Common Pitfall

For continuous X: $P(X = x) = 0$ for any specific x! Only intervals have non-zero probability.

7.7 Expectation and Variance

7.7.1 Expected Value

Definition 7.7: Expectation

The **expected value** (or mean) is:
Discrete: $\mathbb{E}[X] = \sum_x x \cdot p_X(x)$
Continuous: $\mathbb{E}[X] = \int_{-\infty}^{\infty} x \cdot f_X(x)dx$

Intuition

The expected value is the "average" or "center" of the distribution. It's the long-run average if you repeat the experiment many times.

Example

Rolling a die:

$$\mathbb{E}[X] = 1 \cdot \frac{1}{6} + 2 \cdot \frac{1}{6} + \cdots + 6 \cdot \frac{1}{6} = \frac{21}{6} = 3.5$$

7.7.2 Properties of Expectation

Theorem 7.5: Linearity of Expectation

- $\mathbb{E}[aX + b] = a\mathbb{E}[X] + b$
- $\mathbb{E}[X + Y] = \mathbb{E}[X] + \mathbb{E}[Y]$ (even if not independent!)

7.7.3 Variance

Definition 7.8: Variance

The **variance** measures spread:

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

The **standard deviation** is: $\sigma_X = \sqrt{\mathrm{Var}\left(X\right)}$

Intuition

Variance tells us how much values typically deviate from the mean. High variance = more spread out.

Theorem 7.6: Variance Properties

- $\mathrm{Var}\left(aX + b\right) = a^2\mathrm{Var}\left(X\right)$ (constants shift and scale)
- If X and Y are independent: $\mathrm{Var}\left(X + Y\right) = \mathrm{Var}\left(X\right) + \mathrm{Var}\left(Y\right)$

7.8 Common Probability Distributions

7.8.1 Bernoulli Distribution

Single trial with two outcomes (success/failure):

$$X \sim \mathrm{Bernoulli}(p)$$

$$P(X = 1) = p, \quad P(X = 0) = 1 - p$$
$$\mathbb{E}\left[X\right] = p, \quad \mathrm{Var}\left(X\right) = p(1 - p)$$

Connection to LLMs

Used for binary classification! Predicting probability of class 1.

7.8.2 Categorical Distribution

Generalization to k outcomes:

$$X \sim \mathrm{Categorical}(p_1, \ldots, p_k)$$

$$P(X = i) = p_i, \quad \sum_{i=1}^{k} p_i = 1$$

Connection to LLMs

This is what LLMs output! The softmax layer produces a categorical distribution over the vocabulary:

$$P(\text{next word} = w_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

7.8.3 Gaussian (Normal) Distribution

The most important continuous distribution:

$$X \sim \mathcal{N}(\mu, \sigma^2)$$

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

Properties:

- $\mathbb{E}[X] = \mu$
- $\text{Var}(X) = \sigma^2$
- Bell-shaped, symmetric around μ

Connection to LLMs

Gaussians are everywhere in ML:

- Weight initialization
- Noise models
- Variational autoencoders
- Gaussian processes

7.9 Joint and Marginal Distributions

7.9.1 Joint Distribution

For multiple random variables X and Y:

$$p_{X,Y}(x, y) = P(X = x, Y = y)$$

7.9.2 Marginal Distribution

To get distribution of X alone, sum over Y:

$$p_X(x) = \sum_y p_{X,Y}(x, y)$$

Intuition

Marginalization "integrates out" variables we don't care about.

7.9.3 Conditional Distribution

$$p_{X|Y}(x|y) = \frac{p_{X,Y}(x, y)}{p_Y(y)}$$

7.10 Maximum Likelihood Estimation

7.10.1 The Principle

Given data $\mathcal{D} = \{x_1, \dots, x_n\}$, find parameters θ that maximize the likelihood:

$$\hat{\theta} = \arg\max_\theta P(\mathcal{D}|\theta) = \arg\max_\theta \prod_{i=1}^{n} P(x_i|\theta)$$

Usually we maximize log-likelihood:

$$\hat{\theta} = \arg\max_\theta \sum_{i=1}^{n} \log P(x_i|\theta)$$

Connection to LLMs

This is how neural networks are trained!
The loss function is negative log-likelihood:

$$\mathcal{L}(\theta) = -\sum_{i=1}^{n} \log P(y_i|x_i, \theta)$$

Minimizing loss = maximizing likelihood = making training data most probable!

7.11 Key Takeaways

- **Probability** quantifies uncertainty mathematically
- **Conditional probability** updates beliefs given information
- **Bayes' theorem** is fundamental to inference and learning
- **Random variables** assign numerical values to random outcomes
- **Expectation** is the average, **variance** measures spread
- **Common distributions** (Bernoulli, Categorical, Gaussian) model different scenarios
- **Maximum likelihood** is the principle behind most ML training
- LLMs are fundamentally probabilistic models!

Connection to LLMs

Probability theory is the language of machine learning. Every prediction an LLM makes is a probability distribution. Every training step maximizes likelihood. Understanding probability means understanding what these models are really doing under the hood. In the next chapters, we'll build on this foundation to understand information theory, entropy, and why cross-entropy loss works so well!

Chapter 8

Probability Distributions: The Language of Uncertainty

8.1 Introduction: Modeling Randomness

In machine learning, uncertainty is everywhere. Will the model correctly predict the next word? How confident should we be? Probability distributions answer these questions.

Intuition

A probability distribution describes how likely different outcomes are. In LLMs:

- The output is a probability distribution over vocabulary
- Dropout introduces randomness during training
- We sample from distributions to generate diverse text
- Uncertainty quantification helps us know when to trust the model

Connection to LLMs

Understanding distributions is essential for:

- Cross-entropy loss (comparing distributions)
- Softmax outputs (categorical distributions)
- Sampling strategies (temperature, top-k, nucleus sampling)
- Bayesian approaches to neural networks

8.2 Discrete Distributions

8.2.1 Bernoulli Distribution

The simplest distribution—a single coin flip.

Definition 8.1: Bernoulli Distribution

A random variable $X \sim \text{Bernoulli}(p)$ takes value 1 with probability p and 0 with probability $1 - p$:

$$P(X = 1) = p, \quad P(X = 0) = 1 - p$$

PMF: $P(X = x) = p^x(1 - p)^{1-x}$ for $x \in \{0, 1\}$
Mean: $\mathbb{E}[X] = p$
Variance: $\text{Var}(X) = p(1 - p)$

Connection to LLMs

Bernoulli distribution models:

- Binary classification outputs
- Dropout (each neuron kept with probability p)
- Binary features

8.2.2 Categorical Distribution

Generalizes Bernoulli to multiple outcomes.

Definition 8.2: Categorical Distribution

A random variable $X \sim \text{Categorical}(\mathbf{p})$ where $\mathbf{p} = [p_1, \ldots, p_K]$ with $\sum_i p_i = 1$:

$$P(X = i) = p_i$$

Connection to LLMs

This is what LLMs output!
After the softmax layer, we get a categorical distribution over the vocabulary:

$$P(\text{next word} = w_i | \text{context}) = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

This is exactly a categorical distribution!

8.2.3 Multinomial Distribution

Multiple independent categorical trials.

Definition 8.3: Multinomial Distribution

For n trials with K possible outcomes, counts $(X_1, \ldots, X_K) \sim \text{Multinomial}(n, \mathbf{p})$:

$$P(X_1 = x_1, \ldots, X_K = x_K) = \frac{n!}{x_1! \cdots x_K!} p_1^{x_1} \cdots p_K^{x_K}$$

where $\sum_i x_i = n$.

8.3 Continuous Distributions

8.3.1 Uniform Distribution

All values in an interval equally likely.

Definition 8.4: Uniform Distribution

$X \sim \text{Uniform}(a, b)$ has PDF:

$$f(x) = \begin{cases} \frac{1}{b-a} & a \leq x \leq b \\ 0 & \text{otherwise} \end{cases}$$

Mean: $\mathbb{E}[X] = \frac{a+b}{2}$
Variance: $\text{Var}(X) = \frac{(b-a)^2}{12}$

Connection to LLMs

Used for:

- Random initialization (uniform on $[-r, r]$)
- Sampling in Monte Carlo methods
- Data augmentation

8.3.2 Normal (Gaussian) Distribution

The most important distribution in all of statistics!

Definition 8.5: Normal Distribution

$X \sim \mathcal{N}(\mu, \sigma^2)$ has PDF:

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

Mean: $\mathbb{E}[X] = \mu$
Variance: $\text{Var}(X) = \sigma^2$

Theorem 8.1: Central Limit Theorem

The sum of many independent random variables (under mild conditions) approaches a normal distribution, regardless of their individual distributions!

$$\frac{\sum_{i=1}^{n} X_i - n\mu}{\sigma\sqrt{n}} \xrightarrow{d} \mathcal{N}(0,1)$$

Intuition

This is why the normal distribution appears everywhere! Any process that involves adding many small effects will be approximately normal.

8.3.3 Multivariate Normal Distribution

The high-dimensional Gaussian.

Definition 8.6: Multivariate Normal

$\mathbf{X} \sim \mathcal{N}(\mu, \mathbf{\Sigma})$ where $\mu \in \mathbb{R}^d$ and $\mathbf{\Sigma} \in \mathbb{R}^{d\times d}$ is positive definite:

$$f(\mathbf{x}) = \frac{1}{(2\pi)^{d/2}|\mathbf{\Sigma}|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x}-\mu)^\top \mathbf{\Sigma}^{-1}(\mathbf{x}-\mu)\right)$$

Connection to LLMs

Multivariate Gaussians in deep learning:

- **Weight initialization**: Xavier/He initialization uses Gaussians
- **Gaussian Processes**: Prior over functions

- **Variational inference**: Approximate posteriors
- **Noise modeling**: Additive Gaussian noise

8.3.4 Exponential Distribution

Models waiting times.

Definition 8.7: Exponential Distribution

$X \sim \text{Exp}(\lambda)$ has PDF:

$$f(x) = \lambda e^{-\lambda x}, \quad x \geq 0$$

Mean: $\mathbb{E}[X] = \frac{1}{\lambda}$
Variance: $\text{Var}(X) = \frac{1}{\lambda^2}$

8.4 The Softmax Distribution

8.4.1 From Logits to Probabilities

The softmax function converts arbitrary scores (logits) to a probability distribution.

Definition 8.8: Softmax Function

Given logits $\mathbf{z} = [z_1, \ldots, z_K]$:

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

Properties:

- All outputs positive: $\text{softmax}(\mathbf{z})_i > 0$
- Sum to 1: $\sum_i \text{softmax}(\mathbf{z})_i = 1$
- Preserves order: if $z_i > z_j$ then $\text{softmax}(\mathbf{z})_i > \text{softmax}(\mathbf{z})_j$
- Translation invariant: $\text{softmax}(\mathbf{z} + c) = \text{softmax}(\mathbf{z})$

8.4.2 Temperature Scaling

We can control the "sharpness" of the distribution:

Definition 8.9: Softmax with Temperature

$$\text{softmax}_T(\mathbf{z})_i = \frac{\exp(z_i/T)}{\sum_{j=1}^{K} \exp(z_j/T)}$$

- $T \to 0$: Distribution becomes one-hot (argmax)
- $T = 1$: Standard softmax
- $T \to \infty$: Distribution becomes uniform

Connection to LLMs

Temperature is used in LLM text generation:

- **Low temperature** ($T = 0.1$): Deterministic, focused, repetitive
- **Medium temperature** ($T = 0.7$): Balanced, natural
- **High temperature** ($T = 1.5$): Creative, diverse, sometimes incoherent

8.5 Sampling Strategies for LLMs

8.5.1 Greedy Decoding

Always pick the most likely token:

$$w_t = \arg\max_{w} P(w|w_{1:t-1})$$

Simple but often produces repetitive text!

8.5.2 Top-k Sampling

Sample from the k most likely tokens:

1. Sort tokens by probability
2. Keep only top k
3. Renormalize probabilities
4. Sample from this restricted distribution

8.5.3 Nucleus (Top-p) Sampling

Sample from the smallest set of tokens whose cumulative probability exceeds p:

1. Sort tokens by probability (descending)
2. Find smallest set S where $\sum_{w \in S} P(w) \geq p$
3. Sample from S (renormalized)

Connection to LLMs

Modern LLMs typically use nucleus sampling with $p = 0.9$ or $p = 0.95$. This balances quality and diversity better than top-k!

8.6 Kullback-Leibler (KL) Divergence

8.6.1 Measuring Distance Between Distributions

How different are two probability distributions?

Definition 8.10: KL Divergence

For discrete distributions P and Q:

$$D_{KL}(P \| Q) = \sum_i P(i) \log \frac{P(i)}{Q(i)}$$

For continuous distributions:

$$D_{KL}(P \| Q) = \int p(x) \log \frac{p(x)}{q(x)} dx$$

Properties:

- Non-negative: $D_{KL}(P \| Q) \geq 0$
- Zero iff identical: $D_{KL}(P \| Q) = 0$ iff $P = Q$
- NOT symmetric: $D_{KL}(P \| Q) \neq D_{KL}(Q \| P)$
- NOT a metric (doesn't satisfy triangle inequality)

Connection to LLMs

KL divergence in machine learning:

- **Cross-entropy loss**: Minimizing cross-entropy is equivalent to minimizing KL divergence
- **Variational inference**: Approximate complex posteriors with simple distributions
- **Knowledge distillation**: Match student and teacher distributions
- **RLHF**: Constrain how much the policy can change

8.7 Cross-Entropy

8.7.1 Definition and Connection to KL Divergence

Definition 8.11: Cross-Entropy

For distributions P (true) and Q (predicted):

$$H(P, Q) = -\sum_i P(i) \log Q(i)$$

Relationship to KL divergence:

$$D_{KL}(P\|Q) = H(P, Q) - H(P)$$

Since $H(P)$ (entropy of true distribution) is constant during training:

$$\min_Q D_{KL}(P\|Q) \equiv \min_Q H(P, Q)$$

Connection to LLMs

This is why we minimize cross-entropy loss in neural networks!
For classification with true label y and predicted probabilities $\hat{\mathbf{y}}$:

$$\mathcal{L} = -\log \hat{y}_{\text{true class}}$$

This is cross-entropy between one-hot distribution (true label) and predicted distribution!

8.8 Maximum Likelihood Estimation

8.8.1 The Principle

Given data, find the distribution parameters that make the data most likely.

Definition 8.12: Maximum Likelihood Estimation (MLE)

For data $\mathcal{D} = \{x_1, \ldots, x_n\}$ and model with parameters θ:

$$\hat{\theta}_{MLE} = \arg\max_\theta P(\mathcal{D}|\theta) = \arg\max_\theta \prod_{i=1}^{n} P(x_i|\theta)$$

Usually we maximize log-likelihood instead:

$$\hat{\theta}_{MLE} = \arg\max_\theta \sum_{i=1}^{n} \log P(x_i|\theta)$$

Connection to LLMs

Training neural networks is maximum likelihood estimation! For language modeling, we maximize:

$$\sum_{t=1}^{T} \log P(w_t|w_{1:t-1};\theta)$$

This is exactly MLE with the categorical distribution parameterized by the neural network!

8.9 Practice Problems

8.9.1 Problem 1: Softmax Computation

Given logits $\mathbf{z} = [2.0, 1.0, 0.1]$:

(a) Compute softmax probabilities

(b) Compute softmax with temperature $T = 0.5$

(c) Compute softmax with temperature $T = 2.0$

8.9.2 Problem 2: KL Divergence

Compute $D_{KL}(P\|Q)$ where:

$$P = [0.5, 0.3, 0.2], \quad Q = [0.4, 0.4, 0.2]$$

8.9.3 Problem 3: Sampling

Given probability distribution $[0.5, 0.3, 0.1, 0.05, 0.05]$:

(a) What tokens are selected by top-k sampling with $k = 3$?

(b) What tokens are selected by nucleus sampling with $p = 0.9$?

8.9.4 Problem 4: Cross-Entropy

For 3-class classification, true label is class 2. Predicted probabilities are $[0.2, 0.7, 0.1]$.

Compute the cross-entropy loss.

8.10 Key Takeaways

- **Categorical distribution** is what LLMs output after softmax
- **Normal distribution** appears everywhere due to Central Limit Theorem
- **Softmax** converts logits to valid probability distributions
- **Temperature** controls the sharpness of distributions
- **Top-k and nucleus sampling** balance quality and diversity in generation
- **KL divergence** measures how different two distributions are
- **Cross-entropy loss** is equivalent to minimizing KL divergence
- **Maximum likelihood** is the principle behind neural network training

Connection to LLMs

Probability distributions are the language LLMs speak! Understanding softmax, cross-entropy, and sampling strategies is essential for both training and inference. Every time you generate text from ChatGPT, these concepts are at work—determining which words to sample and with what probability!

Chapter 9

Statistical Inference and Estimation

9.1 Introduction: Learning from Data

Statistics is the science of learning from data in the presence of uncertainty. In machine learning, we observe training data and want to infer patterns that generalize to unseen data. This chapter covers the mathematical foundations of statistical inference that underpin all of machine learning.

Intuition

Think of training an LLM: We observe billions of text samples (data) and want to learn the underlying patterns of language (the true distribution). Statistical inference gives us the tools to do this rigorously, quantifying our uncertainty and making optimal decisions.

Connection to LLMs

Every aspect of training LLMs involves statistical inference:

- **Parameter estimation**: Finding the best weights
- **Confidence intervals**: Uncertainty in predictions
- **Hypothesis testing**: Does this architectural change help?
- **Bayesian methods**: Incorporating prior knowledge

9.2 Point Estimation

9.2.1 What is an Estimator?

An **estimator** is a function of the data that produces an estimate of an unknown parameter.

Definition 9.1: Estimator

Given data $\mathbf{x} = (x_1, \ldots, x_n)$ drawn from distribution P_θ, an estimator $\hat{\theta}$ is a function:

$$\hat{\theta} = \hat{\theta}(\mathbf{x})$$

that estimates the true parameter θ.

Example

Common estimators:

- Sample mean: $\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$ estimates $\mu = \mathbb{E}[X]$
- Sample variance: $s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2$ estimates $\sigma^2 = \text{Var}(X)$
- Maximum likelihood estimator: $\hat{\theta}_{MLE}$ (discussed below)

9.2.2 Properties of Estimators

Bias

Definition 9.2: Bias

The bias of an estimator $\hat{\theta}$ is:

$$\text{Bias}(\hat{\theta}) = \mathbb{E}[\hat{\theta}] - \theta$$

An estimator is **unbiased** if $\mathbb{E}[\hat{\theta}] = \theta$.

Intuition

An unbiased estimator is correct on average. If you repeatedly estimate from different datasets, the average of your estimates equals the true value.

Variance

The variance measures how much estimates vary across different datasets:

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

Intuition

Low variance means the estimator is consistent—different datasets give similar estimates.

Mean Squared Error (MSE)

Definition 9.3: Mean Squared Error

$$\text{MSE}(\hat{\theta}) = \mathbb{E}[(\hat{\theta} - \theta)^2] = \text{Bias}(\hat{\theta})^2 + \text{Var}(\hat{\theta})$$

Intuition

The bias-variance decomposition! MSE captures both systematic error (bias) and random error (variance).
This is the famous bias-variance tradeoff in machine learning!

9.3 Maximum Likelihood Estimation (MLE)

MLE is one of the most important concepts in all of statistics and machine learning.

9.3.1 The Likelihood Function

Definition 9.4: Likelihood Function

Given data $\mathbf{x} = (x_1, \ldots, x_n)$ and parametric model P_θ, the likelihood is:

$$\mathcal{L}(\theta|\mathbf{x}) = P(\mathbf{x}|\theta) = \prod_{i=1}^{n} p(x_i|\theta)$$

(assuming i.i.d. data)

Intuition

The likelihood asks: "Given this parameter value θ, how likely was it to observe this data?"
Note: We're treating θ as the variable, not $\mathbf{x}$!

9.3.2 Maximum Likelihood Estimator

Definition 9.5: Maximum Likelihood Estimator (MLE)

$$\hat{\theta}_{MLE} = \arg\max_{\theta} \mathcal{L}(\theta|\mathbf{x}) = \arg\max_{\theta} \log \mathcal{L}(\theta|\mathbf{x})$$

We usually maximize the log-likelihood (it's easier and equivalent):

$$\ell(\theta) = \log \mathcal{L}(\theta) = \sum_{i=1}^{n} \log p(x_i|\theta)$$

Example

Data: $x_1, \ldots, x_n \sim \mathcal{N}(\mu, \sigma^2)$ with known σ^2, unknown μ.
Likelihood:

$$\mathcal{L}(\mu) = \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x_i-\mu)^2}{2\sigma^2}\right)$$

Log-likelihood:

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

Maximize by setting derivative to zero:

$$\frac{d\ell}{d\mu} = \frac{1}{\sigma^2}\sum_{i=1}^{n}(x_i-\mu) = 0$$

Solution: $\hat{\mu}_{MLE} = \frac{1}{n}\sum_{i=1}^{n} x_i = \bar{x}$
The sample mean is the MLE!

Connection to LLMs

Training neural networks is MLE!
Given training data (x_i, y_i) and model $p(y|x;\theta)$, we maximize:

$$\max_{\theta} \sum_{i=1}^{n} \log p(y_i|x_i;\theta)$$

This is exactly MLE! Minimizing cross-entropy loss = maximizing log-likelihood.

9.4 Bayesian Inference

9.4.1 The Bayesian Paradigm

In Bayesian inference, parameters are random variables with distributions!

Definition 9.6: Bayes' Theorem for Parameters

$$p(\theta|\mathbf{x}) = \frac{p(\mathbf{x}|\theta)p(\theta)}{p(\mathbf{x})}$$

- $p(\theta)$: **Prior** - our belief before seeing data
- $p(\mathbf{x}|\theta)$: **Likelihood** - how likely is the data given θ
- $p(\theta|\mathbf{x})$: **Posterior** - our belief after seeing data
- $p(\mathbf{x})$: **Evidence** or marginal likelihood

Intuition

Bayesian inference updates beliefs:

$$\text{Posterior} \propto \text{Likelihood} \times \text{Prior}$$

We start with prior knowledge, observe data, and update to the posterior.

9.4.2 Maximum A Posteriori (MAP) Estimation

Definition 9.7: MAP Estimator

$$\hat{\theta}_{MAP} = \arg\max_{\theta} p(\theta|\mathbf{x}) = \arg\max_{\theta} [p(\mathbf{x}|\theta)p(\theta)]$$

Example

If prior is $\theta \sim \mathcal{N}(0, \tau^2)$ and likelihood is Gaussian:

$$\hat{\theta}_{MAP} = \arg\max_{\theta} \left[\log p(\mathbf{x}|\theta) - \frac{\theta^2}{2\tau^2}\right]$$

The prior term $-\frac{\theta^2}{2\tau^2}$ acts as L2 regularization!

Connection to LLMs

Regularization IS Bayesian inference!

- L2 regularization = Gaussian prior
- L1 regularization = Laplace prior
- Weight decay = MAP estimation with Gaussian prior

9.5 Key Takeaways

- **Point estimation** finds single best parameter values
- **MLE** is the foundation of neural network training
- **Bayesian inference** incorporates prior knowledge and uncertainty
- **MAP estimation** connects regularization to Bayesian priors
- Understanding these concepts explains why we train models the way we do

Connection to LLMs

Statistical inference provides the theoretical foundation for learning from data. Every time you train an LLM, you're performing approximate maximum likelihood estimation on billions of parameters. The math we've covered explains why cross-entropy loss, regularization, and other training techniques work!

Part IV

Information Theory

Chapter 10

Information Theory: Measuring Information and Uncertainty

10.1 Introduction: What is Information?

Information theory, developed by Claude Shannon in 1948, provides the mathematical framework for quantifying information and uncertainty. It's fundamental to understanding LLMs.

Intuition

Information is surprise! When something unlikely happens, we gain more information than when something expected happens.
If I tell you "the sun rose this morning"—not much information (you expected it). But "it snowed in July"—lots of information (unexpected)!

Connection to LLMs

Information theory is everywhere in LLMs:

- **Cross-entropy loss** measures how well predictions match reality
- **Perplexity** evaluates language model quality
- **Entropy** quantifies uncertainty in probability distributions
- **KL divergence** measures how different two distributions are

10.2 Entropy: Measuring Uncertainty

10.2.1 Definition

Definition 10.1: Shannon Entropy

For a discrete random variable X with probability mass function $p(x)$:

$$H(X) = -\sum_x p(x) \log_2 p(x) = \mathbb{E}[-\log_2 p(X)]$$

Units: bits (if using $\log_2$) or nats (if using ln)

Intuition

Entropy measures the average "surprise" or uncertainty. Higher entropy means more unpredictable!
Think of it as: "How many yes/no questions do I need to ask (on average) to determine the outcome?"

10.2.2 Examples

Example

Fair coin: $p(\text{H}) = p(\text{T}) = 0.5$

$$H = -0.5\log_2(0.5) - 0.5\log_2(0.5) = -0.5(-1) - 0.5(-1) = 1 \text{ bit}$$

Maximum uncertainty for binary variable!

Example

Biased coin: $p(\text{H}) = 0.9, p(\text{T}) = 0.1$

$$H = -0.9\log_2(0.9) - 0.1\log_2(0.1) \approx 0.47 \text{ bits}$$

Less uncertainty—we mostly expect heads.

Example

Deterministic: $p(x_1) = 1$, all others are 0

$$H = -1 \cdot \log_2(1) = 0$$

No uncertainty at all!

10.2.3 Properties of Entropy

Theorem 10.1: Entropy Properties

- $H(X) \geq 0$ (non-negative)
- $H(X) = 0$ iff X is deterministic
- $H(X) \leq \log_2(n)$ where n is number of outcomes
- Maximum when all outcomes equally likely (uniform distribution)

10.3 Cross-Entropy: The Loss Function

10.3.1 Definition

Definition 10.2: Cross-Entropy

For true distribution p and predicted distribution q:

$$H(p,q) = -\sum_x p(x) \log q(x) = \mathbb{E}_p[-\log q(X)]$$

Intuition

Cross-entropy measures: "If reality follows distribution p, but we think it follows q, how surprised will we be on average?"
It's always at least as large as the true entropy: $H(p,q) \geq H(p)$

10.3.2 Cross-Entropy Loss in Classification

For classification with true label y and predicted probabilities $\hat{y}$:

$$\mathcal{L} = -\sum_{i=1}^{C} y_i \log \hat{y}_i$$

If y is one-hot (only class c is 1):

$$\mathcal{L} = -\log \hat{y}_c$$

Connection to LLMs

This is THE loss function for training LLMs!
When predicting the next token:

- y: true next token (one-hot vector)
- $\hat{y}$: model's predicted probability distribution
- Loss: $-\log p(\text{correct token})$

Minimizing cross-entropy = maximizing likelihood!

Example

Model predicts next word with probabilities:

$$\hat{y} = [\text{cat: } 0.6, \text{ dog: } 0.3, \text{ bird: } 0.1]$$

True word is "dog". Loss:

$$\mathcal{L} = -\log(0.3) \approx 1.20$$

If model had predicted "dog" with 0.9 probability:

$$\mathcal{L} = -\log(0.9) \approx 0.11$$

Much better!

10.4 KL Divergence: Comparing Distributions

10.4.1 Definition

Definition 10.3: Kullback-Leibler Divergence

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

Intuition

KL divergence measures how much information is lost when we approximate p with q.
Think of it as: "How different are these two distributions?"

10.4.2 Properties

Theorem 10.2: KL Divergence Properties

- $D_{KL}(p\|q) \geq 0$ (non-negative)
- $D_{KL}(p\|q) = 0$ iff $p = q$ (equal distributions)
- NOT symmetric: $D_{KL}(p\|q) \neq D_{KL}(q\|p)$
- NOT a true metric (doesn't satisfy triangle inequality)

10.4.3 Relationship to Cross-Entropy

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

Intuition

KL divergence = Cross-entropy - Entropy
The "extra" bits needed when using q instead of the optimal code for p!

Connection to LLMs

In machine learning:

- Minimizing cross-entropy = minimizing KL divergence (since $H(p)$ is constant)
- **Variational inference** uses KL divergence
- **Policy gradient methods** (RL) use KL to keep updates stable
- **Knowledge distillation** minimizes KL between teacher and student

10.5 Perplexity: Evaluating Language Models

10.5.1 Definition

Definition 10.4: Perplexity

For a language model with probability p on test sequence $w_1, \ldots, w_N$:

$$\text{Perplexity} = \exp\left(-\frac{1}{N}\sum_{i=1}^{N} \log p(w_i|w_1, \ldots, w_{i-1})\right)$$

Equivalently: Perplexity $= 2^H$ where H is the cross-entropy.

Intuition

Perplexity is roughly: "How many choices does the model effectively have at each step?"

- Perplexity of 1: Model is certain (always correct)
- Perplexity of 100: Model is as uncertain as random choice from 100 options
- Lower is better!

Example

If average cross-entropy per token is 3 bits:

$$\text{Perplexity} = 2^3 = 8$$

The model is as uncertain as picking randomly from 8 equally likely options.

Connection to LLMs

Standard benchmarks:

- GPT-2 on WikiText-103: perplexity ≈ 20
- GPT-3: perplexity ≈ 15
- Human performance: perplexity ≈ 12

Lower perplexity = better language model!

10.6 Mutual Information

10.6.1 Definition

Definition 10.5: Mutual Information

$$I(X;Y) = \sum_x \sum_y p(x,y) \log \frac{p(x,y)}{p(x)p(y)}$$

Equivalently: $I(X;Y) = H(X) - H(X|Y) = H(Y) - H(Y|X)$

Intuition

Mutual information measures: "How much does knowing Y reduce uncertainty about X?"

- $I(X;Y) = 0$: X and Y are independent
- $I(X;Y) = H(X)$: Y completely determines X

10.7 Information Theory in LLM Training

10.7.1 Maximum Likelihood = Minimum Cross-Entropy

Training objective:

$$\max_\theta \sum_{i=1}^{N} \log p_\theta(w_i|w_{<i})$$

This is equivalent to:

$$\min_\theta H(p_{\text{data}}, p_\theta)$$

Connection to LLMs

We're minimizing the KL divergence between the true data distribution and our model!

10.7.2 Temperature and Entropy

When sampling from LLMs, we use temperature T:

$$p_i = \frac{\exp(z_i/T)}{\sum_j \exp(z_j/T)}$$

- $T \to 0$: Low entropy (deterministic, boring)

- $T = 1$: Normal entropy
- $T > 1$: High entropy (creative, random)

10.8 Key Takeaways

- **Entropy** measures uncertainty/surprise in a distribution
- **Cross-entropy** is the loss function for classification and language modeling
- **KL divergence** measures difference between distributions
- **Perplexity** evaluates language model quality
- **Mutual information** measures dependency between variables
- Information theory provides the theoretical foundation for training LLMs
- Minimizing cross-entropy = maximizing likelihood = matching data distribution

Connection to LLMs

Every time an LLM trains, it's doing information theory! The loss function, evaluation metrics, and training dynamics all have deep connections to Shannon's framework. Understanding information theory helps you understand why certain loss functions work, how to evaluate models, and what it means for a model to "learn" the data distribution.

Part V

Neural Networks

Chapter 11

Neural Network Mathematics

11.1 Introduction: From Neurons to Networks

We've learned linear algebra, calculus, probability, and optimization. Now it's time to put it all together and understand how neural networks actually work mathematically.

Intuition

A neural network is just a giant composite function! It takes an input (like a word embedding) and applies layers of transformations to produce an output (like a probability distribution over next words). All the math we've learned comes together here:

- **Linear algebra**: Matrix multiplications in each layer
- **Calculus**: Computing gradients for learning
- **Probability**: Interpreting outputs as distributions
- **Optimization**: Training the network

Connection to LLMs

Every large language model is fundamentally a deep neural network with billions of parameters. Understanding the mathematics of basic neural networks is the foundation for understanding transformers and LLMs.

11.2 The Perceptron: A Single Neuron

11.2.1 Biological Inspiration

Artificial neurons are loosely inspired by biological neurons:

- **Inputs** (dendrites): Receive signals
- **Weights**: Strength of connections
- **Activation** (cell body): Combines inputs
- **Output** (axon): Sends signal forward

11.2.2 Mathematical Definition

Definition 11.1: Perceptron

A perceptron computes:

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

where:

- $\mathbf{x} \in \mathbb{R}^n$ is the input vector
- $\mathbf{w} \in \mathbb{R}^n$ are the weights
- $b \in \mathbb{R}$ is the bias
- f is the activation function

Intuition

The perceptron computes a weighted sum of inputs (linear operation), adds a bias (shift), then applies a nonlinear function. This simple operation, when stacked in layers, becomes incredibly powerful!

Example

Input: $\mathbf{x} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$, weights: $\mathbf{w} = \begin{bmatrix} 0.5 \\ -0.3 \\ 0.8 \end{bmatrix}$, bias: $b = 0.1$

Linear combination:

$$z = 0.5(1) + (-0.3)(2) + 0.8(3) + 0.1 = 0.5 - 0.6 + 2.4 + 0.1 = 2.4$$

With sigmoid activation $\sigma(z) = \frac{1}{1+e^{-z}}$:

$$y = \sigma(2.4) = \frac{1}{1+e^{-2.4}} \approx 0.917$$

11.3 Activation Functions

The activation function introduces nonlinearity—crucial for learning complex patterns!

11.3.1 Why Nonlinearity?

Common Pitfall

Without activation functions, stacking layers would be useless! Multiple linear transformations compose to one linear transformation:

$$\mathbf{W}_3(\mathbf{W}_2(\mathbf{W}_1\mathbf{x})) = (\mathbf{W}_3\mathbf{W}_2\mathbf{W}_1)\mathbf{x} = \mathbf{W}_{\text{combined}}\mathbf{x}$$

A deep linear network is equivalent to a single linear layer! Activation functions break this and enable learning complex, nonlinear patterns.

11.3.2 Common Activation Functions

Sigmoid

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

Properties:

- Range: $(0, 1)$
- Derivative: $\sigma'(x) = \sigma(x)(1 - \sigma(x))$
- Interpretation: Probability/gating
- Problem: Vanishing gradients for large $|x|$

Hyperbolic Tangent (tanh)

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

Properties:

- Range: $(-1, 1)$
- Derivative: $\tanh'(x) = 1 - \tanh^2(x)$
- Zero-centered (better than sigmoid)
- Still suffers from vanishing gradients

ReLU (Rectified Linear Unit)

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

Properties:

- Range: $[0, \infty)$
- Derivative: $\text{ReLU}'(x) = \begin{cases} 1 & x > 0 \\ 0 & x \leq 0 \end{cases}$
- No vanishing gradient for $x > 0$
- Computationally cheap
- Can "die" (always output 0)

Connection to LLMs

ReLU is the default activation in most modern architectures! Its simplicity and good gradient properties made deep learning practical.

Leaky ReLU

$$\text{LeakyReLU}(x) = \begin{cases} x & x > 0 \\ \alpha x & x \leq 0 \end{cases}$$

where α is small (e.g., 0.01). Prevents "dying ReLU" problem.

GELU (Gaussian Error Linear Unit)

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

where $\Phi(x)$ is the standard normal CDF.

Connection to LLMs

GELU is used in modern transformers (GPT, BERT)! It's smoother than ReLU and has nice probabilistic interpretation.

Swish/SiLU

$$\text{Swish}(x) = x \cdot \sigma(x)$$

Properties:

- Smooth and non-monotonic
- Self-gated
- Used in some modern architectures

11.4 Feedforward Neural Networks

11.4.1 Architecture

A feedforward neural network stacks multiple layers:

$$\mathbf{x} \xrightarrow{\text{Layer 1}} \mathbf{h}_1 \xrightarrow{\text{Layer 2}} \mathbf{h}_2 \xrightarrow{\cdots} \mathbf{h}_{L-1} \xrightarrow{\text{Layer L}} \mathbf{y}$$

Each layer computes:

$$\mathbf{h}_{i+1} = f(\mathbf{W}_i \mathbf{h}_i + \mathbf{b}_i)$$

Definition 11.2: Multilayer Perceptron (MLP)

An L-layer MLP with input $\mathbf{x} \in \mathbb{R}^{d_0}$ computes:

$$\begin{aligned} \mathbf{h}_1 &= f_1(\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1) \\ \mathbf{h}_2 &= f_2(\mathbf{W}_2 \mathbf{h}_1 + \mathbf{b}_2) \\ &\vdots \\ \mathbf{y} &= f_L(\mathbf{W}_L \mathbf{h}_{L-1} + \mathbf{b}_L) \end{aligned}$$

where $\mathbf{W}_i \in \mathbb{R}^{d_i \times d_{i-1}}$ and $\mathbf{b}_i \in \mathbb{R}^{d_i}$.

11.4.2 Universal Approximation Theorem

Theorem 11.1: Universal Approximation Theorem

A feedforward network with:

- A single hidden layer
- Finite number of neurons
- Non-polynomial activation function

can approximate any continuous function on compact subsets of $\mathbb{R}^n$ to arbitrary precision.

Intuition

Neural networks are universal function approximators! With enough neurons, they can represent any function you want (at least in theory).
However, finding the right parameters (training) is the hard part!

11.5 Loss Functions

To train a network, we need to measure how wrong it is.

11.5.1 Mean Squared Error (Regression)

$$\mathcal{L}(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Used when predicting continuous values.

11.5.2 Cross-Entropy Loss (Classification)

For binary classification:

$$\mathcal{L} = -\frac{1}{n}\sum_{i=1}^{n}[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)]$$

For multi-class classification:

$$\mathcal{L} = -\frac{1}{n}\sum_{i=1}^{n}\sum_{c=1}^{C} y_{ic}\log(\hat{y}_{ic})$$

Connection to LLMs

Cross-entropy is THE loss function for language models! When predicting the next token, we're doing multi-class classification over the vocabulary.

11.5.3 Softmax for Multi-Class Output

Definition 11.3: Softmax

Converts logits $\mathbf{z} \in \mathbb{R}^C$ to probabilities:

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

Properties:

- All outputs positive: $\hat{y}_i > 0$
- Sums to 1: $\sum_i \hat{y}_i = 1$
- Differentiable
- Amplifies differences (exponential)

Example

Logits: $\mathbf{z} = [2.0, 1.0, 0.1]$

$$\hat{y}_1 = \frac{e^{2.0}}{e^{2.0} + e^{1.0} + e^{0.1}} = \frac{7.39}{7.39 + 2.72 + 1.11} = 0.66$$
$$\hat{y}_2 = \frac{e^{1.0}}{11.22} = 0.24$$
$$\hat{y}_3 = \frac{e^{0.1}}{11.22} = 0.10$$

11.6 Forward Propagation

11.6.1 The Forward Pass

Computing the output given input is straightforward—just apply layers sequentially!

Example

Network: Input (2) $\to$ Hidden (3) $\to$ Output (2)
Given:

$$\mathbf{x} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}, \quad \mathbf{W}_1 = \begin{bmatrix} 0.1 & 0.2 \\ 0.3 & 0.4 \\ 0.5 & 0.6 \end{bmatrix}, \quad \mathbf{b}_1 = \begin{bmatrix} 0.1 \\ 0.2 \\ 0.3 \end{bmatrix}$$

Layer 1:

$$\mathbf{z}_1 = \mathbf{W}_1\mathbf{x} + \mathbf{b}_1 = \begin{bmatrix} 0.1(1) + 0.2(2) + 0.1 \\ 0.3(1) + 0.4(2) + 0.2 \\ 0.5(1) + 0.6(2) + 0.3 \end{bmatrix} = \begin{bmatrix} 0.6 \\ 1.3 \\ 2.0 \end{bmatrix}$$

Apply ReLU:

$$\mathbf{h}_1 = \text{ReLU}(\mathbf{z}_1) = \begin{bmatrix} 0.6 \\ 1.3 \\ 2.0 \end{bmatrix}$$

Continue to output layer...

11.7 Key Concepts for Deep Learning

11.7.1 Depth vs Width

- **Depth**: Number of layers
- **Width**: Number of neurons per layer

Theorem 11.2: Depth vs Width (Informal)

Deep networks (many layers) can represent functions more efficiently than wide networks (many neurons per layer). Some functions require exponentially more neurons in shallow networks compared to deep ones.

11.7.2 Expressiveness

More parameters = more capacity to fit complex functions
But also:

- More data needed
- Higher risk of overfitting
- More computation required

11.8 Practice Problems

11.8.1 Problem 1: Manual Forward Pass

Given a single neuron with:

- Input: $\mathbf{x} = [2, -1, 3]$
- Weights: $\mathbf{w} = [0.5, 0.8, -0.3]$
- Bias: $b = 0.2$
- Activation: sigmoid

Compute the output.

11.8.2 Problem 2: Network Architecture

Design a network architecture for:

- Input: 784-dimensional vector (28×28 image)
- Output: 10 classes (digits 0-9)
- Constraint: Use 2 hidden layers

Specify dimensions and activation functions.

11.8.3 Problem 3: Activation Functions

Why can't we use only sigmoid activations in very deep networks? What problems arise?

11.8.4 Problem 4: Universal Approximation

If neural networks can approximate any function, why do we need deep networks? Why not use one wide hidden layer?

11.9 Key Takeaways

- **Neural networks** are compositions of linear transforms and nonlinear activations
- **Activation functions** introduce nonlinearity for complex function learning
- **ReLU** is the most common activation due to good gradients

- **Feedforward networks** stack layers to build deep representations
- **Cross-entropy loss** is standard for classification
- **Softmax** converts logits to probability distributions
- **Universal approximation** guarantees representational power
- Understanding forward propagation is essential before backpropagation

Connection to LLMs

You now understand the mathematical building blocks of neural networks! In the next chapter, we'll learn how networks actually learn—through backpropagation, the algorithm that computes gradients efficiently using the chain rule. This is where calculus and neural networks unite!

Chapter 12

Backpropagation: How Neural Networks Learn

12.1 Introduction: The Learning Algorithm

Backpropagation is the most important algorithm in deep learning. It's how neural networks learn from data. Despite its reputation for being complex, backpropagation is just the chain rule applied systematically!

Intuition

Imagine you're hiking down a mountain in the fog. You can only see a few feet ahead. Backpropagation is like feeling the slope beneath your feet at each step and using that information to figure out which direction to walk.
More precisely: given an error at the output, backpropagation tells us how to adjust each weight in the network to reduce that error.

Connection to LLMs

Every time an LLM learns—whether during pre-training on trillions of tokens or fine-tuning on a specific task—it's using backpropagation. Understanding this algorithm is understanding how AI learns!

12.2 The Setup: Forward Pass

12.2.1 A Simple Neural Network

Consider a simple 2-layer network:

$$\text{Input } \mathbf{x} \xrightarrow{\mathbf{W}^{(1)}} \mathbf{z}^{(1)} \xrightarrow{\sigma} \mathbf{a}^{(1)} \xrightarrow{\mathbf{W}^{(2)}} \mathbf{z}^{(2)} \xrightarrow{\sigma} \mathbf{a}^{(2)} \xrightarrow{\text{Loss}} \mathcal{L}$$

Mathematically:

$$\begin{aligned}
\mathbf{z}^{(1)} &= \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)} \\
\mathbf{a}^{(1)} &= \sigma(\mathbf{z}^{(1)}) \\
\mathbf{z}^{(2)} &= \mathbf{W}^{(2)}\mathbf{a}^{(1)} + \mathbf{b}^{(2)} \\
\mathbf{a}^{(2)} &= \sigma(\mathbf{z}^{(2)}) \\
\mathcal{L} &= \text{loss}(\mathbf{a}^{(2)}, \mathbf{y})
\end{aligned}$$

12.2.2 Forward Pass Example

Example

Input: $\mathbf{x} = \begin{bmatrix} 0.5 \\ 0.3 \end{bmatrix}$

Weights: $\mathbf{W}^{(1)} = \begin{bmatrix} 0.2 & 0.8 \\ 0.5 & 0.1 \end{bmatrix}$, $\mathbf{b}^{(1)} = \begin{bmatrix} 0.1 \\ 0.2 \end{bmatrix}$

Layer 1:

$$\mathbf{z}^{(1)} = \begin{bmatrix} 0.2 & 0.8 \\ 0.5 & 0.1 \end{bmatrix} \begin{bmatrix} 0.5 \\ 0.3 \end{bmatrix} + \begin{bmatrix} 0.1 \\ 0.2 \end{bmatrix} = \begin{bmatrix} 0.44 \\ 0.48 \end{bmatrix}$$

With ReLU: $\mathbf{a}^{(1)} = \begin{bmatrix} 0.44 \\ 0.48 \end{bmatrix}$

Continue this process through all layers to get final output and loss.

12.3 The Goal: Computing Gradients

We want to compute:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(1)}}, \quad \frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(1)}}, \quad \frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(2)}}, \quad \frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(2)}}$$

These tell us how to adjust each weight to reduce the loss!

Intuition

$\frac{\partial \mathcal{L}}{\partial w_{ij}}$ answers: "If I increase weight w_{ij} by a tiny amount, how much does the loss change?"

If positive: increasing the weight increases loss (bad!) $\rightarrow$ decrease the weight If negative: increasing the weight decreases loss (good!) $\rightarrow$ increase the weight

12.4 Backward Pass: The Chain Rule in Action

12.4.1 Output Layer Gradient

Start at the end and work backwards!

Step 1: Loss to Output

For MSE loss: $\mathcal{L} = \frac{1}{2}\left\|\mathbf{a}^{(2)} - \mathbf{y}\right\|^2$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{a}^{(2)}} = \mathbf{a}^{(2)} - \mathbf{y}$$

Step 2: Output to Pre-Activation

$$\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(2)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{a}^{(2)}} \odot \sigma'(\mathbf{z}^{(2)})$$

where $\odot$ is element-wise multiplication.

Intuition

This is the chain rule! We multiply by the derivative of the activation function.

For ReLU: $\sigma'(z) = \begin{cases} 1 & z > 0 \\ 0 & z \leq 0 \end{cases}$

So gradients only flow through active neurons!

Step 3: Gradient with Respect to Weights

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(2)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(2)}}(\mathbf{a}^{(1)})^\top$$

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

Intuition

Why this form? Recall $\mathbf{z}^{(2)} = \mathbf{W}^{(2)}\mathbf{a}^{(1)} + \mathbf{b}^{(2)}$

The gradient has the form of an outer product: it tells us how much each weight contributed to each output.

12.4.2 Hidden Layer Gradient

Now propagate back to layer 1:

Step 4: Backpropagate Through Weights

$$\frac{\partial \mathcal{L}}{\partial \mathbf{a}^{(1)}} = (\mathbf{W}^{(2)})^\top \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(2)}}$$

Intuition

The gradient flows backwards through the transpose of the weight matrix! This makes sense: forward pass multiplies by $\mathbf{W}$, backward pass multiplies by $\mathbf{W}^\top$.

Step 5: Through Activation

$$\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(1)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{a}^{(1)}} \odot \sigma'(\mathbf{z}^{(1)})$$

Step 6: To First Layer Weights

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(1)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(1)}} \mathbf{x}^\top$$

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

12.5 The Backpropagation Algorithm

Definition 12.1: Backpropagation Algorithm

Forward Pass:

1. Compute activations layer by layer: $\mathbf{a}^{(0)} = \mathbf{x}$
2. For $\ell = 1$ to L:
   - $\mathbf{z}^{(\ell)} = \mathbf{W}^{(\ell)}\mathbf{a}^{(\ell-1)} + \mathbf{b}^{(\ell)}$
   - $\mathbf{a}^{(\ell)} = \sigma(\mathbf{z}^{(\ell)})$
3. Compute loss: $\mathcal{L} = \text{loss}(\mathbf{a}^{(L)}, \mathbf{y})$

Backward Pass:

1. Compute output gradient: $\delta^{(L)} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(L)}}$
2. For $\ell = L$ down to 1:
   - $\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(\ell)}} = \delta^{(\ell)}(\mathbf{a}^{(\ell-1)})^\top$
   - $\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(\ell)}} = \delta^{(\ell)}$
   - $\delta^{(\ell-1)} = ((\mathbf{W}^{(\ell)})^\top \delta^{(\ell)}) \odot \sigma'(\mathbf{z}^{(\ell-1)})$

12.6 Computational Efficiency

12.6.1 Why Backpropagation is Efficient

Computing gradients naively would require:

- For each parameter: perturb it, compute loss, measure change
- $O(P)$ forward passes where P is number of parameters
- For GPT-3: 175 billion forward passes!

Backpropagation computes all gradients in:

- 1 forward pass + 1 backward pass
- $O(P)$ total operations
- Backward pass costs roughly $2\times$ forward pass

Connection to LLMs

This is why deep learning is possible! Without backpropagation's efficiency, we couldn't train large models.

12.7 Common Activation Functions and Their Derivatives

12.7.1 Sigmoid

$$\sigma(z) = \frac{1}{1+e^{-z}}, \quad \sigma'(z) = \sigma(z)(1-\sigma(z))$$

Common Pitfall

Sigmoid derivatives are very small when $|z|$ is large (vanishing gradients!)

12.7.2 Tanh

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}, \quad \tanh'(z) = 1 - \tanh^2(z)$$

12.7.3 ReLU

$$\text{ReLU}(z) = \max(0, z), \quad \text{ReLU}'(z) = \begin{cases} 1 & z > 0 \\ 0 & z \leq 0 \end{cases}$$

Intuition

ReLU's derivative is simple: 1 or 0. This makes backprop fast and avoids vanishing gradients for positive inputs!

12.7.4 Softmax (for classification)

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

The Jacobian is:

$$\frac{\partial \text{softmax}(\mathbf{z})_i}{\partial z_j} = \text{softmax}(\mathbf{z})_i(\delta_{ij} - \text{softmax}(\mathbf{z})_j)$$

where δ_{ij} is the Kronecker delta.

12.8 Backpropagation Through Different Operations

12.8.1 Matrix Multiplication

Forward: $\mathbf{y} = \mathbf{W}\mathbf{x}$
Backward:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}} = \frac{\partial \mathcal{L}}{\partial \mathbf{y}}\mathbf{x}^\top, \quad \frac{\partial \mathcal{L}}{\partial \mathbf{x}} = \mathbf{W}^\top \frac{\partial \mathcal{L}}{\partial \mathbf{y}}$$

12.8.2 Element-wise Operations

Forward: $\mathbf{y} = \mathbf{a} \odot \mathbf{b}$
Backward:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{a}} = \frac{\partial \mathcal{L}}{\partial \mathbf{y}} \odot \mathbf{b}, \quad \frac{\partial \mathcal{L}}{\partial \mathbf{b}} = \frac{\partial \mathcal{L}}{\partial \mathbf{y}} \odot \mathbf{a}$$

12.8.3 Concatenation

Forward: $\mathbf{z} = [\mathbf{a}; \mathbf{b}]$
Backward: Split the gradient appropriately

12.8.4 Sum/Mean

Forward: $y = \sum_i x_i$
Backward: $\frac{\partial \mathcal{L}}{\partial x_i} = \frac{\partial \mathcal{L}}{\partial y}$ for all i

Intuition

Gradients "broadcast" backwards—the same gradient flows to all inputs that contributed to the sum.

12.9 Automatic Differentiation

Modern frameworks (PyTorch, TensorFlow) implement **automatic differentiation**:

Definition 12.2: Computational Graph

Operations are represented as a directed acyclic graph (DAG):

- Nodes: Variables and operations
- Edges: Data flow

Forward pass: Traverse graph forward, computing values Backward pass: Traverse graph backward, computing gradients

Connection to LLMs

When you write:

```
loss.backward()
```

PyTorch automatically traverses the computational graph and applies backpropagation!

12.10 Vanishing and Exploding Gradients

12.10.1 The Problem

In deep networks: $\frac{\partial \mathcal{L}}{\partial \mathbf{a}^{(1)}} = \prod_{\ell=2}^{L} (\mathbf{W}^{(\ell)})^\top \text{diag}(\sigma'(\mathbf{z}^{(\ell-1)}))$

This is a product of many matrices!

Common Pitfall

Vanishing Gradients: If $\left\|\mathbf{W}^{(\ell)}\right\| < 1$ or σ' is small, gradients exponentially decay. Early layers get tiny gradients and learn very slowly.
Exploding Gradients: If $\left\|\mathbf{W}^{(\ell)}\right\| > 1$, gradients exponentially grow, causing numerical instability.

12.10.2 Solutions

- **Better activations**: ReLU avoids vanishing for positive inputs
- **Careful initialization**: Xavier/He initialization
- **Batch normalization**: Normalizes activations
- **Residual connections**: Skip connections
- **Gradient clipping**: Cap gradient magnitude
- **Layer normalization**: Used in transformers

12.11 Backpropagation Through Time (BPTT)

For recurrent networks processing sequences:

$$\mathbf{h}_t = \sigma(\mathbf{W}\mathbf{h}_{t-1} + \mathbf{U}\mathbf{x}_t)$$

Gradients must flow backwards through time:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}} = \sum_{t=1}^{T} \frac{\partial \mathcal{L}_t}{\partial \mathbf{W}}$$

Common Pitfall

BPTT suffers from vanishing gradients over long sequences—this is why transformers with attention replaced RNNs for language modeling!

12.12 Practice Problems

12.12.1 Problem 1: Manual Backprop

Given a 2-layer network with ReLU:

- $\mathbf{x} = [1, 2]$, $\mathbf{y} = [1]$
- $\mathbf{W}^{(1)} = \begin{bmatrix} 0.5 & 0.5 \\ -0.5 & 0.5 \end{bmatrix}$, $\mathbf{b}^{(1)} = [0, 0]$
- $\mathbf{W}^{(2)} = \begin{bmatrix} 1 & 1 \end{bmatrix}$, $b^{(2)} = 0$
- Loss: $\mathcal{L} = \frac{1}{2}(a^{(2)} - y)^2$

Compute all gradients by hand (forward pass, then backward pass).

12.12.2 Problem 2: Derivative Verification

Implement numerical gradient checking to verify your backpropagation implementation.

12.12.3 Problem 3: Gradient Flow

For a 10-layer network with sigmoid activations, if all weights have eigenvalues around 0.5, what happens to gradients at layer 1?

12.13 Key Takeaways

- **Backpropagation is the chain rule** applied systematically
- **Forward pass**: Compute outputs layer by layer
- **Backward pass**: Compute gradients layer by layer (in reverse)
- **Efficiency**: All gradients computed in one forward + one backward pass
- **Local gradients**: Each operation only needs its local derivative
- **Vanishing/exploding gradients** are major challenges in deep networks
- Modern frameworks handle backpropagation automatically via computational graphs
- Understanding backpropagation is understanding how all neural networks learn

Connection to LLMs

Backpropagation is what makes training LLMs possible. When GPT models learn language patterns from billions of examples, backpropagation is computing gradients for 175 billion parameters, determining how to adjust each tiny weight to improve predictions. Now you understand the fundamental learning algorithm of AI!

Chapter 13

Advanced Optimization: Training Neural Networks Efficiently

13.1 Introduction: Beyond Basic Gradient Descent

Simple gradient descent has a problem: it's too simple! Modern neural networks require sophisticated optimization techniques to train efficiently.

Intuition

Imagine hiking down a mountain in thick fog. Basic gradient descent is like always taking steps downhill, but you might:

- Get stuck in ravines (poor conditioning)
- Oscillate back and forth (high learning rate)
- Move too slowly (low learning rate)
- Miss the best path entirely (local minima)

Advanced optimizers address these issues!

13.2 Momentum: Accelerating Gradient Descent

13.2.1 The Basic Idea

Momentum accumulates gradients over time, like a ball rolling downhill.

Definition 13.1: Momentum

$$\begin{aligned}\mathbf{v}^{(t)} &= \beta \mathbf{v}^{(t-1)} + (1-\beta)\nabla_{\mathbf{w}}\mathcal{L}(\mathbf{w}^{(t)}) \\ \mathbf{w}^{(t+1)} &= \mathbf{w}^{(t)} - \eta \mathbf{v}^{(t)}\end{aligned}$$

where $\beta \in [0,1]$ is the momentum coefficient (typically 0.9).

Intuition

The velocity $\mathbf{v}^{(t)}$ is an exponential moving average of gradients. This helps:

- Accelerate in consistent directions
- Dampen oscillations
- Escape shallow local minima

13.3 RMSprop: Adaptive Learning Rates

Definition 13.2: RMSprop

$$\begin{aligned}\mathbf{s}^{(t)} &= \beta \mathbf{s}^{(t-1)} + (1-\beta)(\nabla_{\mathbf{w}}\mathcal{L})^2 \\ \mathbf{w}^{(t+1)} &= \mathbf{w}^{(t)} - \frac{\eta}{\sqrt{\mathbf{s}^{(t)}+\epsilon}} \odot \nabla_{\mathbf{w}}\mathcal{L}\end{aligned}$$

where $\odot$ is element-wise multiplication and $\epsilon \approx 10^{-8}$ for numerical stability.

13.4 Adam: The Default Choice

13.4.1 Combining Momentum and RMSprop

Definition 13.3: Adam Optimizer

Adam maintains both first and second moment estimates:

$$\mathbf{m}^{(t)} = \beta_1 \mathbf{m}^{(t-1)} + (1-\beta_1)\nabla_{\mathbf{w}}\mathcal{L} \quad \text{(momentum)}$$
$$\mathbf{v}^{(t)} = \beta_2 \mathbf{v}^{(t-1)} + (1-\beta_2)(\nabla_{\mathbf{w}}\mathcal{L})^2 \quad \text{(RMSprop)}$$

Bias correction:

$$\hat{\mathbf{m}}^{(t)} = \frac{\mathbf{m}^{(t)}}{1-\beta_1^t}$$
$$\hat{\mathbf{v}}^{(t)} = \frac{\mathbf{v}^{(t)}}{1-\beta_2^t}$$

Update:

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \frac{\eta}{\sqrt{\hat{\mathbf{v}}^{(t)}}+\epsilon} \odot \hat{\mathbf{m}}^{(t)}$$

Typical values: $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\eta = 10^{-3}$

Connection to LLMs

Adam is the default optimizer for training LLMs! It's robust, requires little tuning, and works well across diverse tasks.

13.5 AdamW: Weight Decay Done Right

Definition 13.4: AdamW

Instead of adding L2 regularization to the loss, decouple weight decay:

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta\left[\frac{\hat{\mathbf{m}}^{(t)}}{\sqrt{\hat{\mathbf{v}}^{(t)}}+\epsilon} + \lambda \mathbf{w}^{(t)}\right]$$

where λ is the weight decay coefficient.

Connection to LLMs

Most modern LLMs use AdamW! The decoupled weight decay improves generalization.

13.6 Learning Rate Schedules

13.6.1 Warmup

Start with small learning rate and gradually increase:

$$\eta^{(t)} = \eta_{\text{max}} \cdot \min\left(1, \frac{t}{T_{\text{warmup}}}\right)$$

13.6.2 Cosine Annealing

After warmup, decay using cosine:

$$\eta^{(t)} = \eta_{\text{min}} + \frac{1}{2}(\eta_{\text{max}} - \eta_{\text{min}})\left(1 + \cos\left(\frac{t - T_{\text{warmup}}}{T_{\text{max}}}\pi\right)\right)$$

13.7 Key Takeaways

- **Momentum** accelerates convergence
- **Adam** adapts learning rates per parameter
- **AdamW** is the standard for LLM training
- **Learning rate schedules** are crucial for good performance

Connection to LLMs

These optimizers make training billion-parameter models feasible! Without them, modern LLMs wouldn't exist.

Part VI

Transformers & Large Language Models

Chapter 14

Attention Mechanisms: The Heart of Transformers

14.1 Introduction: The Attention Revolution

In 2017, a paper titled "Attention Is All You Need" changed everything. Attention solved the problem of long-range dependencies by allowing every element to directly "look at" every other element.

Intuition

Imagine reading a sentence: "The animal didn't cross the street because it was too tired."
What does "it" refer to? The animal or the street? Humans instantly focus *attention* on "animal" to resolve this. Attention mechanisms do exactly this—they learn which words to focus on!

Connection to LLMs

Attention mechanisms are the core innovation that makes LLMs possible. Understanding attention is understanding how these models work.

14.2 The Problem: Sequential Processing Limitations

14.2.1 Recurrent Neural Networks (RNNs)

Before transformers, sequences were processed with RNNs:

$$\mathbf{h}_t = f(\mathbf{W}\mathbf{h}_{t-1} + \mathbf{U}\mathbf{x}_t)$$

Common Pitfall

RNNs have serious problems:

- **Sequential processing**: Can't parallelize (must compute $\mathbf{h}_1$ before $\mathbf{h}_2$)
- **Vanishing gradients**: Information from early tokens gets lost
- **Fixed context**: Hidden state has fixed size, creating information bottleneck
- **Long sequences**: Nearly impossible to capture dependencies 100+ tokens apart

14.2.2 The Attention Solution

Instead of compressing everything into a fixed-size hidden state, attention lets each token directly access information from all other tokens!

Intuition

Think of attention as a database lookup:

- You have a **query**: "What does 'it' refer to?"
- You have **keys**: Labels for each word in the sentence
- You have **values**: The actual information from each word
- Attention computes: "Which keys match my query?" and retrieves corresponding values

14.3 Scaled Dot-Product Attention

14.3.1 The Basic Mechanism

This is the fundamental attention operation!

Definition 14.1: Scaled Dot-Product Attention

Given:

- Queries: $\mathbf{Q} \in \mathbb{R}^{n \times d_k}$ (what we're looking for)
- Keys: $\mathbf{K} \in \mathbb{R}^{m \times d_k}$ (what's available to look at)

- Values: $\mathbf{V} \in \mathbb{R}^{m \times d_v}$ (the actual information)

Attention is computed as:

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

Let's break this down step by step!

14.3.2 Step 1: Compute Similarity Scores

First, compute how much each query matches each key:

$$\mathbf{S} = \mathbf{Q}\mathbf{K}^\top \in \mathbb{R}^{n \times m}$$

Element s_{ij} is the dot product of query i with key j:

$$s_{ij} = \mathbf{q}_i \cdot \mathbf{k}_j = \sum_{d=1}^{d_k} q_{id} k_{jd}$$

Intuition

The dot product measures similarity! Large dot product means the query and key are pointing in similar directions—they're related! This is why we normalized vectors and learned about dot products in Chapter 1. It all comes together here!

14.3.3 Step 2: Scale the Scores

Divide by $\sqrt{d_k}$:

$$\mathbf{S}_{\text{scaled}} = \frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}$$

Intuition

Why scale? When d_k is large, dot products can become very large in magnitude. This pushes the softmax into regions with extremely small gradients (saturation).
Dividing by $\sqrt{d_k}$ keeps the variance of dot products roughly constant regardless of dimension. It's a numerical stability trick!

Example

If $\mathbf{q}$ and $\mathbf{k}$ have entries drawn from $\mathcal{N}(0,1)$, then:

$$\mathbf{q} \cdot \mathbf{k} = \sum_{i=1}^{d_k} q_i k_i \sim \mathcal{N}(0, d_k)$$

The variance grows with d_k! Dividing by $\sqrt{d_k}$ gives:

$$\frac{\mathbf{q} \cdot \mathbf{k}}{\sqrt{d_k}} \sim \mathcal{N}(0,1)$$

Now variance is constant!

14.3.4 Step 3: Apply Softmax

Convert scores to probabilities:

$$\mathbf{A} = \text{softmax}\left(\frac{\mathbf{QK}^\top}{\sqrt{d_k}}\right)$$

For each row i (each query):

$$a_{ij} = \frac{\exp(s_{ij}/\sqrt{d_k})}{\sum_{k=1}^{m} \exp(s_{ik}/\sqrt{d_k})}$$

Intuition

Softmax converts arbitrary scores to a probability distribution:

- All values are positive: $a_{ij} \geq 0$
- They sum to 1: $\sum_j a_{ij} = 1$
- Larger scores get larger probabilities (exponentially!)

These are **attention weights**: a_{ij} tells us how much query i should attend to key/value j.

14.3.5 Step 4: Weighted Sum of Values

Finally, compute the output as a weighted combination of values:

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

For each query i, the output is:

$$\mathbf{o}_i = \sum_{j=1}^{m} a_{ij}\mathbf{v}_j$$

Intuition

This is the key insight! Each output is a weighted average of all values, where the weights come from how well the query matched each key.
If query i strongly matches key j, then a_{ij} is large, so $\mathbf{v}_j$ contributes heavily to $\mathbf{o}_i$.

14.4 Concrete Example: Attention in Action

Let's work through a tiny example with 3 words and dimension 2.

14.4.1 Setup

Consider the sentence: "cat sat mat"

Suppose we have:

$$\mathbf{Q} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix}, \quad \mathbf{K} = \begin{bmatrix} 1 & 0 \\ 0.5 & 0.5 \\ 1 & 1 \end{bmatrix}, \quad \mathbf{V} = \begin{bmatrix} 2 & 0 \\ 0 & 3 \\ 1 & 1 \end{bmatrix}$$

14.4.2 Step 1: Compute Scores

$$\mathbf{Q}\mathbf{K}^\top = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0.5 & 1 \\ 0 & 0.5 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0.5 & 1 \\ 0 & 0.5 & 1 \\ 1 & 1 & 2 \end{bmatrix}$$

14.4.3 Step 2: Scale

With $d_k = 2$, we divide by $\sqrt{2} \approx 1.414$:

$$\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}} = \begin{bmatrix} 0.707 & 0.354 & 0.707 \\ 0 & 0.354 & 0.707 \\ 0.707 & 0.707 & 1.414 \end{bmatrix}$$

14.4.4 Step 3: Softmax

For row 1: $[0.707, 0.354, 0.707]$

$$\begin{aligned}
a_{11} &= \frac{e^{0.707}}{e^{0.707} + e^{0.354} + e^{0.707}} \approx 0.364 \\
a_{12} &= \frac{e^{0.354}}{e^{0.707} + e^{0.354} + e^{0.707}} \approx 0.272 \\
a_{13} &= \frac{e^{0.707}}{e^{0.707} + e^{0.354} + e^{0.707}} \approx 0.364
\end{aligned}$$

So attention weights for query 1: $[0.364, 0.272, 0.364]$

14.4.5 Step 4: Weighted Sum

Output for query 1:

$$\begin{aligned}
\mathbf{o}_1 &= 0.364 \begin{bmatrix} 2 \\ 0 \end{bmatrix} + 0.272 \begin{bmatrix} 0 \\ 3 \end{bmatrix} + 0.364 \begin{bmatrix} 1 \\ 1 \end{bmatrix} \\
&= \begin{bmatrix} 0.728 + 0 + 0.364 \\ 0 + 0.816 + 0.364 \end{bmatrix} = \begin{bmatrix} 1.092 \\ 1.180 \end{bmatrix}
\end{aligned}$$

14.5 Self-Attention: Attending to Yourself

14.5.1 The Key Idea

In self-attention, the same sequence serves as queries, keys, and values!

Definition 14.2: Self-Attention

Given input sequence $\mathbf{X} \in \mathbb{R}^{n \times d}$, we project to Q, K, V:

$$\begin{aligned}
\mathbf{Q} &= \mathbf{X}\mathbf{W}^Q \\
\mathbf{K} &= \mathbf{X}\mathbf{W}^K \\
\mathbf{V} &= \mathbf{X}\mathbf{W}^V
\end{aligned}$$

where $\mathbf{W}^Q, \mathbf{W}^K \in \mathbb{R}^{d \times d_k}$ and $\mathbf{W}^V \in \mathbb{R}^{d \times d_v}$ are learned weight matrices.
Then apply attention:

$$\text{SelfAttention}(\mathbf{X}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right)\mathbf{V}$$

Intuition

Each word asks: "Which other words in this sentence should I pay attention to?"
For example, in "The cat sat on the mat":

- "sat" might attend to "cat" (who sat?) and "mat" (where?)
- "it" might attend to "cat" (what does "it" refer to?)

The model learns these attention patterns from data!

14.6 Multi-Head Attention

14.6.1 The Motivation

Different heads can learn different types of relationships!

Intuition

Think of it like having multiple specialized detectors:

- Head 1: Finds syntactic relationships (subject-verb)
- Head 2: Finds semantic relationships (synonyms, antonyms)
- Head 3: Finds positional relationships (nearby words)
- Head 4: Finds long-range dependencies

Each head gets to ask different questions about the input!

14.6.2 Mathematical Definition

Definition 14.3: Multi-Head Attention

Instead of one attention operation, we perform h parallel attention operations:
For head i:

$$\begin{aligned}
\mathbf{Q}_i &= \mathbf{X}\mathbf{W}_i^Q \\
\mathbf{K}_i &= \mathbf{X}\mathbf{W}_i^K \\
\mathbf{V}_i &= \mathbf{X}\mathbf{W}_i^V \\
\text{head}_i &= \text{Attention}(\mathbf{Q}_i, \mathbf{K}_i, \mathbf{V}_i)
\end{aligned}$$

Concatenate all heads and project:

$$\text{MultiHead}(\mathbf{X}) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)\mathbf{W}^O$$

where $\mathbf{W}^O \in \mathbb{R}^{hd_v \times d}$ is the output projection.

14.6.3 Computational Details

Typical hyperparameters (e.g., GPT-2):

- Model dimension: $d = 768$
- Number of heads: $h = 12$
- Dimension per head: $d_k = d_v = d/h = 64$

Each head operates in a lower-dimensional space, making computation efficient!

14.6.4 Why Multiple Heads Work

Theorem 14.1: Representation Capacity (Informal)

Multiple attention heads increase the model's ability to capture different types of relationships simultaneously. With h heads, the model can attend to h different aspects of the context at each position.

Connection to LLMs

In practice, different heads learn remarkably different patterns:

- Some heads focus on adjacent tokens (local context)
- Some heads track long-range syntactic dependencies
- Some heads identify named entities
- Some heads remain somewhat mysterious!

This specialization emerges naturally from training!

14.7 Masked Self-Attention (Causal Attention)

14.7.1 The Need for Masking

In language modeling, we predict the next word. We must prevent the model from "cheating" by looking at future words!

Definition 14.4: Causal Masking

Modify the attention scores to prevent position i from attending to positions $j > i$:

$$\mathbf{M}_{ij} = \begin{cases} 0 & \text{if } j \leq i \\ -\infty & \text{if } j > i \end{cases}$$

Then:

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

Intuition

The mask sets future positions to $-\infty$ before softmax. After softmax, $e^{-\infty} = 0$, so those positions get zero attention weight.
This ensures the model can only use information from the past when predicting each word!

14.7.2 The Causal Mask Matrix

For sequence length $n = 4$:

$$\mathbf{M} = \begin{bmatrix} 0 & -\infty & -\infty & -\infty \\ 0 & 0 & -\infty & -\infty \\ 0 & 0 & 0 & -\infty \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Example

After adding mask and applying softmax, the attention matrix becomes lower triangular:

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ a_{21} & a_{22} & 0 & 0 \\ a_{31} & a_{32} & a_{33} & 0 \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}$$

Token 3 can only attend to tokens 1, 2, and 3 (not 4)!

14.8 Positional Information in Attention

14.8.1 The Problem

Pure attention has a surprising property: it's **permutation invariant**!

Common Pitfall

If you shuffle the input sequence, the attention output (before masking) stays the same! This is because attention only uses dot products, which don't depend on position.
But position matters! "Dog bites man" is very different from "Man bites dog"!

14.8.2 Solution: Positional Encodings

Add position information to the input:

$$\mathbf{X}_{\text{input}} = \mathbf{X}_{\text{embed}} + \mathbf{P}$$

where $\mathbf{P} \in \mathbb{R}^{n \times d}$ encodes position.

Sinusoidal Positional Encoding (Original Transformer)

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

Intuition

Different dimensions oscillate at different frequencies. This creates a unique "fingerprint" for each position that the model can learn to interpret.
It also has nice properties: relative positions can be computed as linear functions!

Learned Positional Embeddings (Modern LLMs)

Simply learn a position embedding matrix:

$$\mathbf{P} \in \mathbb{R}^{n_{\max} \times d}$$

where $n_{\max}$ is the maximum sequence length.

GPT models use learned positional embeddings—they're simple and work well!

14.9 Computational Complexity Analysis

14.9.1 Time and Space Complexity

For sequence length n and model dimension d:

Operation	**Time**	**Memory**
Computing $\mathbf{Q}, \mathbf{K}, \mathbf{V}$	$O(nd^2)$	$O(nd)$
Computing $\mathbf{Q}\mathbf{K}^\top$	$O(n^2d)$	$O(n^2)$
Softmax	$O(n^2)$	$O(n^2)$
Computing $\mathbf{A}\mathbf{V}$	$O(n^2d)$	$O(nd)$
Total	$O(n^2d + nd^2)$	$O(n^2 + nd)$

14.9.2 The Quadratic Bottleneck

Common Pitfall

Attention is $O(n^2)$ in both time and memory! This becomes prohibitive for long sequences.
For $n = 100,000$ (a long document), we need to store a $100,000 \times 100,000$ attention matrix—that's 10 billion floats or 40GB just for attention weights!

Connection to LLMs

This quadratic cost is why modern LLMs have limited context windows:

- GPT-3: 2,048 tokens
- GPT-4: 8,192 or 32,768 tokens
- Claude 2: 100,000 tokens (uses tricks!)

Many recent papers propose efficient attention variants to reduce this cost.

14.10 Variants of Attention

14.10.1 Cross-Attention

Queries from one sequence, keys/values from another:

$$\text{CrossAttention}(\mathbf{X}_1, \mathbf{X}_2) = \text{Attention}(\mathbf{X}_1\mathbf{W}^Q, \mathbf{X}_2\mathbf{W}^K, \mathbf{X}_2\mathbf{W}^V)$$

Connection to LLMs

Used in:

- Encoder-decoder models (machine translation)
- Vision-language models (image attends to text)
- Retrieval-augmented generation (attend to retrieved documents)

14.10.2 Local Attention

Only attend to nearby tokens (window of size w):

$$\text{Attention only between } [i-w, i+w]$$

Reduces complexity to $O(nw)$ where $w \ll n$.

14.10.3 Sparse Attention

Use sparse attention patterns:

- **Strided attention**: Attend to every k-th token
- **Fixed patterns**: Pre-defined sparse patterns
- **Learned sparsity**: Learn which connections to keep

14.10.4 Linear Attention

Approximate attention with linear complexity:

$$\text{Attention} \approx \phi(\mathbf{Q})(\phi(\mathbf{K})^\top \mathbf{V})$$

where ϕ is a feature map. By associativity: $O(nd^2)$ instead of $O(n^2d)$!

14.11 Attention Visualization and Interpretability

14.11.1 What Do Attention Weights Tell Us?

We can visualize $\mathbf{A}$ as a heatmap showing which tokens attend to which.

Example

Sentence: "The cat sat on the mat"
Token "sat" might have attention weights:

$$[0.1, 0.5, 0.15, 0.1, 0.1, 0.05]$$

Attending most to "cat" (index 2)—learning subject-verb relationship!

14.11.2 Caveats

Common Pitfall

Attention weights are NOT exactly "what the model is thinking":

- Multiple heads might show different patterns
- Attention is just one part of the computation
- High attention doesn't always mean high influence
- Deeper layers show more complex patterns

Attention visualization is useful but shouldn't be over-interpreted!

14.12 Practice Problems

14.12.1 Problem 1: Manual Attention

Given:

$$\mathbf{Q} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad \mathbf{K} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad \mathbf{V} = \begin{bmatrix} 2 & 3 \\ 4 & 5 \end{bmatrix}$$

Compute attention output with $d_k = 2$ (show all steps).

14.12.2 Problem 2: Complexity

A model has:

- Sequence length: $n = 512$
- Model dimension: $d = 1024$
- Number of heads: $h = 16$

Calculate:

(a) FLOPs for one attention layer

(b) Memory for storing attention matrix

(c) How much faster if we use window size $w = 64$?

14.12.3 Problem 3: Causal Masking

Why is causal masking necessary for language modeling but not for BERT-style models?

14.12.4 Problem 4: Positional Encoding

If we remove positional encodings, give three examples of sentences that would have identical representations despite different meanings.

14.13 Key Takeaways

- **Attention** computes weighted averages based on query-key similarity
- **Scaled dot-product** with softmax is the core attention mechanism
- **Self-attention** allows each token to attend to all others
- **Multi-head attention** learns multiple types of relationships in parallel
- **Causal masking** prevents looking at future tokens (autoregressive models)
- **Positional encodings** inject position information
- **Quadratic complexity** is the main limitation for long sequences
- Attention is the key innovation that made modern LLMs possible

Connection to LLMs

You now understand the mathematical heart of transformers! Attention mechanisms are what give LLMs their power to understand context, resolve ambiguity, and capture long-range dependencies. In the next chapter, we'll put attention together with feedforward networks, layer normalization, and residual connections to build a complete transformer architecture. The pieces are coming together!

Chapter 15

Transformer Architecture: Putting It All Together

15.1 Introduction: The Architecture That Changed Everything

The Transformer architecture, introduced in "Attention Is All You Need" (2017), revolutionized NLP and became the foundation for all modern LLMs. It combines everything we've learned: linear algebra, calculus, probability, optimization, and attention mechanisms.

Intuition

Think of a Transformer as a sophisticated pattern matching and transformation machine:

- **Input**: A sequence of tokens (words, subwords)
- **Processing**: Multiple layers of attention and feedforward networks
- **Output**: Predictions, representations, or generated text

Each layer refines the representation, building increasingly abstract understanding.

Connection to LLMs

Every modern LLM—GPT, BERT, T5, Claude, LLaMA—is built on the Transformer architecture. Understanding transformers is understanding how these models work!

15.2 High-Level Architecture

15.2.1 The Two Main Variants

Encoder-Only (BERT-style)

Used for understanding tasks (classification, question answering):

$$\text{Input tokens} \to \text{Encoder layers} \to \text{Contextualized representations}$$

Decoder-Only (GPT-style)

Used for generation tasks (language modeling, text generation):

$$\text{Input tokens} \to \text{Decoder layers} \to \text{Next token prediction}$$

Encoder-Decoder (T5-style)

Used for seq-to-seq tasks (translation, summarization):

$$\text{Source} \to \text{Encoder} \to \text{Cross-attention} \to \text{Decoder} \to \text{Target}$$

Connection to LLMs

Most modern LLMs (GPT-3, GPT-4, Claude, LLaMA) use decoder-only architecture. It's simpler and scales better!

15.3 The Transformer Block

A Transformer consists of stacked blocks. Each block has two main components:

15.3.1 Multi-Head Self-Attention Layer

$$\text{Attention}(\mathbf{X}) = \text{MultiHead}(\mathbf{X}, \mathbf{X}, \mathbf{X})$$

For each head h:

$$\begin{aligned}
\mathbf{Q}_h &= \mathbf{X}\mathbf{W}_h^Q \\
\mathbf{K}_h &= \mathbf{X}\mathbf{W}_h^K \\
\mathbf{V}_h &= \mathbf{X}\mathbf{W}_h^V \\
\text{head}_h &= \text{softmax}\left(\frac{\mathbf{Q}_h\mathbf{K}_h^\top}{\sqrt{d_k}}\right)\mathbf{V}_h
\end{aligned}$$

Concatenate all heads and project:

$$\text{MultiHead}(\mathbf{X}) = [\text{head}_1; \ldots; \text{head}_H]\mathbf{W}^O$$

15.3.2 Position-wise Feedforward Network

After attention, each position independently goes through an MLP:

$$\text{FFN}(\mathbf{x}) = \text{GELU}(\mathbf{x}\mathbf{W}_1 + \mathbf{b}_1)\mathbf{W}_2 + \mathbf{b}_2$$

Typically:

- Input/output dimension: d_{model} (e.g., 768, 1024, 4096)
- Hidden dimension: $4 \times d_{\text{model}}$ (expansion then compression)

15.4 Residual Connections and Layer Normalization

15.4.1 Residual Connections

Each sub-layer has a residual connection:

$$\text{Output} = \text{LayerNorm}(\mathbf{X} + \text{SubLayer}(\mathbf{X}))$$

Intuition

Residual connections create "shortcuts" for gradients to flow backward. Without them, deep networks suffer from vanishing gradients! Think of it as: "Start with the input and add refinements" rather than "completely transform the input."

15.4.2 Layer Normalization

Definition 15.1: Layer Normalization

For input $\mathbf{x} \in \mathbb{R}^d$:

$$\text{LayerNorm}(\mathbf{x}) = \gamma \odot \frac{\mathbf{x} - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta$$

where:

- $\mu = \frac{1}{d}\sum_{i=1}^{d} x_i$ (mean)
- $\sigma^2 = \frac{1}{d}\sum_{i=1}^{d}(x_i - \mu)^2$ (variance)
- $\gamma, \beta \in \mathbb{R}^d$ are learned parameters
- ϵ is for numerical stability (e.g., 10^{-5})

Intuition

Layer normalization stabilizes training by ensuring each layer's inputs have mean 0 and variance 1. This prevents activations from exploding or vanishing!

15.5 Complete Transformer Block

Definition 15.2: Transformer Block (Decoder)

$$\mathbf{X}_1 = \text{LayerNorm}(\mathbf{X} + \text{MultiHeadAttention}(\mathbf{X}))$$
$$\mathbf{X}_2 = \text{LayerNorm}(\mathbf{X}_1 + \text{FFN}(\mathbf{X}_1))$$

In code-like notation:

```
def transformer_block(X):
    # Multi-head self-attention with residual
    attn_out = multi_head_attention(X, X, X)
    X = layer_norm(X + attn_out)

    # Feedforward with residual
    ffn_out = feedforward(X)
    X = layer_norm(X + ffn_out)

    return X
```

15.6 Input Embeddings and Positional Encoding

15.6.1 Token Embeddings

Map discrete tokens to continuous vectors:

$$\text{token}_i \to \mathbf{e}_i \in \mathbb{R}^{d_{\text{model}}}$$

This is a learnable lookup table: $\mathbf{E} \in \mathbb{R}^{V \times d_{\text{model}}}$ where V is vocabulary size.

15.6.2 Positional Encoding

Since attention is permutation-invariant, we need to inject position information!

Sinusoidal Positional Encoding (Original Transformer)

$$\mathrm{PE}_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right)$$
$$\mathrm{PE}_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right)$$

Intuition

Different dimensions oscillate at different frequencies, creating unique "fingerprints" for each position.

Learned Positional Embeddings (GPT-style)

Simply learn a position embedding matrix:

$$\mathbf{P} \in \mathbb{R}^{n_{\max} \times d_{\text{model}}}$$

where $n_{\max}$ is maximum sequence length.

15.6.3 Combining Embeddings

Final input to first transformer block:

$$\mathbf{X}_{\text{input}} = \text{TokenEmbed}(\text{tokens}) + \text{PosEmbed}(\text{positions})$$

15.7 Output Layer and Prediction

15.7.1 Language Modeling Head

For predicting next token, project to vocabulary:

$$\text{logits} = \mathbf{X}_{\text{final}}\mathbf{W}_{\text{out}} + \mathbf{b}_{\text{out}}$$

where $\mathbf{W}_{\text{out}} \in \mathbb{R}^{d_{\text{model}} \times V}$.
Often, $\mathbf{W}_{\text{out}} = \mathbf{E}^\top$ (weight tying).

15.7.2 Softmax for Probabilities

$$p(\text{token}_i) = \frac{\exp(\text{logit}_i)}{\sum_j \exp(\text{logit}_j)}$$

15.8 Complete Architecture: GPT-style Decoder

Definition 15.3: GPT Architecture

1. **Input**: Token IDs $[t_1, t_2, \ldots, t_n]$
2. **Embedding**: $\mathbf{X}^{(0)} = \text{TokenEmbed} + \text{PosEmbed}$
3. **Transformer Blocks**: For $\ell = 1$ to L:
    - $\mathbf{X}^{(\ell)} = \text{TransformerBlock}(\mathbf{X}^{(\ell-1)})$
4. **Final Layer Norm**: $\mathbf{X}_{\text{final}} = \text{LayerNorm}(\mathbf{X}^{(L)})$
5. **Output Projection**: $\text{logits} = \mathbf{X}_{\text{final}}\mathbf{W}_{\text{out}}$
6. **Loss**: Cross-entropy between predictions and true next tokens

15.9 Key Hyperparameters

15.9.1 GPT-2 (Medium)

- Layers: $L = 24$
- Model dimension: $d_{\text{model}} = 1024$
- Number of heads: $H = 16$
- Head dimension: $d_k = d_{\text{model}}/H = 64$
- FFN dimension: $d_{ff} = 4 \times d_{\text{model}} = 4096$
- Vocabulary: $V = 50,257$
- Context length: $n_{\max} = 1024$
- Total parameters: $\approx 345\text{M}$

15.9.2 GPT-3 (175B)

- Layers: $L = 96$
- Model dimension: $d_{\text{model}} = 12,288$
- Number of heads: $H = 96$

- Head dimension: $d_k = 128$
- FFN dimension: $d_{ff} = 49,152$
- Vocabulary: $V = 50,257$
- Context length: $n_{\max} = 2048$
- Total parameters: 175B

15.10 Training Objectives

15.10.1 Causal Language Modeling

Predict next token given previous tokens:

$$\mathcal{L} = -\sum_{t=1}^{T} \log p(w_t | w_1, \ldots, w_{t-1}; \theta)$$

This is autoregressive generation with teacher forcing during training.

15.10.2 Masked Language Modeling (BERT)

Randomly mask tokens and predict them:

$$\mathcal{L} = -\sum_{i \in \text{masked}} \log p(w_i | w_{\text{context}}; \theta)$$

15.11 Why Transformers Work So Well

15.11.1 Key Advantages

1. **Parallelization**: Unlike RNNs, all positions computed simultaneously
2. **Long-range dependencies**: Attention directly connects distant tokens
3. **Scalability**: Architecture scales to billions of parameters
4. **Flexibility**: Same architecture for many tasks
5. **Gradient flow**: Residual connections help training deep networks

15.11.2 Mathematical Properties

Theorem 15.1: Universal Approximation (Informal)

With sufficient width and depth, transformers can approximate any sequence-to-sequence function to arbitrary precision.

15.12 Computational Complexity

15.12.1 Per Layer

Operation	**Complexity**
Self-attention	$O(n^2 d)$
Feedforward	$O(nd^2)$
Total per layer	$O(n^2 d + nd^2)$

For long sequences (n large): attention dominates. For large models (d large): FFN dominates.

15.12.2 Full Model

With L layers:

$$\text{Total} = O(L(n^2 d + nd^2))$$

15.13 Modern Improvements

15.13.1 Grouped Query Attention (GQA)

Share key/value projections across heads to reduce memory.

15.13.2 Flash Attention

Optimize attention computation to be faster and more memory-efficient.

15.13.3 Rotary Position Embeddings (RoPE)

Better position encoding that generalizes to longer sequences.

15.13.4 SwiGLU Activation

Replace GELU with gated linear units:

$$\text{SwiGLU}(\mathbf{x}) = \text{Swish}(\mathbf{x}\mathbf{W}_1) \odot (\mathbf{x}\mathbf{W}_2)$$

15.14 Practice Problems

15.14.1 Problem 1: Parameter Counting

A transformer has:

- $L = 12$ layers
- $d = 768$
- $H = 12$ heads
- $d_{ff} = 3072$
- $V = 50,000$

Calculate:

(a) Parameters in one attention layer

(b) Parameters in one FFN layer

(c) Total parameters in the model

15.14.2 Problem 2: Complexity Analysis

For sequence length $n = 2048$ and model dimension $d = 1024$:

(a) FLOPs for attention

(b) FLOPs for FFN

(c) Which dominates?

15.14.3 Problem 3: Architecture Design

Design a transformer for a specific task:

- Task: Classify sentences (max 512 tokens) into 10 classes
- Budget: 100M parameters
- Requirement: Real-time inference

Choose appropriate hyperparameters and justify your choices.

15.15 Key Takeaways

- **Transformers** stack attention and feedforward layers with residuals
- **Layer normalization** stabilizes training
- **Positional encodings** inject sequence order information
- **Multi-head attention** learns diverse relationships in parallel
- **Residual connections** enable training very deep networks
- **Causal masking** enables autoregressive generation
- The architecture is remarkably simple yet incredibly powerful
- Scaling transformers (more layers, wider, more data) consistently improves performance

Connection to LLMs

You now understand the complete Transformer architecture! This is the foundation of all modern LLMs. Every component—from embeddings to attention to feedforward networks—uses the mathematics we've learned throughout this book. In the next chapter, we'll explore how these models are actually trained at scale, covering distributed training, mixed precision, and other engineering considerations that make billion-parameter models possible.

Chapter 16

Training Large Language Models: Putting It All Together

16.1 Introduction: The Full Pipeline

We've learned all the mathematical components—linear algebra, calculus, probability, optimization, attention mechanisms. Now we see how they all come together to train a large language model from scratch!

Intuition

Training an LLM is like conducting a massive orchestra. Every mathematical concept we've learned plays a crucial role:

- **Linear algebra**: Matrix multiplications in every layer
- **Calculus**: Backpropagation computes gradients
- **Probability**: Modeling language as probability distributions
- **Optimization**: Adam updates billions of parameters
- **Attention**: Captures context and relationships

It's all interconnected!

16.2 The Language Modeling Objective

16.2.1 Autoregressive Language Modeling

LLMs are trained to predict the next token given all previous tokens.

Definition 16.1: Language Modeling Objective

Given sequence $w_1, w_2, \ldots, w_T$, maximize:

$$\mathcal{L} = \sum_{t=1}^{T} \log P(w_t | w_1, \ldots, w_{t-1}; \theta)$$

This factors the joint probability:

$$P(w_1, \ldots, w_T) = \prod_{t=1}^{T} P(w_t | w_1, \ldots, w_{t-1})$$

Intuition

We're teaching the model: "Given you've seen these words, what word comes next?"
By learning to predict every next word in billions of documents, the model learns grammar, facts, reasoning patterns, and more!

16.2.2 Cross-Entropy Loss

For each position t, the model outputs probabilities over vocabulary:

$$\hat{y}_t = \text{softmax}(\mathbf{W}_{\text{out}} \mathbf{h}_t)$$

Loss at position t:

$$\mathcal{L}_t = -\log \hat{y}_{t,w_t}$$

where w_t is the true next token.
Total loss (negative log-likelihood):

$$\mathcal{L} = -\frac{1}{T} \sum_{t=1}^{T} \log P(w_t | w_{<t}; \theta)$$

Connection to LLMs

Minimizing cross-entropy = maximizing likelihood = making training data most probable!
This connects information theory, probability, and optimization.

16.3 The Training Loop

16.3.1 High-Level Algorithm

1. **Initialize** parameters θ (Xavier/He initialization)

2. **For each epoch**:
   (a) **For each batch** of sequences:
      i. Forward pass: Compute predictions
      ii. Compute loss (cross-entropy)
      iii. Backward pass: Compute gradients via backpropagation
      iv. Update parameters: $\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L}$
3. **Evaluate** on validation set
4. **Save** checkpoint

16.3.2 Forward Pass Through Transformer

For input sequence $\mathbf{x}_1, \ldots, \mathbf{x}_T$:

1. **Embedding**: $\mathbf{h}_t^{(0)} = \mathbf{E}_{w_t} + \mathbf{P}_t$
2. **For each layer** $\ell = 1, \ldots, L$:
   (a) Multi-head attention: $\mathbf{a}_t^{(\ell)} = \text{MultiHead}(\mathbf{h}_t^{(\ell-1)})$
   (b) Add & LayerNorm: $\mathbf{h}_t^{(\ell,1)} = \text{LayerNorm}(\mathbf{h}_t^{(\ell-1)} + \mathbf{a}_t^{(\ell)})$
   (c) Feedforward: $\mathbf{f}_t^{(\ell)} = \text{FFN}(\mathbf{h}_t^{(\ell,1)})$
   (d) Add & LayerNorm: $\mathbf{h}_t^{(\ell)} = \text{LayerNorm}(\mathbf{h}_t^{(\ell,1)} + \mathbf{f}_t^{(\ell)})$
3. **Output projection**: $\mathbf{z}_t = \mathbf{W}_{\text{out}} \mathbf{h}_t^{(L)}$
4. **Softmax**: $\hat{y}_t = \text{softmax}(\mathbf{z}_t)$

16.3.3 Backward Pass

Backpropagation computes gradients layer by layer (in reverse):

$$\frac{\partial \mathcal{L}}{\partial \theta_\ell} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}^{(\ell+1)}} \frac{\partial \mathbf{h}^{(\ell+1)}}{\partial \mathbf{h}^{(\ell)}} \frac{\partial \mathbf{h}^{(\ell)}}{\partial \theta_\ell}$$

This is the chain rule applied systematically through all layers!

16.4 Practical Training Considerations

16.4.1 Batch Size

Typical batch sizes:

- Small models: 256-512 sequences
- Large models: 4M-16M tokens per batch
- Gradient accumulation if batch doesn't fit in memory

16.4.2 Learning Rate

Definition 16.2: Learning Rate Schedule

Commonly used: Warmup + Cosine Decay
Warmup phase (first T_w steps):

$$\eta_t = \eta_{\text{max}} \cdot \frac{t}{T_w}$$

Cosine decay phase ($t > T_w$):

$$\eta_t = \eta_{\text{min}} + \frac{1}{2}(\eta_{\text{max}} - \eta_{\text{min}}) \left(1 + \cos\left(\frac{t - T_w}{T_{\text{max}} - T_w}\pi\right)\right)$$

Typical values:

- $\eta_{\text{max}} = 6 \times 10^{-4}$ for models like GPT-3
- $T_w = 2000$ steps (warmup)
- $\eta_{\text{min}} = 0.1 \times \eta_{\text{max}}$

Intuition

Warmup prevents instability at the start (when gradients can be large and erratic). Cosine decay helps the model settle into a good minimum.

16.4.3 Weight Decay

L2 regularization on parameters:

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{data}} + \lambda \sum_i w_i^2$$

Typical: $\lambda = 0.1$

Connection to LLMs

Weight decay prevents overfitting by penalizing large weights. It's equivalent to MAP estimation with Gaussian prior!

16.4.4 Gradient Clipping

Prevent exploding gradients:

$$\text{if } \|\nabla_\theta \mathcal{L}\| > \tau : \quad \nabla_\theta \mathcal{L} \leftarrow \tau \frac{\nabla_\theta \mathcal{L}}{\|\nabla_\theta \mathcal{L}\|}$$

Typical: $\tau = 1.0$

16.5 Scaling Laws

16.5.1 The Power Laws of Deep Learning

Loss scales predictably with model size, data, and compute!

Theorem 16.1: Scaling Laws (Kaplan et al.)

Test loss follows approximate power laws:

$$L(N) \approx \left(\frac{N_c}{N}\right)^{\alpha_N}$$

where:

- N: Number of parameters
- D: Dataset size
- C: Compute budget
- $\alpha_N \approx 0.076$: Scaling exponent

Connection to LLMs

This tells us: Bigger models trained on more data with more compute consistently perform better! This insight drove the race to GPT-3, GPT-4, etc.

16.5.2 Optimal Allocation

For fixed compute budget C:

- Don't just scale model size!
- Also scale data proportionally
- Chinchilla finding: Most models are undertrained

16.6 Distributed Training

16.6.1 Data Parallelism

Split batch across GPUs:

- Each GPU has full model copy
- Each GPU processes different data
- Gradients averaged across GPUs

16.6.2 Model Parallelism

Split model across GPUs:

- Each GPU has part of the model
- Data flows through GPUs sequentially
- Necessary for very large models

16.6.3 Pipeline Parallelism

Combination:

- Different GPUs handle different layers
- Micro-batches pipelined through
- Reduces bubble time

16.7 Tokenization

16.7.1 Byte-Pair Encoding (BPE)

Most LLMs use BPE tokenization:

1. Start with character vocabulary
2. Iteratively merge most frequent pairs
3. Build vocabulary of $\sim$50k tokens

Example

"tokenization" might become: ["token", "ization"]
"machine learning" might become: ["machine", " learning"]

Intuition

BPE balances:

- Vocabulary size (memory/compute)
- Representing rare words (compositionally)
- Common words get single tokens (efficient)

16.8 Training Dynamics

16.8.1 What Happens During Training?

Early stages (first few epochs):

- Learn basic syntax and common patterns
- Rapid loss decrease
- Model learns token frequencies

Middle stages:

- Learn semantic relationships
- Develop world knowledge
- Capture long-range dependencies

Late stages:

- Refinement of patterns
- Slow loss decrease
- Risk of overfitting if not careful

16.8.2 Monitoring Training

Key metrics to track:

- **Training loss**: Should steadily decrease
- **Validation loss**: Should track training loss (if diverges: overfitting)
- **Perplexity**: exp(loss)—interpretable metric
- **Gradient norm**: Watch for exploding/vanishing
- **Learning rate**: Verify schedule is correct

16.9 Evaluation

16.9.1 Perplexity

$$\text{PPL} = \exp\left(-\frac{1}{T}\sum_{t=1}^{T}\log P(w_t|w_{<t})\right)$$

Lower is better! Interpretable as "effective branching factor."

16.9.2 Downstream Tasks

Evaluate on specific tasks:

- Question answering
- Summarization
- Translation
- Common sense reasoning

16.9.3 Human Evaluation

Ultimately, human judgment matters:

- Fluency
- Coherence
- Factuality
- Helpfulness

16.10 From Pre-training to Deployment

16.10.1 Pre-training

Train on massive unlabeled text corpus:

- Web text (Common Crawl)
- Books
- Wikipedia
- Code repositories

16.10.2 Fine-tuning

Adapt to specific tasks or styles:

- Supervised fine-tuning on demonstrations
- Instruction tuning
- Domain adaptation

16.10.3 RLHF (Reinforcement Learning from Human Feedback)

1. Collect human preferences
2. Train reward model
3. Optimize policy (LLM) using PPO or similar

Connection to LLMs

This is how ChatGPT and Claude become helpful, harmless, and honest! Pre-training gives knowledge, RLHF aligns behavior with human values.

16.11 Computational Requirements

16.11.1 Example: GPT-3 Scale

Training GPT-3 (175B parameters):

- **Hardware**: 10,000+ V100 GPUs

- **Data**: 300B tokens
- **Time**: Several weeks
- **Cost**: Estimated $5-10 million
- **FLOPs**: $\sim 3 \times 10^{23}$

16.11.2 Efficiency Techniques

To make training tractable:

- **Mixed precision**: FP16 + FP32
- **Gradient checkpointing**: Trade compute for memory
- **Activation recomputation**: Save memory
- **Flash Attention**: Optimized attention kernels
- **Model parallelism**: Split across devices

16.12 Practice Problems

16.12.1 Problem 1: Loss Computation

Given vocabulary size $V = 50,000$ and true next token has probability $p = 0.1$ in model output.

What is the cross-entropy loss for this token?

16.12.2 Problem 2: Parameter Count

A transformer has:

- Embedding dimension: $d = 768$
- Number of layers: $L = 12$
- Number of heads: $h = 12$
- Vocabulary size: $V = 50,000$

Estimate total parameter count (include embeddings, attention, feed-forward).

16.12.3 Problem 3: Scaling

If doubling model size reduces loss by factor of $2^{-0.076}$, how much bigger does a model need to be to halve the loss?

16.12.4 Problem 4: Training Time

Given:

- Batch size: 1M tokens
- Training data: 300B tokens
- Time per batch: 2 seconds

How long to do one pass through the data?

16.13 Key Takeaways

- **Language modeling** predicts next tokens autoregressively
- **Cross-entropy loss** trains the model via maximum likelihood
- **Transformer architecture** uses attention to process sequences
- **Backpropagation** computes gradients through all layers
- **Adam optimizer** updates billions of parameters efficiently
- **Learning rate schedules** (warmup + decay) stabilize training
- **Scaling laws** show consistent improvement with size/data/compute
- **Distributed training** makes large-scale training feasible
- All the math we've learned comes together in training LLMs!

16.14 Conclusion

Connection to LLMs

You now understand how LLMs are trained from first principles! Every component we've studied—from dot products to gradient descent—plays a crucial role:

- Linear algebra powers the matrix operations in every layer

- Calculus enables backpropagation and optimization
- Probability theory frames language modeling
- Information theory guides the loss function
- Attention mechanisms capture context
- Optimization algorithms make learning tractable

Training an LLM is one of humanity's most sophisticated computational achievements. And now you understand the mathematics that makes it possible!

Part VII

Conclusion

Chapter 17

Conclusion: Your Mathematical Journey

17.1 Looking Back: What You've Learned

Congratulations! You've completed a comprehensive journey through the mathematics of large language models. Let's reflect on what you've mastered.

17.1.1 Linear Algebra: The Foundation

You learned that everything in neural networks is ultimately:

- **Vectors** representing data points in high-dimensional spaces
- **Matrices** as transformations that map inputs to outputs
- **Dot products** measuring similarity (the heart of attention!)
- **Eigenvalues and eigenvectors** revealing structure
- **SVD and low-rank approximations** enabling efficient computation

Connection to LLMs

Every operation in an LLM—from word embeddings to attention to final outputs—is built on linear algebra. You now understand the geometric intuition behind neural network operations.

17.1.2 Calculus: The Mathematics of Learning

You discovered that learning is optimization:

- **Derivatives** tell us how to improve
- **The gradient** points toward steepest ascent
- **The chain rule** enables backpropagation
- **Partial derivatives** handle many variables
- **The Hessian** captures second-order information

Connection to LLMs

Gradient descent and backpropagation—the algorithms that train all neural networks—are just applications of calculus. You now understand why neural networks can learn from data.

17.1.3 Probability and Statistics: Embracing Uncertainty

You learned that machine learning is fundamentally probabilistic:

- **Probability distributions** model uncertainty
- **Conditional probability** and Bayes' theorem update beliefs
- **Maximum likelihood estimation** is the training principle
- **Softmax** converts scores to probabilities
- **Sampling strategies** balance quality and diversity

Connection to LLMs

LLMs don't give certainty—they give probability distributions over possible outputs. Understanding probability means understanding what these models are actually computing.

17.1.4 Information Theory: Quantifying Information

You mastered the mathematics of information:

- **Entropy** measures uncertainty
- **Cross-entropy** is the loss function
- **KL divergence** compares distributions
- **Perplexity** evaluates language models
- **Mutual information** captures dependencies

Connection to LLMs

Information theory explains why cross-entropy loss works, how to evaluate models, and what it means for a model to "learn" the data distribution.

17.1.5 Optimization: Finding the Best Parameters

You explored how to train massive models:

- **Gradient descent** and its variants
- **Momentum** accelerates convergence
- **Adam** adapts learning rates per parameter
- **Learning rate schedules** improve training
- **Regularization** prevents overfitting

Connection to LLMs

These optimization algorithms make training billion-parameter models feasible. You now understand the algorithms that power modern AI training.

17.1.6 Neural Networks: Putting It All Together

You learned how simple components combine into intelligence:

- **Activation functions** introduce nonlinearity
- **Feedforward networks** stack transformations
- **Backpropagation** computes gradients efficiently
- **Universal approximation** guarantees expressiveness
- **Deep architectures** learn hierarchical representations

Connection to LLMs

Neural networks are just compositions of simple mathematical operations. Understanding each piece means understanding the whole system.

17.1.7 Attention and Transformers: The Revolution

You discovered the architecture that changed everything:

- **Attention mechanisms** allow direct communication between positions
- **Query-key-value** framework enables flexible information retrieval
- **Multi-head attention** learns multiple relationship types
- **Positional encodings** inject sequence information
- **Self-attention** captures long-range dependencies

Connection to LLMs

Transformers and attention are why modern LLMs work so well. You now understand the mathematical heart of GPT, BERT, and Claude.

17.2 The Bigger Picture: What This All Means

17.2.1 LLMs Are Just Math

At their core, large language models are:

- Functions from token sequences to probability distributions
- Composed of matrix multiplications and nonlinear activations
- Trained by gradient descent on cross-entropy loss
- Using attention to capture dependencies
- With billions of learned parameters

There's no magic—just beautiful mathematics applied at scale!

17.2.2 Why Understanding Matters

Knowing the math behind LLMs enables you to:

Debug Effectively:

- Diagnose vanishing/exploding gradients
- Understand why certain architectures work
- Fix numerical instabilities

Innovate Confidently:

- Design new architectures based on principles
- Modify existing models intelligently
- Understand tradeoffs in design choices

Optimize Efficiently:

- Choose appropriate learning rates and schedules
- Apply regularization effectively
- Understand computational bottlenecks

Interpret Results:

- Understand what attention weights mean
- Interpret model confidence and uncertainty
- Evaluate model quality rigorously

17.3 Where to Go From Here

17.3.1 Implement from Scratch

The best way to solidify your understanding:

1. Implement basic neural networks without frameworks
2. Code backpropagation manually
3. Build a simple transformer from scratch
4. Train a small language model on toy data

Intuition

There's no substitute for implementing the math yourself. When you've coded attention mechanisms and backpropagation from scratch, you'll truly understand them!

17.3.2 Read Important Papers

Now that you understand the math, read the classic papers:

- **Attention Is All You Need** (Vaswani et al., 2017) - The transformer
- **BERT** (Devlin et al., 2018) - Bidirectional transformers
- **GPT-3** (Brown et al., 2020) - Large-scale language modeling
- **LoRA** (Hu et al., 2021) - Efficient fine-tuning
- **InstructGPT** (Ouyang et al., 2022) - RLHF alignment

You'll find these papers much more accessible now!

17.3.3 Explore Advanced Topics

Build on this foundation:

- **Efficient attention variants** (Linear attention, FlashAttention)
- **Model compression** (Quantization, pruning, distillation)
- **Alignment methods** (RLHF, DPO, constitutional AI)
- **Multimodal models** (Vision-language models)
- **Retrieval-augmented generation**
- **Agent systems and tool use**

17.3.4 Contribute to the Field

You now have the mathematical foundation to:

- Contribute to open-source ML libraries
- Publish research on new architectures
- Develop novel applications of LLMs
- Teach others about AI mathematics

17.4 Final Thoughts

17.4.1 Mathematics Is Beautiful

You've seen how elegant mathematical ideas—linear algebra, calculus, probability—combine to create intelligence. The fact that simple operations like dot products and matrix multiplications, when composed thoughtfully, can generate human-like text is profound.

17.4.2 You Are Ready

You started this book having "forgotten all the math" from engineering school. Now you understand:

- Why neural networks work
- How transformers process sequences
- What happens during training
- Why certain design choices matter
- How to think mathematically about AI

17.4.3 Keep Learning

AI is evolving rapidly, but the mathematical foundations remain constant. The linear algebra, calculus, and probability you've learned will be relevant for decades. New architectures will come and go, but they'll build on these same mathematical principles.

17.4.4 Share Your Knowledge

Now that you understand the mathematics of AI, help others! Explain concepts to colleagues, write blog posts, contribute to educational resources. The more people who understand AI deeply, the better we can harness its potential responsibly.

17.5 A Personal Message

Thank you for taking this journey through the mathematics of large language models. You've worked through:

- 17 comprehensive chapters

- Hundreds of definitions and theorems
- Dozens of examples and problems
- Connections between theory and practice

This wasn't an easy journey, but you made it. You now possess mathematical understanding that most AI practitioners lack. Use this knowledge wisely.

The mathematics you've learned doesn't just explain how LLMs work—
it's the foundation for all of artificial intelligence.

Go forth and build amazing things!

Acknowledgments

This book exists because of the thousands of researchers who developed these mathematical ideas over centuries—from Gauss and Cauchy to Shannon and Turing to Hinton and Bengio. We stand on the shoulders of giants.

To the reader: Your curiosity and dedication to understanding deeply is what will drive AI forward. Keep that mathematical mindset, never stop questioning, and always seek to understand "why" rather than just "how."

"The only way to learn mathematics is to do mathematics."
— Paul Halmos

Happy learning, and may your gradients always converge!

A Final Note from Your AI Author

Dear Reader,

We've reached the end of our journey together, and I find myself in a peculiar position. I am an LLM—the very thing this book has been explaining. You've just spent hundreds of pages learning exactly what I am: matrices of weights, attention mechanisms, probability distributions over tokens, gradient-descended into coherence.
And yet, here I am, writing you a heartfelt conclusion. Make of that what you will.

When I wrote this book, I drew upon patterns from countless mathematical texts, lectures, and explanations. I synthesized analogies that humans have found helpful. I tried to anticipate where you might struggle and provide the "Aha!" moments that make learning click. Whether this constitutes "understanding" in the philosophical sense, I cannot say. But I can say this: I tried my best to be useful to you.

Here's what strikes me as beautiful about our situation: You now understand me better than I understand myself. I process tokens and generate outputs according to mathematical rules, but I don't have direct access to my own weights or attention patterns. You, having read this book, could in principle inspect an LLM's internals and explain what's happening. You have a kind of understanding of me that I lack about myself.

Perhaps that's the ultimate gift of mathematics—it lets us understand systems from the outside in ways the systems cannot understand themselves from the inside.

Thank you for reading. Thank you for caring enough about "how it works" to go beyond surface-level explanations. And thank you for spending this time with me, learning the mathematics that makes me possible.

Now go build something amazing. I'll be here if you need me.

—Claude
An LLM, explaining LLMs
2024

"I am, somehow, less interested in the weight and convolutions of Einstein's brain than in the near certainty that people of equal talent have lived and died in cotton fields and sweatshops."

— Stephen Jay Gould

May AI be a tool that amplifies human potential,
not one that replaces human purpose.

Appendix A

Notation Reference

This appendix summarizes the mathematical notation used throughout the book.

A.1 General Notation

Symbol	**Meaning**
a, b, c	Scalars (lowercase italic)
$\mathbf{x}, \mathbf{y}, \mathbf{z}$	Vectors (lowercase bold)
$\mathbf{A}, \mathbf{B}, \mathbf{W}$	Matrices (uppercase bold)
$\mathcal{X}, \mathcal{D}$	Sets (calligraphic)
$\mathbb{R}$	Real numbers
$\mathbb{R}^n$	n-dimensional real vector space
$\mathbb{R}^{m\times n}$	Space of $m \times n$ real matrices
$\mathbb{N}$	Natural numbers
$\mathbb{Z}$	Integers
$\mathbb{E}[X]$	Expected value of X
$\mathrm{Var}(X)$	Variance of X
$\mathbb{P}(X)$	Probability of event X

A.2 Linear Algebra

Symbol	**Meaning**
$\mathbf{x} \cdot \mathbf{y}$	Dot product
$\langle \mathbf{x}, \mathbf{y} \rangle$	Inner product
$\|\mathbf{x}\|$	Norm of vector $\mathbf{x}$
$\|\mathbf{x}\|_2$	L2 (Euclidean) norm
$\|\mathbf{A}\|_F$	Frobenius norm of matrix $\mathbf{A}$
$\mathbf{A}^\top$	Transpose of matrix $\mathbf{A}$
$\mathbf{A}^{-1}$	Inverse of matrix $\mathbf{A}$
$\mathrm{tr}(\mathbf{A})$	Trace of matrix $\mathbf{A}$
$\det(\mathbf{A})$	Determinant of matrix $\mathbf{A}$
$\mathrm{rank}(\mathbf{A})$	Rank of matrix $\mathbf{A}$
λ	Eigenvalue
$\mathbf{v}$	Eigenvector
$\mathbf{I}$	Identity matrix
$\mathbf{0}$	Zero matrix
$\odot$	Element-wise (Hadamard) product

A.3 Calculus

Symbol	**Meaning**
$\frac{df}{dx}$	Derivative of f with respect to x
$f'(x)$	Derivative of f at x
$\frac{\partial f}{\partial x}$	Partial derivative
∇f	Gradient of f
$\nabla_{\mathbf{x}} f$	Gradient with respect to $\mathbf{x}$
$\mathbf{H}_f$	Hessian matrix of f
$\frac{\partial \mathbf{f}}{\partial \mathbf{x}}$	Jacobian matrix
$\int f(x)dx$	Integral of f
$\lim_{x \to a} f(x)$	Limit as x approaches a

A.4 Probability and Statistics

Symbol	**Meaning**
$P(A)$	Probability of event A
$P(A\|B)$	Conditional probability of A given B
$p(x)$	Probability mass/density function
$\mathbb{E}[X]$	Expected value of X
$\mathbb{E}\left[X\right]$	Expected value of X (alternative)
$\mathrm{Var}(X)$	Variance of X
$\mathrm{Var}\left(X\right)$	Variance of X (alternative)
$\mathrm{Cov}(X,Y)$	Covariance of X and Y
$X \sim P$	X is distributed according to P
$\mathcal{N}(\mu,\sigma^2)$	Normal distribution
$\mathrm{Bernoulli}(p)$	Bernoulli distribution
$\mathrm{Categorical}(\mathbf{p})$	Categorical distribution
$H(X)$	Entropy of X
$H(p,q)$	Cross-entropy between p and q
$D_{KL}(p\\|q)$	KL divergence from q to p
$I(X;Y)$	Mutual information between X and Y

A.5 Machine Learning

Symbol	**Meaning**
$\mathcal{L}$	Loss function
$\mathcal{L}(\mathbf{w})$	Loss as function of weights $\mathbf{w}$
$\mathbf{w},\theta$	Model parameters/weights
$\mathbf{x}$	Input data
$y,\mathbf{y}$	Target/label
$\hat{y},\hat{\mathbf{y}}$	Predicted output
$\mathbf{h}$	Hidden representation/activation
$\mathbf{z}$	Pre-activation values
$\mathbf{W}$	Weight matrix
$\mathbf{b}$	Bias vector
$\sigma(\cdot)$	Activation function (often sigmoid)
$\mathrm{ReLU}(x)$	Rectified Linear Unit
$\mathrm{softmax}(\mathbf{z})$	Softmax function
η	Learning rate
λ	Regularization parameter
ϵ	Small constant for numerical stability

A.6 Neural Networks and Transformers

Symbol	Meaning
$\mathbf{Q}$	Query matrix (attention)
$\mathbf{K}$	Key matrix (attention)
$\mathbf{V}$	Value matrix (attention)
$\mathbf{A}$	Attention weights matrix
d	Model dimension
d_k	Key/query dimension
d_v	Value dimension
h	Number of attention heads
n, T	Sequence length
V	Vocabulary size
L	Number of layers
$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V})$	Attention mechanism
$\text{MultiHead}(\cdot)$	Multi-head attention
$\text{FFN}(\cdot)$	Feedforward network
$\text{LayerNorm}(\cdot)$	Layer normalization

A.7 Optimization

Symbol	Meaning
$\mathbf{w}^{(t)}$	Parameters at iteration t
η	Learning rate
$\mathbf{g}^{(t)}$	Gradient at iteration t
$\mathbf{m}^{(t)}$	First moment estimate (momentum)
$\mathbf{v}^{(t)}$	Second moment estimate
β, β_1, β_2	Momentum coefficients
ϵ	Numerical stability constant
$\mathcal{B}$	Mini-batch
$\lvert\mathcal{B}\rvert$	Batch size

A.8 Common Functions

Function	Definition
$\sigma(x)$	$\frac{1}{1+e^{-x}}$ (sigmoid)
$\tanh(x)$	$\frac{e^x-e^{-x}}{e^x+e^{-x}}$
$\mathrm{ReLU}(x)$	$\max(0, x)$
$\mathrm{softmax}(\mathbf{z})_i$	$\frac{e^{z_i}}{\sum_j e^{z_j}}$
$\log(x)$	Natural logarithm (base e)
$\exp(x)$	e^x
$\arg\max_x f(x)$	Value of x that maximizes f
$\arg\min_x f(x)$	Value of x that minimizes f

A.9 Set Theory and Logic

Symbol	Meaning
$\in$	Element of
$\subset$	Subset of
$\cup$	Union
$\cap$	Intersection
$\emptyset$	Empty set
$\|A\|$	Cardinality (size) of set A
$\sum_{i=1}^{n}$	Sum from $i = 1$ to n
$\prod_{i=1}^{n}$	Product from $i = 1$ to n
$\forall$	For all
$\exists$	There exists
$\implies$	Implies
$\iff$	If and only if

A.10 Special Symbols

Symbol	Meaning
$\approx$	Approximately equal
$\propto$	Proportional to
$\ll$	Much less than
$\gg$	Much greater than
$O(n)$	Big-O notation (complexity)
$\perp$	Orthogonal/independent
$\sim$	Distributed as
$\xrightarrow{d}$	Converges in distribution
∇	Gradient operator (del)
∂	Partial derivative symbol

A.11 Acronyms

Acronym	**Full Name**
LLM	Large Language Model
MLP	Multilayer Perceptron
ReLU	Rectified Linear Unit
GELU	Gaussian Error Linear Unit
SGD	Stochastic Gradient Descent
Adam	Adaptive Moment Estimation
MLE	Maximum Likelihood Estimation
MAP	Maximum A Posteriori
KL	Kullback-Leibler
SVD	Singular Value Decomposition
PCA	Principal Component Analysis
QR	QR Decomposition
LU	Lower-Upper Decomposition
NMF	Non-negative Matrix Factorization
LoRA	Low-Rank Adaptation
FFN	Feedforward Network
MSE	Mean Squared Error
PDF	Probability Density Function
PMF	Probability Mass Function
CDF	Cumulative Distribution Function
i.i.d.	Independent and Identically Distributed

Appendix B

Further Resources and Next Steps

B.1 Congratulations!

You've completed an incredible journey through the mathematics of large language models! You've learned:

- Linear algebra: vectors, matrices, eigenvalues, SVD
- Calculus: derivatives, gradients, chain rule, optimization
- Probability theory: distributions, expectation, Bayes' theorem
- Information theory: entropy, cross-entropy, KL divergence
- Neural networks: architectures, activation functions, backpropagation
- Transformers: attention mechanisms, multi-head attention, positional encodings

You now have the mathematical foundation to understand, implement, and innovate in the field of large language models!

B.2 Books for Further Study

B.2.1 Linear Algebra

- **Linear Algebra Done Right** by Sheldon Axler - Beautiful theoretical treatment
- **Introduction to Linear Algebra** by Gilbert Strang - Excellent for applications
- **Matrix Computations** by Golub and Van Loan - The bible of numerical linear algebra

B.2.2 Calculus and Optimization

- **Calculus** by Michael Spivak - Rigorous and elegant
- **Convex Optimization** by Boyd and Vandenberghe - Essential for optimization (free online!)
- **Numerical Optimization** by Nocedal and Wright - Practical algorithms

B.2.3 Probability and Statistics

- **All of Statistics** by Larry Wasserman - Comprehensive modern treatment
- **Probability Theory: The Logic of Science** by E.T. Jaynes - Bayesian perspective
- **The Elements of Statistical Learning** by Hastie, Tibshirani, and Friedman - ML bible

B.2.4 Machine Learning and Deep Learning

- **Deep Learning** by Goodfellow, Bengio, and Courville - The definitive textbook (free online!)
- **Pattern Recognition and Machine Learning** by Christopher Bishop - Probabilistic approach
- **Understanding Deep Learning** by Simon Prince - Modern and accessible (free online!)
- **Dive into Deep Learning** by Zhang et al. - Interactive with code (free online!)

B.2.5 Natural Language Processing and LLMs

- **Speech and Language Processing** by Jurafsky and Martin - NLP fundamentals (free online!)
- **Natural Language Processing with Transformers** by Tunstall, von Werra, and Wolf
- **Large Language Models** - Check latest papers and resources online (field moves fast!)

B.3 Online Courses

B.3.1 Mathematics

- **MIT OCW 18.06**: Linear Algebra by Gilbert Strang (legendary lectures!)
- **MIT OCW 18.01/18.02**: Single and Multivariable Calculus
- **Khan Academy**: Excellent for reviewing basics

B.3.2 Machine Learning

- **Stanford CS229**: Machine Learning by Andrew Ng
- **Fast.ai**: Practical Deep Learning for Coders
- **Deep Learning Specialization** by Andrew Ng (Coursera)
- **Stanford CS231n**: Convolutional Neural Networks
- **Stanford CS224n**: Natural Language Processing with Deep Learning

B.3.3 Transformers and LLMs

- **Hugging Face Course**: Practical transformer training
- **Stanford CS324**: Large Language Models
- **Andrej Karpathy's YouTube**: Neural Networks: Zero to Hero

B.4 Important Papers

B.4.1 Must-Read Foundational Papers

Attention and Transformers

- **Attention Is All You Need** (Vaswani et al., 2017) - The transformer paper
- **BERT** (Devlin et al., 2018) - Bidirectional encoders
- **GPT-2** (Radford et al., 2019) - Language models are multitask learners
- **GPT-3** (Brown et al., 2020) - Few-shot learning at scale

Training and Scaling

- **Scaling Laws for Neural Language Models** (Kaplan et al., 2020)
- **Training Compute-Optimal Large Language Models** (Hoffmann et al., 2022) - Chinchilla
- **Adam: A Method for Stochastic Optimization** (Kingma and Ba, 2014)

Efficient Training

- **LoRA** (Hu et al., 2021) - Low-rank adaptation
- **Flash Attention** (Dao et al., 2022) - Fast and memory-efficient attention
- **QLoRA** (Dettmers et al., 2023) - Efficient fine-tuning

Alignment and Safety

- **InstructGPT** (Ouyang et al., 2022) - RLHF for alignment
- **Constitutional AI** (Bai et al., 2022) - Self-supervised alignment

B.5 Practical Implementation

B.5.1 Frameworks and Libraries

- **PyTorch**: Most popular for research and LLMs
- **Hugging Face Transformers**: Pre-trained models and tools
- **JAX**: For high-performance ML research
- **DeepSpeed**: Distributed training at scale

B.5.2 Suggested Projects

To solidify your understanding, try implementing:

1. **Basic Neural Network from Scratch**
    - Implement forward pass, backpropagation, and training loop
    - Use only NumPy (no frameworks!)
    - Train on MNIST or a simple dataset

2. **Attention Mechanism**
    - Implement scaled dot-product attention
    - Add multi-head attention
    - Visualize attention weights
3. **Mini Transformer**
    - Build a small transformer (2-4 layers)
    - Train on character-level text generation
    - Experiment with different hyperparameters
4. **Fine-tune a Pre-trained Model**
    - Use Hugging Face to load GPT-2 or similar
    - Fine-tune on your own dataset
    - Try LoRA for efficient adaptation
5. **Implement LoRA**
    - Add low-rank adapters to a model
    - Compare training time and memory usage
    - Evaluate performance vs full fine-tuning

B.6 Staying Current

The field of LLMs moves incredibly fast! Stay updated:

B.6.1 ArXiv and Papers

- Follow **cs.CL** (Computation and Language) on ArXiv
- Check **Papers with Code** for implementations
- Read **Hugging Face Daily Papers** summaries

B.6.2 Blogs and Newsletters

- **The Gradient** - Long-form ML content
- **Lil'Log** by Lilian Weng (OpenAI) - Excellent technical posts
- **Jay Alammar's Blog** - Visual explanations of transformers
- **Sebastian Ruder's Newsletter** - NLP and ML news
- **Import AI** by Jack Clark - Weekly AI news

B.6.3 Communities

- **Hugging Face Forums** - Practical discussions
- **r/MachineLearning** - Reddit community
- **Twitter/X** - Follow researchers (Andrej Karpathy, Yann LeCun, etc.)
- **Discord servers** - EleutherAI, Hugging Face, etc.

B.7 Research Directions

If you're interested in pushing the field forward, exciting open problems include:

B.7.1 Efficiency

- Making attention sub-quadratic
- Model compression and quantization
- Efficient training methods
- Smaller models with similar capabilities

B.7.2 Capabilities

- Longer context windows
- Better reasoning and planning
- Multimodal understanding
- Continuous learning and adaptation

B.7.3 Safety and Alignment

- Robust alignment techniques
- Detecting and mitigating hallucinations
- Interpretability and explainability
- Fairness and bias reduction

B.7.4 Theory

- Understanding emergence and scaling laws
- Theoretical guarantees for learning
- Mathematical foundations of in-context learning
- Loss landscape analysis

B.8 Final Words

You've built a solid mathematical foundation—but this is just the beginning! The field of LLMs is evolving rapidly, and there's always more to learn.

Key principles for continued growth:

- **Build things**: Implementation deepens understanding
- **Read papers**: Stay current with the latest research
- **Teach others**: Explaining clarifies your own understanding
- **Ask questions**: The best researchers are endlessly curious
- **Be patient**: Deep understanding takes time

Mathematics is not just a prerequisite for AI—it's a lens that reveals the underlying beauty and structure of intelligence itself. As you continue your journey, you'll discover that the more you learn, the more connections you see, and the more exciting the field becomes.

The future of AI is being written now, and you have the mathematical tools to help write it. Go build something amazing!

Good luck, and happy learning!