From Numbers to Neural Networks

A Mathematical Journey into Machine Learning

Shuvam Banerji Seal
Slashdot - The Programming Club
sbs22ms076@iiserkol.ac.in

ML Workshop 2026

Contents

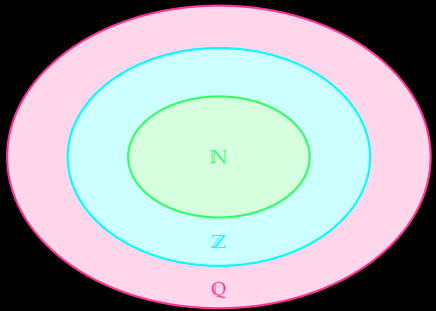
1. Foundations of Numbers

Fun Fact

"In the beginning, there was 1. Then someone wanted more pizza, and mathematics was born."

Our Journey:

- Natural numbers $\mathbb{N}$
- Integers $\mathbb{Z}$
- Rational numbers $\mathbb{Q}$



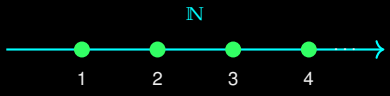
Natural Numbers $\mathbb{N}$: The Universe's Original Counting App

Natural Numbers

The **natural numbers** are the counting numbers:

$$\mathbb{N} = \{1, 2, 3, 4, 5, \ldots\}$$

Some mathematicians include 0, giving $\mathbb{N}_0 = \{0, 1, 2, \ldots\}$



Fun Fact

"Counting: so easy a caveman did it!"

Cave Person Math:

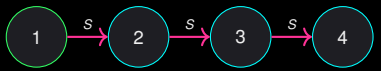
- SHEEP One sheep
- SHEEPSHEEP Two sheep
- SHEEPSHEEPSHEEP Three sheep... *zzz*

Peano Axioms: Making Counting Rigorous

Peano Axioms (1889)

The natural numbers satisfy:

1. $1 \in \mathbb{N}$ *(1 exists)*
2. $\forall n \in \mathbb{N} : S(n) \in \mathbb{N}$ *(every number has a successor)*
3. $\forall n \in \mathbb{N} : S(n) \neq 1$ *(1 is not a successor)*
4. $S(n) = S(m) \Rightarrow n = m$ *(successors are unique)*
5. **Induction axiom** *(if true for 1 and $n \Rightarrow n+1$, true for all)*



Fun Fact

Giuseppe Peano basically said: "Here's how to count. You're welcome, humanity."

Properties of Natural Numbers

Closure Properties

For all $a, b \in \mathbb{N}$:

$$a + b \in \mathbb{N} \quad \checkmark$$
$$a \times b \in \mathbb{N} \quad \checkmark$$
$$a - b \in \mathbb{N} \quad \times$$
$$a \div b \in \mathbb{N} \quad \times$$

The Problem

What is $3 - 5$?

Natural numbers can't handle this! We need... **negative numbers**!

$5 - 3 = 2$ ✓

$3 - 5 =$???

Integers $\mathbb{Z}$: When Mathematicians Discovered Debt

Integers

The **integers** extend natural numbers with negatives and zero:

$$\mathbb{Z} = \{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}$$

Real-world integers:

- Temperature: $-10°C$ (brrr!)
- Bank account: $-\$500$ (oops!)
- Elevation: $-100m$ (underwater)

Fun Fact

The name $\mathbb{Z}$ comes from German "Zahlen" (numbers). Germans: always precise.

Integer Properties

Closure Properties

For all $a, b \in \mathbb{Z}$:

$$a + b \in \mathbb{Z} \quad \checkmark$$
$$a - b \in \mathbb{Z} \quad \checkmark$$
$$a \times b \in \mathbb{Z} \quad \checkmark$$
$$a \div b \in \mathbb{Z} \quad \times$$

Now $3 - 5 = -2 \in \mathbb{Z}$ ✓

Still a Problem!

What is $1 \div 2$?

$\frac{1}{2} = 0.5 \notin \mathbb{Z}$

We need **fractions**!

Key Takeaway

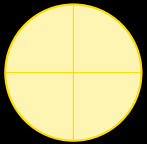
Each number system fixes a problem but creates a new one. Mathematics evolves by solving its own limitations!

Rational Numbers $\mathbb{Q}$: Sharing Pizza Among Friends

Rational Numbers

The **rational numbers** are all fractions:

$$\mathbb{Q} = \left\{ \frac{p}{q} \;\middle|\; p, q \in \mathbb{Z}, q \neq 0 \right\}$$



$\frac{1}{4}$ each!

Examples:

- $\frac{1}{2} = 0.5$
- $\frac{22}{7} \approx 3.14159...$
- $\frac{-3}{4} = -0.75$
- $\frac{6}{3} = 2$ (integers are rational too!)

Fun Fact

"$\mathbb{Q}$" for **Q**uotient. Mathematicians love their abbreviations!

The Density of Rational Numbers

Density of $\mathbb{Q}$

Between **any** two rational numbers, there exists another rational number.

If $a, b \in \mathbb{Q}$ with $a < b$, then $\frac{a+b}{2} \in \mathbb{Q}$ and $a < \frac{a+b}{2} < b$.

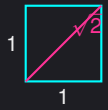
Fun Fact

There are **infinitely many** rationals between any two rationals. Mind = blown! MINDBLOWN

But Wait... Are Rationals Enough?

The Pythagorean Nightmare

Consider a square with side 1:



By Pythagoras: $d^2 = 1^2 + 1^2 = 2$
So $d = \sqrt{2}$... but is $\sqrt{2} \in \mathbb{Q}$?

Spoiler Alert

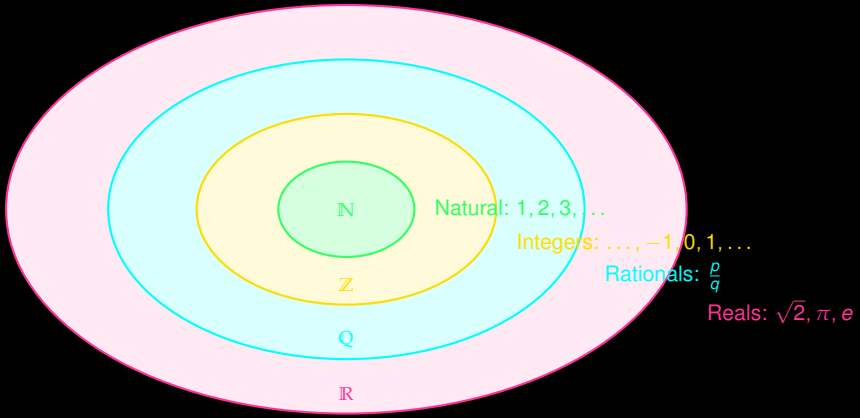
$\sqrt{2}$ is **NOT** rational!

There exist numbers that **cannot** be written as $\frac{p}{q}$.

These are called **irrational numbers**.

Coming up next: Real numbers $\mathbb{R}$!

The Number Hierarchy



$$\mathbb{N} \subset \mathbb{Z} \subset \mathbb{Q} \subset \mathbb{R}$$

Key Takeaways: Numbers

Key Takeaway

1. **Natural numbers** $\mathbb{N}$: Counting (1, 2, 3, ...)
2. **Integers** $\mathbb{Z}$: Add zero and negatives (..., -1, 0, 1, ...)
3. **Rationals** $\mathbb{Q}$: All fractions $\frac{p}{q}$
4. Each extension solves a problem:
   - $\rightarrow$ $\mathbb{Z}$ allows subtraction
   - $\rightarrow$ $\mathbb{Q}$ allows division
5. Rationals are **dense** but not **complete**
6. Some numbers (like $\sqrt{2}$) are **irrational**

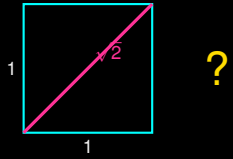
*Next: Real and Complex Numbers — where it gets **really** interesting!*

2. Real & Complex Numbers

Beyond Rationals: The Irrational Truth

Fun Fact

"When the Pythagoreans discovered irrational numbers, they were so upset they allegedly drowned the messenger. Math: serious business since 500 BC."



The Problem:
Not all numbers can be written as $\frac{p}{q}$!

Proof: $\sqrt{2}$ is Irrational

Theorem

$\sqrt{2}$ cannot be expressed as a fraction $\frac{p}{q}$ where $p, q \in \mathbb{Z}$.

Proof by Contradiction:

1. Assume $\sqrt{2} = \frac{p}{q}$ in lowest terms ($\gcd(p, q) = 1$)
2. Then $2 = \frac{p^2}{q^2}$, so $p^2 = 2q^2$
3. Therefore p^2 is even, which means p is even
4. Let $p = 2k$, then $(2k)^2 = 2q^2 \Rightarrow 4k^2 = 2q^2 \Rightarrow q^2 = 2k^2$
5. So q^2 is even, meaning q is even
6. Contradiction! Both p and q are even, but we said $\gcd(p, q) = 1$

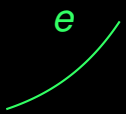
Key Takeaway

$\sqrt{2} \approx 1.41421356...$ goes on forever without repeating! $\square$

Famous Irrational Numbers

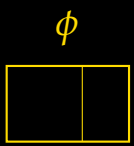
$\pi = 3.14159...$

Ratio of circumference to diameter



$e = 2.71828...$

Base of natural logarithm



$\phi = 1.61803...$

Golden ratio: $\frac{1+\sqrt{5}}{2}$

Fun Fact

These numbers show up *everywhere*: nature, art, physics, and yes... machine learning!

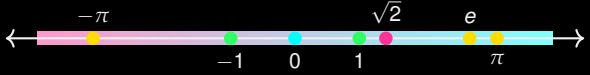
Real Numbers $\mathbb{R}$: Filling the Gaps

Real Numbers

The **real numbers** $\mathbb{R}$ include all rationals AND all irrationals:

$$\mathbb{R} = \mathbb{Q} \cup \{\text{irrational numbers}\}$$

They form a **complete**, **ordered** field — every point on the number line!



Fun Fact

"Nature hates empty spaces" — the number line is now completely filled!

The Completeness Axiom

Completeness of $\mathbb{R}$

Every non-empty subset of $\mathbb{R}$ that is bounded above has a **least upper bound** (supremum) in $\mathbb{R}$.

What this means:

- No "holes" in the number line
- Limits of convergent sequences always exist
- This is what makes calculus possible!

$\mathbb{Q}$ is NOT complete

The sequence $1, 1.4, 1.41, 1.414, \ldots$ converges to $\sqrt{2}$, but $\sqrt{2} \notin \mathbb{Q}$!

$\mathbb{R}$ IS complete

Every convergent sequence of reals has its limit in $\mathbb{R}$.

Complex Numbers $\mathbb{C}$: Inventing the Impossible

The Last Problem

What is $\sqrt{-1}$?

No real number squared gives -1!

$x^2 = -1 \Rightarrow x = ???$

Imaginary Unit

Define i such that:

$$\mathbf{i^2 = -1}$$

Then $\sqrt{-1} = i$

Fun Fact

"Can't solve it? Just **invent** a number!"

— Mathematicians, probably

Powers of i:

$i^0 = 1$

$i^1 = i$

$i^2 = -1$

$i^3 = -i$

$i^4 = 1$ (cycle repeats!)

Complex Numbers

A **complex number** has the form:

$$z = a + bi$$

where $a, b \in \mathbb{R}$ and $i^2 = -1$.

- a = **real part**: $\mathrm{Re}(z)$
- b = **imaginary part**: $\mathrm{Im}(z)$

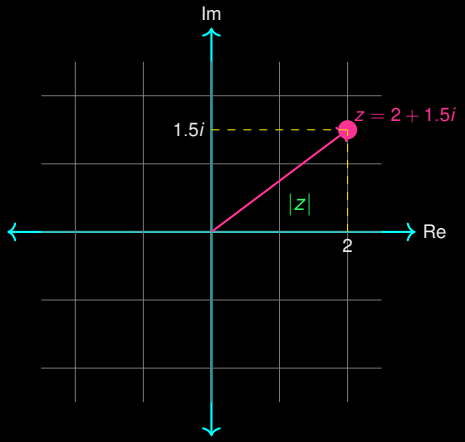
The set of all complex numbers:

$$\mathbb{C} = \{a + bi \mid a, b \in \mathbb{R}\}$$

Examples:

- $3 + 2i$ (real part 3, imaginary part 2)
- $-1 - 4i$ (real part -1, imaginary part -4)
- $5 = 5 + 0i$ (real numbers are complex too!)
- $2i = 0 + 2i$ (purely imaginary)

The Complex Plane



Argand Diagram

Complex numbers live on a 2D plane!

- x-axis: real part
- y-axis: imaginary part

Modulus (magnitude):

$$|z| = \sqrt{a^2 + b^2}$$

Argument (angle):

$$\arg(z) = \tan^{-1}\left(\frac{b}{a}\right)$$

Complex Arithmetic

Let $z_1 = a + bi$ and $z_2 = c + di$

Addition

$$z_1 + z_2 = (a + c) + (b + d)i$$

Just add real and imaginary parts separately!

Subtraction

$$z_1 - z_2 = (a - c) + (b - d)i$$

Multiplication

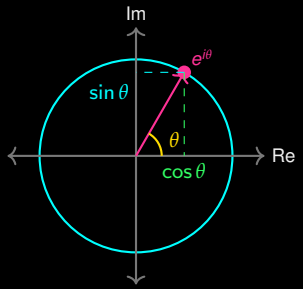
$$\begin{aligned} z_1 \cdot z_2 &= (a + bi)(c + di) \\ &= ac + adi + bci + bdi^2 \\ &= (ac - bd) + (ad + bc)i \end{aligned}$$

Remember: $i^2 = -1$

Example: $(2 + 3i)(1 - i) = 2 - 2i + 3i - 3i^2 = 2 + i + 3 = 5 + i$

Euler's Formula

$$e^{i\theta} = \cos\theta + i\sin\theta$$



When $\theta = \pi$:

$$e^{i\pi} = \cos\pi + i\sin\pi = -1$$

Euler's Identity

$$e^{i\pi} + 1 = 0$$

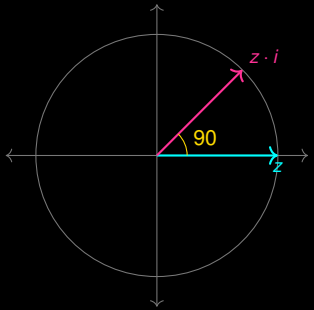
Links five fundamental constants:

- e (natural base)
- i (imaginary unit)
- π (pi)
- 1 (multiplicative identity)
- 0 (additive identity)

Why Complex Numbers Matter in ML

Applications

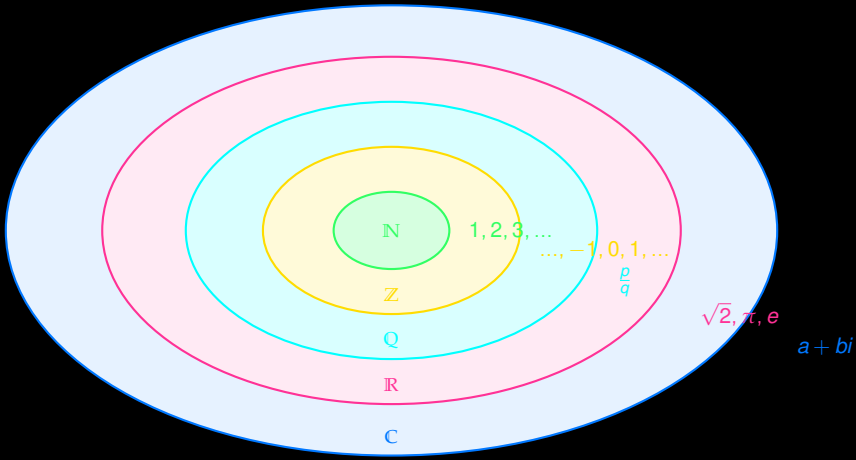
- **Fourier Transforms**
  - → Signal processing
  - → Audio/image analysis
- **Quantum Computing**
  - → Quantum states use $\mathbb{C}$
- **Eigenvalues**
  - → Can be complex!
  - → PCA, neural network analysis
- **Rotations**
  - → Complex multiplication = rotation



Fun Fact

Multiplying by i rotates by 90°!
Complex numbers = rotation superpowers!

The Complete Number Hierarchy



$$\mathbb{N} \subset \mathbb{Z} \subset \mathbb{Q} \subset \mathbb{R} \subset \mathbb{C}$$

Key Takeaways: Real & Complex Numbers

Key Takeaway

1. **Irrational numbers** ($\sqrt{2}, \pi, e$) cannot be written as fractions
2. **Real numbers** $\mathbb{R}$ = rationals + irrationals (complete number line)
3. **Complex numbers** $\mathbb{C}$: $z = a + bi$ where $i^2 = -1$
4. **Euler's formula**: $e^{i\theta} = \cos\theta + i\sin\theta$
5. **Euler's identity**: $e^{i\pi} + 1 = 0$ (the most beautiful equation!)
6. Complex numbers enable:
   - $\rightarrow$ Fourier transforms (signal processing)
   - $\rightarrow$ Quantum computing
   - $\rightarrow$ Elegant rotations

Next: Functions — the building blocks of everything!

3. Functions & Parameters

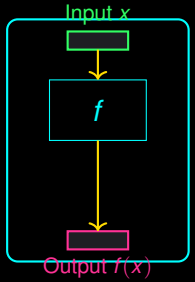
Functions: Mathematical Vending Machines

Fun Fact

"A function is like a vending machine: put in coins (input), get a snack (output). No coins? No snack. Wrong coins? Error!"

Key Questions:

- What goes in? (Domain)
- What comes out? (Range)
- What's the rule? (Function definition)

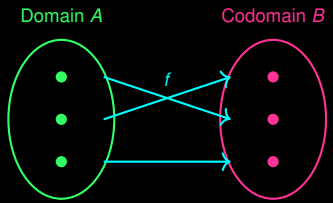


Formal Definition of a Function

Function

A **function** f from set A to set B, written $f : A \to B$, is a rule that assigns to **each** element $x \in A$ **exactly one** element $f(x) \in B$.

- A = **Domain** (all valid inputs)
- B = **Codomain** (possible outputs)
- $\{f(x) : x \in A\}$ = **Range** (actual outputs)



Key Rule

Each input maps to **exactly one** output!

One input $\to$ multiple outputs = NOT a function!

Function Notation

Standard Notation

Defining a function:

$$f(x) = x^2$$
$$g(x) = 2x + 1$$
$$h(x, y) = x^2 + y^2$$

Evaluating:

$$f(3) = 3^2 = 9$$
$$g(-1) = 2(-1) + 1 = -1$$
$$h(1, 2) = 1 + 4 = 5$$

Arrow Notation

$$f : \mathbb{R} \to \mathbb{R}$$
$$x \mapsto x^2$$

"f takes real numbers to real numbers, mapping x to x^2"

Fun Fact

Same thing, fancier packaging. Mathematicians love options!

Domain, Codomain, and Range

Example: $f(x) = \sqrt{x}$ where $f : \mathbb{R}_{\geq 0} \to \mathbb{R}$

Domain

All valid inputs

For $\sqrt{x}$: $x \geq 0$
(Can't sqrt negative reals!)

$$\text{Dom}(f) = [0, \infty)$$

Codomain

Set of possible outputs

Here: all real numbers $\mathbb{R}$
(We *could* land anywhere)

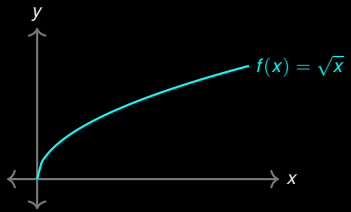
$$\text{Codomain} = \mathbb{R}$$

Range

Actual outputs produced

$\sqrt{x} \geq 0$ always!
Range $\subseteq$ Codomain

$$\text{Range}(f) = [0, \infty)$$



Parameters vs Variables: The Director and the Actor

Variables

Variables are the **inputs** to a function.

They vary — that's why we call them variables!

Example: In $f(x) = x^2$, x is the variable.

Variables = **Actors**
(They perform based on the script)

Parameters

Parameters are **constants** that control behavior.

They're fixed during evaluation but can be tuned!

Example: In $f(x) = mx + b$, m and b are parameters.

Parameters = **Director**
(They control how the show runs)

Parametric Notation

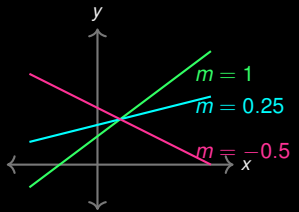
We write $f(x;\theta)$ to show:

- x = variable (input)
- θ = parameter(s) (controls shape/behavior)

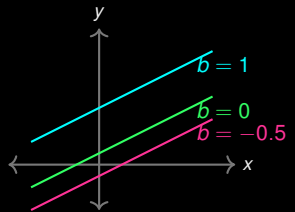
The semicolon separates variables from parameters.

Example: Linear Function

$$f(x;m,b) = mx + b$$



Varying m (slope)



Varying b (intercept)

Why Parameters Matter in Machine Learning

The ML Connection

In Machine Learning:

- **Variables** (x) = your data (inputs/features)
- **Parameters** (θ) = what the model **learns**!

Training a neural network = finding the best parameters θ^*

Example: Neural Network

$$\hat{y} = f(x; W, b) = \sigma(Wx + b)$$

- x = input data (fixed during prediction)
- W = weight matrix (learned)
- b = bias vector (learned)
- σ = activation function

Fun Fact

Training = Turning knobs (θ) until the output looks right!

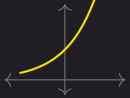
Common Function Types

Linear

$f(x) = mx + b$

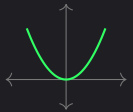
Exponential

$f(x) = a^x$ or e^x



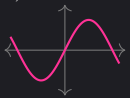
Polynomial

$f(x) = a_n x^n + ... + a_1 x + a_0$



Trigonometric

$\sin(x), \cos(x), \tan(x)$



Function Composition: Chaining Functions

Composition

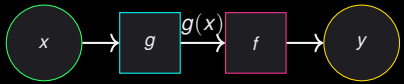
The **composition** of f and g, written $(f \circ g)(x)$:

$$(f \circ g)(x) = f(g(x))$$

"Apply g first, then apply f to the result"

Example:

$$f(x) = x^2$$
$$g(x) = x + 1$$
$$(f \circ g)(x) = f(g(x)) = f(x+1) = (x+1)^2$$
$$(g \circ f)(x) = g(f(x)) = g(x^2) = x^2 + 1$$



$$y = (f \circ g)(x) = f(g(x))$$

Alert

Order matters! $f \circ g \neq g \circ f$ in general!

Neural Networks = Composed Functions

A neural network is just a big composition of simple functions!

$$\hat{y} = (f_4 \circ f_3 \circ f_2 \circ f_1)(x) = f_4(f_3(f_2(f_1(x))))$$

Where each $f_i(x) = \sigma(W_i x + b_i)$ is a simple layer operation!

Inverse Function

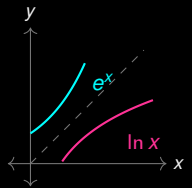
If $f : A \to B$, the **inverse** $f^{-1} : B \to A$ satisfies:

$$f^{-1}(f(x)) = x \quad \text{and} \quad f(f^{-1}(y)) = y$$

The inverse "undoes" what the function does.

Examples:

- $f(x) = x + 3 \Rightarrow f^{-1}(x) = x - 3$
- $f(x) = 2x \Rightarrow f^{-1}(x) = \frac{x}{2}$
- $f(x) = e^x \Rightarrow f^{-1}(x) = \ln(x)$
- $f(x) = x^2 \Rightarrow f^{-1}(x) = \sqrt{x}$ (for $x \geq 0$)



Inverse reflects across $y = x$

Key Takeaways: Functions & Parameters

Key Takeaway

1. A **function** maps inputs to outputs: $f : A \to B$
2. **Domain** = valid inputs, **Range** = actual outputs
3. **Variables** (x) = inputs that vary
4. **Parameters** (θ) = constants that control behavior
5. In ML: x = data, θ = what we **learn**!
6. **Composition**: $(f \circ g)(x) = f(g(x))$
7. Neural networks = composed functions with learnable parameters
8. **Inverse** f^{-1} undoes the function

Next: Limits — the foundation of calculus!

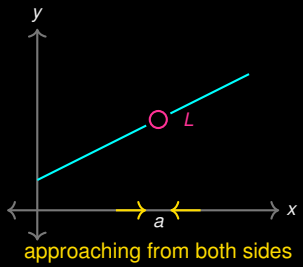
4. Limits

Limits: Getting Infinitely Close

Fun Fact

"A limit is like trying to touch your nose with your tongue — you can get really, really close, but maybe never quite there!"

The Big Question:
What happens to $f(x)$ as x gets **infinitely close** to some value a?



Informal Limit Definition

We write:

$$\lim_{x \to a} f(x) = L$$

and say "the limit of $f(x)$ as x approaches a equals L"

Meaning: As x gets closer and closer to a, $f(x)$ gets closer and closer to L.

Example: $\lim_{x \to 2}(3x + 1) = ?$

x	1.9	1.99	1.999	2.001	2.01	2.1
$f(x)$	6.7	6.97	6.997	7.003	7.03	7.3

As $x \to 2$, we have $f(x) \to 7$!

Epsilon-Delta Definition

$$\lim_{x \to a} f(x) = L$$

means: For every $\varepsilon > 0$, there exists $\delta > 0$ such that:

$$0 < |x - a| < \delta \implies |f(x) - L| < \varepsilon$$

In English:

- Pick any tolerance ε (how close to L)
- I can find a δ (how close x needs to be to a)
- Such that staying within δ of a keeps $f(x)$ within ε of L

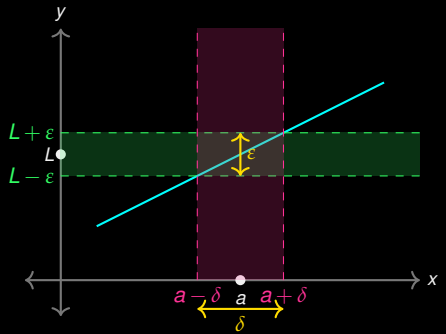
Fun Fact

It's like a game:

You: "Get within ε of L!"
Me: "Fine, stay within δ of a."

If I can *always* win, the limit exists!

Visualizing ε-δ



If x is in the pink zone, $f(x)$ stays in the green zone!

Limit Laws: The Algebra of Limits

If $\lim_{x \to a} f(x) = L$ and $\lim_{x \to a} g(x) = M$, then:

Addition

$$\lim_{x \to a} [f(x) + g(x)] = L + M$$

Subtraction

$$\lim_{x \to a} [f(x) - g(x)] = L - M$$

Multiplication

$$\lim_{x \to a} [f(x) \cdot g(x)] = L \cdot M$$

Division

$$\lim_{x \to a} \frac{f(x)}{g(x)} = \frac{L}{M} \quad (M \neq 0)$$

Constant Multiple

$$\lim_{x \to a} [c \cdot f(x)] = c \cdot L$$

Power

$$\lim_{x \to a} [f(x)]^n = L^n$$

One-Sided Limits

Left-Hand Limit

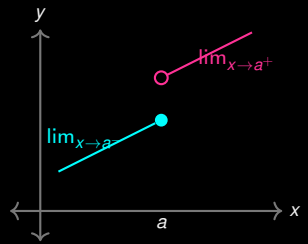
$$\lim_{x \to a^-} f(x) = L$$

Approaching from the **left** (values less than a)

Right-Hand Limit

$$\lim_{x \to a^+} f(x) = L$$

Approaching from the **right** (values greater than a)



Alert

Limit exists $\iff$ both one-sided limits exist and are equal!

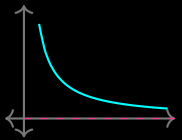
Limit at Infinity

$$\lim_{x \to \infty} f(x) = L$$

As x grows without bound, $f(x)$ approaches L.

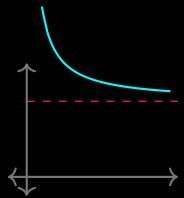
Examples:

$$\lim_{x \to \infty} \frac{1}{x} = 0$$



Horizontal asymptote at $y = 0$

$$\lim_{x \to \infty} \frac{2x+1}{x} = 2$$



Horizontal asymptote at $y = 2$

Indeterminate Forms

Danger Zone!

Some limits look like they give answers but don't:

$$\frac{0}{0} \quad \frac{\infty}{\infty} \quad 0 \cdot \infty \quad \infty - \infty \quad 0^0 \quad 1^\infty \quad \infty^0$$

These are **indeterminate** — they need more work!

Example: $\lim_{x \to 0} \frac{\sin x}{x} = \frac{0}{0}$???

But actually:

$$\lim_{x \to 0} \frac{\sin x}{x} = 1$$

(This is a famous result!)

Fun Fact

$\frac{0}{0}$ doesn't mean "zero" — it means "figure it out another way!"

L'Hôpital's Rule

If $\lim_{x \to a} \frac{f(x)}{g(x)}$ gives $\frac{0}{0}$ or $\frac{\infty}{\infty}$, then:

$$\lim_{x \to a} \frac{f(x)}{g(x)} = \lim_{x \to a} \frac{f'(x)}{g'(x)}$$

(provided the right-hand limit exists)

Example: $\lim_{x \to 0} \frac{\sin x}{x}$

- Direct: $\frac{\sin 0}{0} = \frac{0}{0}$ (indeterminate!)
- L'Hôpital: $\lim_{x \to 0} \frac{\cos x}{1} = \frac{\cos 0}{1} = 1$ (YES)

Info

We'll learn about derivatives (f') in the next section — then L'Hôpital makes sense!

Continuity: No Jumps, No Holes

Continuous Function

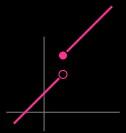
A function f is **continuous** at $x = a$ if:

$$\lim_{x \to a} f(x) = f(a)$$

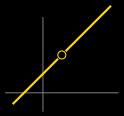
Three conditions:

1. $f(a)$ exists (defined at a)
2. $\lim_{x \to a} f(x)$ exists
3. They're equal!

Continuous (YES)



Jump (NO)



Hole (NO)

Why Limits Matter in Machine Learning

Applications in ML

1. **Derivatives** (next section!) are defined using limits
   → Gradients for backpropagation!
2. **Convergence of training**
   → Does the loss approach a minimum?
3. **Asymptotic analysis**
   → Algorithm complexity: $O(n)$, $O(n^2)$
4. **Activation functions**
   → $\lim_{x \to \infty} \sigma(x) = 1$ (sigmoid saturation)

Key Takeaways: Limits

Key Takeaway

1. **Limit**: What $f(x)$ approaches as $x \to a$
2. **Notation**: $\lim_{x \to a} f(x) = L$
3. ε-δ **definition**: The rigorous foundation
4. **Limit laws**: Add, subtract, multiply, divide limits
5. **One-sided limits**: From left (a^-) or right (a^+)
6. **Limits at infinity**: Behavior as $x \to \pm\infty$
7. **Indeterminate forms**: $\frac{0}{0}$, $\frac{\infty}{\infty}$, etc. need special handling
8. **Continuity**: No jumps, no holes, no surprises

Next: Differentiation — the heart of calculus!

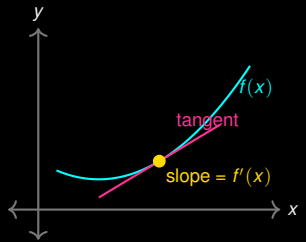
5. Differentiation

Differentiation: The Art of Measuring Change

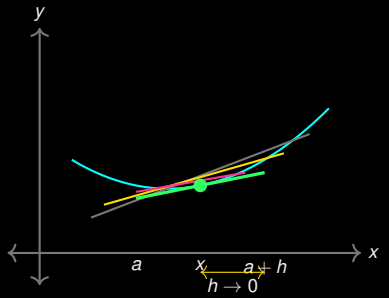
Fun Fact

"How fast is your position changing? Ask the derivative! How fast is your speed changing? Ask the second derivative! How fast is your bank account changing? Don't ask."

The Big Question:
How fast is $f(x)$ changing at point x?



From Secant to Tangent



As $h \to 0$, the secant line becomes the tangent line!

The Derivative: Definition

Derivative

The **derivative** of f at x is:

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

provided this limit exists.

Alternative notations:

$$f'(x) = \frac{df}{dx} = \frac{d}{dx} f(x) = Df(x) = \dot{f}$$

What it measures:

- Instantaneous rate of change
- Slope of tangent line
- Sensitivity of output to input

Fun Fact

The derivative is asking: "If I nudge x a tiny bit, how much does $f(x)$ change?"

Example: Deriving $f(x) = x^2$

$$
\begin{aligned}
f'(x) &= \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \\
&= \lim_{h \to 0} \frac{(x+h)^2 - x^2}{h} \\
&= \lim_{h \to 0} \frac{x^2 + 2xh + h^2 - x^2}{h} \\
&= \lim_{h \to 0} \frac{2xh + h^2}{h} \\
&= \lim_{h \to 0} (2x + h) \\
&= \mathbf{2x}
\end{aligned}
$$

Key Takeaway

If $f(x) = x^2$, then $f'(x) = 2x$. At $x = 3$: slope $= 6$!

Differentiation Rules: Power Rule

Power Rule

If $f(x) = x^n$, then:

$$\mathbf{f'(x) = nx^{n-1}}$$

Works for any real n!

Examples:

$$\frac{d}{dx}(x^3) = 3x^2$$

$$\frac{d}{dx}(x^5) = 5x^4$$

$$\frac{d}{dx}(x^{100}) = 100x^{99}$$

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

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

$$\frac{d}{dx}(1) = \frac{d}{dx}(x^0) = 0$$

Fun Fact

"Bring down the power, reduce by one" — calculus's most satisfying rule!

More Differentiation Rules

Constant Rule

$$\frac{d}{dx}(c) = 0$$

Constants don't change!

Constant Multiple

$$\frac{d}{dx}(cf) = c \cdot f'$$

Sum Rule

$$\frac{d}{dx}(f+g) = f' + g'$$

Product Rule

$$\frac{d}{dx}(fg) = f'g + fg'$$

"First times derivative of second, plus second times derivative of first"

Quotient Rule

$$\frac{d}{dx}\left(\frac{f}{g}\right) = \frac{f'g - fg'}{g^2}$$

"Low d-high minus high d-low, over low squared"

Chain Rule

If $y = f(g(x))$, then:

$$\frac{\mathbf{dy}}{\mathbf{dx}} = \mathbf{f}'(\mathbf{g}(\mathbf{x})) \cdot \mathbf{g}'(\mathbf{x})$$

Or in Leibniz notation:

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

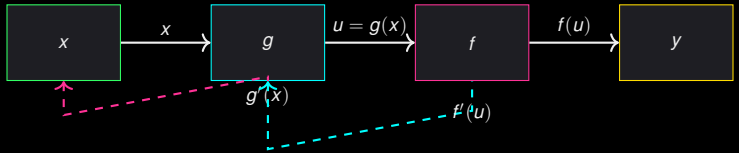
where $u = g(x)$

Example: Find $\frac{d}{dx}(x^2 + 1)^3$
Let $u = x^2 + 1$, so $y = u^3$

$$\frac{dy}{du} = 3u^2, \quad \frac{du}{dx} = 2x$$

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

Chain Rule: The Pipeline



$$\frac{dy}{dx} = \underbrace{f'(g(x))}_{\text{outer derivative}} \cdot \underbrace{g'(x)}_{\text{inner derivative}}$$

Fun Fact

Chain rule: Multiply all the “rates of change” along the pipeline!

Important Derivatives to Know

Exponential & Log

$$\frac{d}{dx}(e^x) = e^x$$
$$\frac{d}{dx}(a^x) = a^x \ln(a)$$
$$\frac{d}{dx}(\ln x) = \frac{1}{x}$$
$$\frac{d}{dx}(\log_a x) = \frac{1}{x \ln a}$$

Trigonometric

$$\frac{d}{dx}(\sin x) = \cos x$$
$$\frac{d}{dx}(\cos x) = -\sin x$$
$$\frac{d}{dx}(\tan x) = \sec^2 x$$

Star of the Show

$\frac{d}{dx}(e^x) = e^x$ — the exponential is its own derivative!
This is why e appears everywhere in calculus and ML!

Higher Derivatives

The **second derivative** is the derivative of the derivative:

$$f''(x) = \frac{d^2f}{dx^2} = \frac{d}{dx}\left(\frac{df}{dx}\right)$$

And we can keep going: $f'''(x)$, $f^{(4)}(x)$, etc.

Physical Interpretation:

- $f(t)$ = position at time t
- $f'(t)$ = velocity (rate of change of position)
- $f''(t)$ = acceleration (rate of change of velocity)
- $f'''(t)$ = jerk (rate of change of acceleration)

Fun Fact

Jerk is what makes roller coasters fun... or terrifying!

Partial Derivative

For $f(x, y)$, the **partial derivative** with respect to x:

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

Treat y as constant, differentiate with respect to x only.

Example: $f(x, y) = x^2 y + 3xy^2$

$$\frac{\partial f}{\partial x} = 2xy + 3y^2 \quad \text{(treat } y \text{ as constant)}$$

$$\frac{\partial f}{\partial y} = x^2 + 6xy \quad \text{(treat } x \text{ as constant)}$$

ML Connection

Neural networks have **many** parameters. Partial derivatives tell us how to adjust each one!

The Gradient: All Partials Together

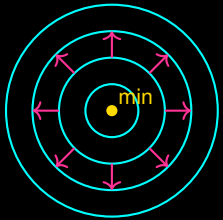
Gradient

The **gradient** of $f(x_1, x_2, ..., x_n)$ is a vector of all partial derivatives:

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n} \right)$$

Properties:

- Points in direction of steepest increase
- Magnitude = rate of steepest increase
- Perpendicular to level curves



∇f points “uphill”

Why Derivatives Matter in ML

The Core of Machine Learning

Training = Minimizing a Loss Function

To find the minimum, we need **derivatives**!

Gradient Descent:

$$\theta_{\text{new}} = \theta_{\text{old}} - \eta \cdot \nabla_\theta L(\theta)$$

- θ = model parameters
- L = loss function
- η = learning rate
- $\nabla_\theta L$ = gradient (tells us which way is "downhill")

Fun Fact

Gradient descent: "Follow the slope downhill until you reach the bottom!"

Key Takeaways: Differentiation

Key Takeaway

1. **Derivative** $f'(x) = \lim_{h \to 0} \frac{f(x+h)-f(x)}{h}$
2. **Geometric meaning**: Slope of tangent line
3. **Power rule**: $\frac{d}{dx}(x^n) = nx^{n-1}$
4. **Chain rule**: $(f \circ g)' = f'(g) \cdot g'$ — crucial for neural nets!
5. **Partial derivatives**: Rate of change in one variable
6. **Gradient** ∇f: Vector of all partial derivatives
7. **ML connection**: Gradients guide learning (gradient descent)

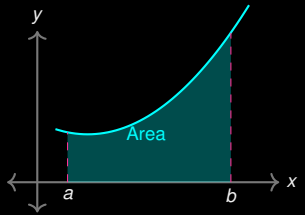
Next: Integration — the inverse of differentiation!

6. Integration

Integration: Adding Infinitely Many Infinitely Small Pieces

Fun Fact

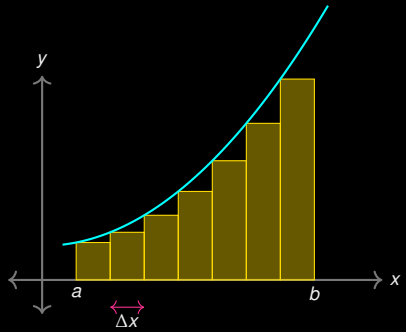
"Integration is like eating pizza: one slice at a time, you can consume the whole thing. Except with integration, the slices are infinitely thin!"



Two Big Ideas:

1. Finding areas under curves
2. Reversing differentiation

Riemann Sums: Approximating Area

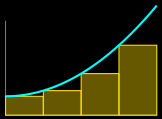


Riemann Sum

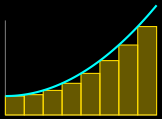
$$\text{Area} \approx \sum_{i=1}^{n} f(x_i) \cdot \Delta x$$

where $\Delta x = \frac{b-a}{n}$ is the width of each rectangle.

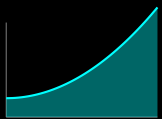
From Rectangles to Integrals



$n = 4$



$n = 8$



$n \to \infty$

Definition of Definite Integral

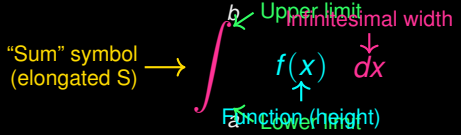
$$\int_a^b f(x)\,dx = \lim_{n\to\infty} \sum_{i=1}^{n} f(x_i) \cdot \Delta x$$

Fun Fact

As we use more and more rectangles ($n \to \infty$), we get the *exact* area!

Understanding Integral Notation

$$\int_a^b f(x)\,dx$$



Info

The integral $\int$ is a stretched "S" for "Sum" — we're adding up infinitely many $f(x) \cdot dx$ pieces!

Antiderivatives: Reversing Differentiation

Antiderivative

$F(x)$ is an **antiderivative** of $f(x)$ if:

$$F'(x) = f(x)$$

The antiderivative "undoes" differentiation!

Examples:

$f(x)$	$F(x)$ (antiderivative)	Check: $F'(x) = f(x)$?
x^2	$\frac{x^3}{3}$	$\frac{d}{dx}\left(\frac{x^3}{3}\right) = x^2$ (YES)
$\cos x$	$\sin x$	$\frac{d}{dx}(\sin x) = \cos x$ (YES)
e^x	e^x	$\frac{d}{dx}(e^x) = e^x$ (YES)

Alert

But wait — if $F(x)$ is an antiderivative, so is $F(x) + C$ for any constant C!
(Because $\frac{d}{dx}(F + C) = F' + 0 = f$)

Indefinite Integral

The **indefinite integral** represents all antiderivatives:

$$\int f(x)\,dx = F(x) + C$$

where C is the "constant of integration" (can be any number).

Basic Integrals:

$$\int x^n\,dx = \frac{x^{n+1}}{n+1} + C \quad (n \neq -1)$$

$$\int e^x\,dx = e^x + C$$

$$\int \frac{1}{x}\,dx = \ln|x| + C$$

$$\int \sin x\,dx = -\cos x + C$$

$$\int \cos x\,dx = \sin x + C$$

$$\int 1\,dx = x + C$$

Fun Fact

Don't forget the $+C$! It's the most forgotten symbol in calculus.

FTC Part 1

If f is continuous on $[a, b]$ and:

$$g(x) = \int_a^x f(t)\,dt$$

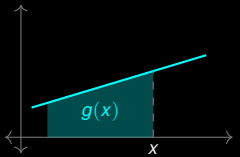
Then g is differentiable and:

$$g'(x) = f(x)$$

In words:
The derivative of the "area so far" function equals the original function!

Symbolically:

$$\frac{d}{dx}\int_a^x f(t)\,dt = f(x)$$



Rate of area growth = height of curve!

FTC Part 2 (Evaluation Theorem)

If f is continuous on $[a, b]$ and F is any antiderivative of f, then:

$$\int_{\mathbf{a}}^{\mathbf{b}} \mathbf{f(x)}\,\mathbf{dx} = \mathbf{F(b)} - \mathbf{F(a)}$$

This is HUGE! Instead of computing limits of Riemann sums, just:

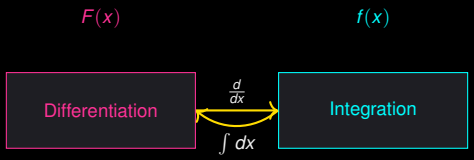
1. Find an antiderivative F
2. Evaluate at the endpoints
3. Subtract!

Example: $\int_0^2 x^2\,dx$

Antiderivative: $F(x) = \frac{x^3}{3}$

Answer: $F(2) - F(0) = \frac{8}{3} - 0 = \mathbf{\frac{8}{3}}$

The Beautiful Connection



Key Takeaway

Differentiation and Integration are **inverse operations**!

- Differentiate then integrate: back where you started (+ C)
- Integrate then differentiate: exactly back where you started

Fun Fact

They're like frenemies — opposite but inseparable!

u-Substitution

For $\int f(g(x)) \cdot g'(x)\, dx$:

1. Let $u = g(x)$
2. Then $du = g'(x)\, dx$
3. Substitute: $\int f(u)\, du$
4. Integrate and substitute back

Example: $\int 2x \cdot e^{x^2}\, dx$

- Let $u = x^2$, so $du = 2x\, dx$
- Substitute: $\int e^u\, du = e^u + C$
- Substitute back: $\mathbf{e^{x^2} + C}$

Fun Fact

"When in doubt, u-sub it out!" — Every calculus student

Integration by Parts

$$\int u\,dv = uv - \int v\,du$$

Derived from the product rule: $(uv)' = u'v + uv'$

Example: $\int x \cdot e^x\,dx$

Choose:

- $u = x \Rightarrow du = dx$
- $dv = e^x\,dx \Rightarrow v = e^x$

Apply:

$$= x \cdot e^x - \int e^x\,dx$$
$$= xe^x - e^x + C$$
$$= \mathbf{e^x(x-1)+C}$$

LIATE Rule

Choose u in order: **L**og, **I**nverse trig, **A**lgebraic, **T**rig, **E**xponential

Applications

1. **Probability Distributions**

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

   (Area under probability density function)

2. **Expected Values**

$$E[X] = \int_{-\infty}^{\infty} x \cdot f(x)\,dx$$

3. **Loss Functions** (continuous case)
4. **Backpropagation** involves computing areas/volumes
5. **Gaussian Integrals** (normal distribution!)

Key Takeaways: Integration

Key Takeaway

1. **Riemann sum**: Approximate area with rectangles
2. **Definite integral** $\int_a^b f(x)\,dx$: Exact area under curve
3. **Antiderivative**: F where $F' = f$
4. **Indefinite integral**: $\int f\,dx = F + C$
5. **FTC**: $\int_a^b f\,dx = F(b) - F(a)$
6. **Integration and differentiation are inverses!**
7. **Techniques**: Substitution, by parts
8. **ML uses**: Probability, expectations, continuous losses

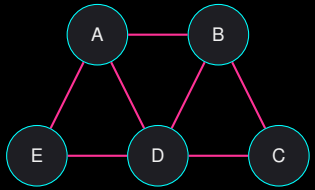
Next: Graph Theory — the foundation for neural network structure!

7. Graph Theory Basics

Graph Theory: Networks Before the Internet

Fun Fact

"Facebook, Twitter, neural networks — they're all just graphs in disguise. Euler figured this out in 1736 while solving a bridge puzzle!"



Why Graphs for ML?

- Neural networks ARE graphs
- Information flows through edges
- Structure determines computation

What is a Graph?

Graph

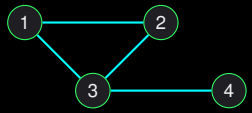
A **graph** $G = (V, E)$ consists of:

- V = set of **vertices** (or nodes)
- E = set of **edges** (connections between vertices)

Each edge $e \in E$ connects two vertices.

Example:

$$V = \{1, 2, 3, 4\}$$
$$E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{3, 4\}\}$$



Types of Graphs

Undirected Graph

Edges have no direction — connection is mutual.

A and B are connected (symmetric).

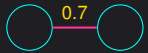
Directed Graph (Digraph)

Edges have direction — one-way connection.

A points to B (asymmetric).

Weighted Graph

Edges have associated values (weights).



Connection strength = 0.7

Neural Networks

Neural networks are **directed, weighted** graphs!

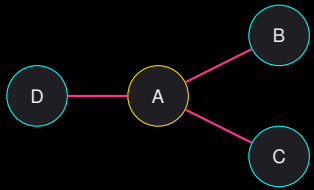
- Neurons = vertices
- Weights = edge values
- Direction = information flow

Degree of a Vertex

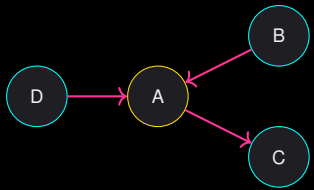
Degree

The **degree** of a vertex is the number of edges connected to it.
For directed graphs:

- **In-degree**: edges coming IN
- **Out-degree**: edges going OUT



deg(A) = 3



in(A)=2, out(A)=1

Path

A **path** is a sequence of vertices connected by edges:

$$v_0 \to v_1 \to v_2 \to \cdots \to v_k$$

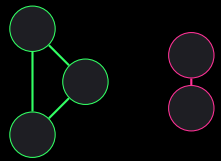
Path length = number of edges (here: k)

Connected Graph

A graph is **connected** if there exists a path between every pair of vertices.

Fun Fact

Can you get from any vertex to any other? If yes: connected!



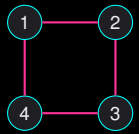
NOT connected (2 components)

Adjacency Matrix: Graphs as Matrices

Adjacency Matrix

For a graph with n vertices, the **adjacency matrix** A is an $n \times n$ matrix where:

$$A_{ij} = \begin{cases} 1 & \text{if edge from } i \text{ to } j \\ w_{ij} & \text{if weighted edge} \\ 0 & \text{otherwise} \end{cases}$$



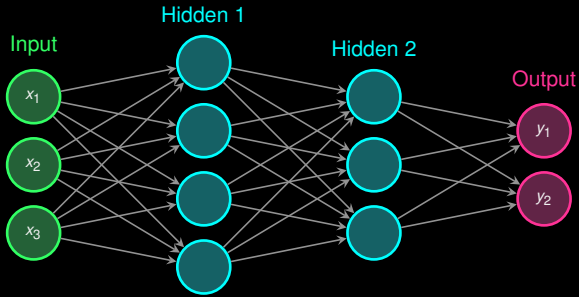
$$A = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix}$$

Row i, Column j = connection from i to j

Key Takeaway

For undirected graphs: A is symmetric ($A = A^T$)

Neural Networks AS Graphs



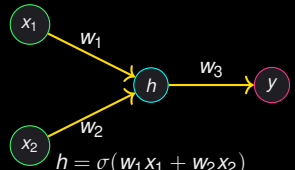
- **Vertices**: Neurons (circles)
- **Edges**: Connections (arrows)
- **Weights**: Edge values (learned!)
- **Direction**: Left to right (feedforward)
- **Layers**: Groups of vertices
- **Computation**: Flows through graph

Information Flow Through Graphs

Forward Pass

Information flows from input to output:

1. Input enters at source nodes
2. Each node computes a value
3. Values propagate along edges
4. Output emerges at sink nodes



Key Takeaway

The graph structure determines:

- What information each neuron receives
- How information is combined
- What the network can compute!

Graph Properties for ML

DAG

A **Directed Acyclic Graph** has:

- Directed edges
- No cycles (can't loop back)

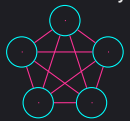
Feedforward neural networks are DAGs!

Bipartite Graph

Vertices split into two groups — edges only between groups.
Adjacent layers in a NN form bipartite subgraphs!

Complete Graph

Every vertex connected to every other.



"Fully connected" layers!

Key Takeaways: Graph Theory

Key Takeaway

1. **Graph** $G = (V, E)$: vertices and edges
2. **Types**: undirected, directed, weighted
3. **Degree**: number of connections
4. **Path**: sequence of connected vertices
5. **Adjacency matrix**: graph as a matrix
6. **Neural networks are graphs!**
   - $\rightarrow$ Neurons = vertices
   - $\rightarrow$ Connections = weighted, directed edges
   - $\rightarrow$ Feedforward NN = DAG
7. Structure determines computation capability

Next: Python Environment Setup — preparing our tools!

8. Python Environment Setup

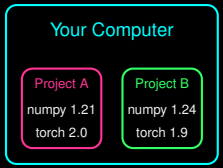
Python Environment Setup

Fun Fact

"Before we teach a machine to learn, we must first teach our computer to find Python. And not just any Python — the RIGHT Python!"

Why environments matter:

- Different projects need different packages
- Version conflicts are REAL
- Reproducibility is essential



Isolated environments = no conflicts!

Enter: uv — The Fast Python Package Manager

What is uv?

uv is an extremely fast Python package and project manager, written in Rust.

- 10-100x faster than pip
- Handles virtual environments automatically
- Modern project management
- Replaces: pip, pip-tools, virtualenv, poetry, pyenv

Installing uv

macOS/Linux:

```
curl -LsSf https://astral.sh/uv/install.sh
| sh
```

Windows:

```
powershell -c "irm https://astral.sh/uv/inst
| iex"
```

Fun Fact

Why “uv”?
Because it’s so fast it operates at **UV frequencies** — ultraviolet light!
(Actually, it’s just a cool name.)

Creating a New Project: `uv init`

uv init

Creates a new Python project with proper structure:

```
uv init my_ml_project
cd my_ml_project
```

What gets created:

```
my_ml_project/
  +- .python-version
  +- pyproject.toml
  +- README.md
  +- hello.py
```

Key Files

- `.python-version`: Python version
- `pyproject.toml`: Project config
- `README.md`: Documentation
- `hello.py`: Sample code

Understanding `pyproject.toml`

pyproject.toml

The **modern standard** for Python project configuration.
Defines: name, version, dependencies, tools, scripts.

```
[project]
name = "ml-workshop"
version = "0.1.0"
requires-python = ">=3.11"
dependencies = [
  "numpy>=1.24.0",
  "torch>=2.0.0",
]
[tool.uv]
dev-dependencies = [
  "pytest>=7.0.0",
]
```

Sections

- `[project]`: Metadata
- `dependencies`: Runtime deps
- `[tool.uv]`: uv-specific
- `dev-dependencies`: Dev tools

Installing Dependencies: `uv sync`

uv sync

The **magic command** that:

- Creates a virtual environment (if needed)
- Installs all dependencies from pyproject.toml
- Locks versions in `uv.lock`

Basic usage:

```
$ uv sync
```

With dev dependencies:

```
$ uv sync --dev
```

Add new package:

```
$ uv add pandas
```

Lock File

`uv.lock` ensures:

- Exact reproducibility
- Same versions everywhere
- Commit to git!

Running Python with uv

uv run

Run commands in the project's virtual environment:

```
uv run python script.py
uv run pytest
uv run jupyter notebook
```

No Activation Needed!

Unlike traditional venvs:

- No `source .venv/bin/activate`
- No `deactivate`
- Just prefix with `uv run`

Fun Fact

Think of `uv run` as a magic portal that teleports your command into the right Python universe!

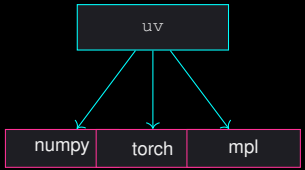
Setting Up Our Workshop Environment

Step by Step

1. Install uv (if not done)
2. Clone/create workshop project
3. Run `uv sync`
4. Start coding!

Our dependencies:

- `numpy` — numerical computing
- `torch` — deep learning
- `matplotlib` — plotting
- `scikit-learn` — ML algorithms
- `jupyter` — notebooks



uv manages everything!

Quick Reference: Essential `uv` Commands

Project Setup

`uv init` — Create project
`uv sync` — Install dependencies
`uv add PKG` — Add package
`uv remove PKG` — Remove package

Running Code

```
uv run python X.py
uv run pytest
uv run jupyter notebook
uv run CMD
```

Remember

- Always use `uv run` to execute Python in the project environment
- Commit both `pyproject.toml` AND `uv.lock` to version control
- Run `uv sync` after pulling changes

Python Setup: Summary

What We Learned

- Why virtual environments matter
- How to use `uv` for fast package management
- Project structure with `pyproject.toml`
- Running code with `uv run`

Fun Fact

Now your Python environment is ready for machine learning!
Let's make those neurons fire!

Next: Neural Networks!

9. Introduction to Neural Networks

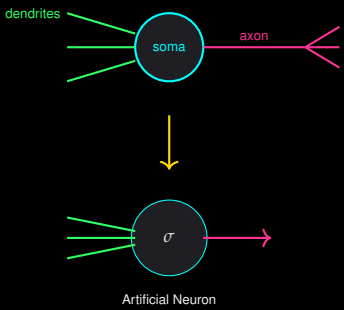
Neural Networks: Inspired by Biology

Fun Fact

"Humans tried to build flying machines by copying birds. It kinda worked. Then we tried copying brains. It... also kinda worked?"

The Big Idea:

- Brains are amazing at learning
- Brains are made of neurons
- Let's build artificial neurons!



Artificial Neuron

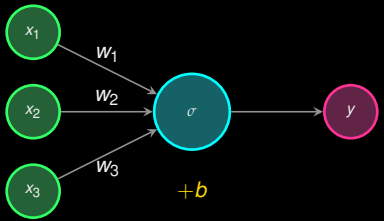
Perceptron (Rosenblatt, 1958)

A **perceptron** computes:

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

Where:

- $\mathbf{x} = (x_1, \ldots, x_n)$: inputs
- $\mathbf{w} = (w_1, \ldots, w_n)$: weights
- b: bias
- σ: activation function



Step 1: Weighted Sum (Linear Combination)

Weighted Sum

The neuron computes a weighted sum of inputs:

$$z = \sum_{i=1}^{n} w_i x_i + b = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n + b$$

Example:

$$\mathbf{x} = (0.5, 0.8, -0.3)$$
$$\mathbf{w} = (0.4, -0.2, 0.6)$$
$$b = 0.1$$

$$\begin{aligned} z &= 0.4(0.5) + (-0.2)(0.8) + 0.6(-0.3) + 0.1 \\ &= 0.2 - 0.16 - 0.18 + 0.1 \\ &= -0.04 \end{aligned}$$

Fun Fact

Weighted sum = "voting with different levels of influence"

Big $|w_i|$ = strong opinion
$w_i > 0$ = votes YES
$w_i < 0$ = votes NO

Step 2: Activation Function — Adding Non-Linearity

The Problem

Linear functions composed = still linear!

$$f_2(f_1(\mathbf{x})) = W_2(W_1\mathbf{x} + b_1) + b_2 = W_2W_1\mathbf{x} + (W_2b_1 + b_2) = W'\mathbf{x} + b'$$

Solution: Non-Linear Activation

Apply a non-linear function σ after each linear transform:

$$y = \sigma(z) = \sigma(\mathbf{w}^\top \mathbf{x} + b)$$

Fun Fact

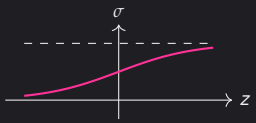
Without activation functions, a 1000-layer network is just... a single matrix multiplication wearing a disguise.

Common Activation Functions

Sigmoid

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

Range: $(0, 1)$



Tanh

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

Range: $(-1, 1)$

ReLU (Most Popular!)

$$\text{ReLU}(z) = \max(0, z)$$

Range: $[0, \infty)$

Key Takeaway

ReLU is fast, simple, and works!
"If in doubt, use ReLU."

The Universal Approximation Theorem

Universal Approximation (Cybenko, 1989)

A feedforward neural network with a single hidden layer containing a finite number of neurons can approximate any continuous function on compact subsets of $\mathbb{R}^n$ to arbitrary accuracy.

Fun Fact

"Given enough neurons, a neural network can approximate ANY function!"

The catch? "Enough" might be astronomically large.

Implications

- NNs are *universal function approximators*
- Existence theorem (not constructive)
- Depth helps: deeper = more efficient

Learning = Finding Good Weights

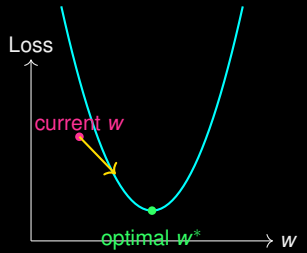
The Learning Problem

Given:

- Data: $\{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^{N}$
- Network architecture

Find: weights $\mathbf{W}$ such that

$$f_{\mathbf{W}}(\mathbf{x}^{(i)}) \approx y^{(i)} \quad \forall i$$



Minimize loss by adjusting weights

Key Takeaway

Training = searching for weights that minimize prediction error
Tool: **Gradient Descent** (uses derivatives we learned!)

What Can Neural Networks Do?

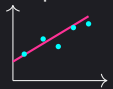
Classification

Is this a cat or dog?

Separate classes

Regression

Predict house price



Continuous output

Generation

Create new images/text

Create content

Fun Fact

Neural networks power: image recognition, language translation, game playing, drug discovery, art generation, and... whatever GPT does when nobody's looking.

Key Takeaways: Neural Networks Introduction

Key Takeaway

1. A **neuron** computes: $y = \sigma(\mathbf{w}^\top \mathbf{x} + b)$
2. Components:
   - → **Weights** ($\mathbf{w}$): learned parameters
   - → **Bias** (b): threshold/offset
   - → **Activation** (σ): adds non-linearity
3. **ReLU** is the most common activation: $\max(0, z)$
4. **Universal Approximation**: NNs can approximate any function
5. **Learning** = finding weights that minimize error
6. Applications: classification, regression, generation

Next: Building deeper networks with multiple layers!

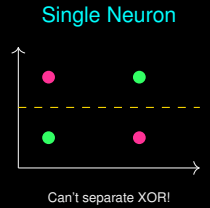
10. Neural Network Architectures

Fun Fact

"One neuron is smart. Many neurons together are... either genius or chaos. Let's aim for genius."

Why go deeper?

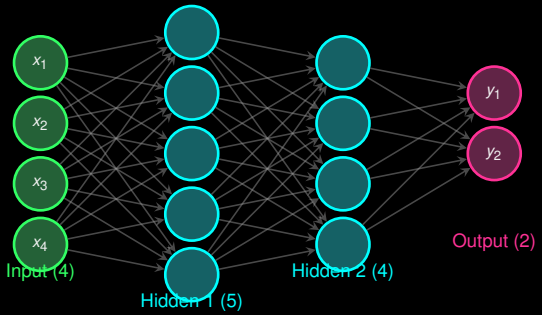
- Single neuron: linear decision boundary
- Multiple neurons: complex patterns
- More layers: hierarchical features



MLP

A **Multi-Layer Perceptron** consists of:

- **Input layer**: receives data
- **Hidden layers**: intermediate computations
- **Output layer**: produces predictions



Layer-wise Computation

Layer Computation

For layer l with n_{l-1} inputs and n_l outputs:

$$\mathbf{a}^{[l]} = \sigma\left(W^{[l]}\mathbf{a}^{[l-1]} + \mathbf{b}^{[l]}\right)$$

Where:

- $W^{[l]} \in \mathbb{R}^{n_l \times n_{l-1}}$: weight matrix
- $\mathbf{b}^{[l]} \in \mathbb{R}^{n_l}$: bias vector
- $\mathbf{a}^{[l]} \in \mathbb{R}^{n_l}$: activations (output)

Dimensions matter!

Input: $\mathbf{x} \in \mathbb{R}^4$

$W^{[1]} \in \mathbb{R}^{5\times 4}$

$\mathbf{a}^{[1]} = \sigma(W^{[1]}\mathbf{x} + \mathbf{b}^{[1]}) \in \mathbb{R}^5$

Key Takeaway

Matrix multiplication handles all neurons in a layer simultaneously!

Vectorization = speed

Notation Convention

Standard Notation

L	Total number of layers
$n^{[l]}$	Number of neurons in layer l
$W^{[l]}$	Weights from layer $l-1$ to l
$\mathbf{b}^{[l]}$	Biases for layer l
$\mathbf{z}^{[l]}$	Pre-activation: $W^{[l]}\mathbf{a}^{[l-1]} + \mathbf{b}^{[l]}$
$\mathbf{a}^{[l]}$	Post-activation: $\sigma(\mathbf{z}^{[l]})$
$\mathbf{a}^{[0]} = \mathbf{x}$	Input
$\mathbf{a}^{[L]} = \hat{\mathbf{y}}$	Output (prediction)

Fun Fact

Superscript $[l]$ = layer number, not exponent!
$W^{[2]}$ means "weights of layer 2", not "W squared"

Full Forward Pass

Forward Propagation

For a network with L layers:

1. Input: $\mathbf{a}^{[0]} = \mathbf{x}$
2. For $l = 1, 2, \ldots, L$:

$$\mathbf{z}^{[l]} = W^{[l]}\mathbf{a}^{[l-1]} + \mathbf{b}^{[l]}$$
$$\mathbf{a}^{[l]} = \sigma_l(\mathbf{z}^{[l]})$$

3. Output: $\hat{\mathbf{y}} = \mathbf{a}^{[L]}$

Width vs Depth

Width

Width = number of neurons per layer

Wide network

More width:

- More parameters per layer
- Can memorize more patterns
- Risk of overfitting

Depth

Depth = number of layers

Deep network

More depth:

- Hierarchical features
- More expressive (often)
- Harder to train

Fun Fact

"Deep Learning" = lots of layers. Who would have guessed?

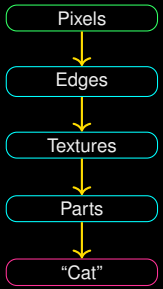
What Do Hidden Layers Learn?

Feature Hierarchy

Each layer learns increasingly abstract features:

Image Recognition Example:

1. Layer 1: Edges, gradients
2. Layer 2: Textures, patterns
3. Layer 3: Parts (eyes, wheels)
4. Layer 4+: Objects, concepts



Key Takeaway

Hidden layers = learned feature extractors. They transform raw data into useful representations!

Output Layer Design

Binary Classification

1 output neuron with sigmoid

$$P(y = 1|\mathbf{x}) = \sigma(z) \in (0, 1)$$

Is it a cat? Yes/No

Regression

1 output neuron (no activation or linear)

$$\hat{y} = z \in \mathbb{R}$$

Predict house price

Multi-class Classification

K **output neurons** with softmax

$$P(y = k|\mathbf{x}) = \frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j}}$$

Outputs sum to 1!
Cat, dog, or bird?

Key Takeaway

Softmax = “soft” version of argmax
Converts scores to probabilities

Counting Parameters

Parameter Count

For a fully connected layer from n_{in} to n_{out}:

$$\text{Parameters} = n_{out} \times n_{in} + n_{out} = n_{out}(n_{in} + 1)$$

(weights + biases)

Example: Network [4, 5, 4, 2]

Layer 1: $5 \times 4 + 5 = 25$
Layer 2: $4 \times 5 + 4 = 24$
Layer 3: $2 \times 4 + 2 = 10$
Total: 59 parameters

Fun Fact

GPT-3 has 175 **billion** parameters.
Our example: 59.
We all start somewhere!

Key Takeaways: Network Architectures

Key Takeaway

1. **MLP**: Input $\rightarrow$ Hidden layers $\rightarrow$ Output
2. **Layer computation**: $\mathbf{a}^{[l]} = \sigma(\mathbf{W}^{[l]}\mathbf{a}^{[l-1]} + \mathbf{b}^{[l]})$
3. **Forward pass**: Sequentially apply all layers
4. **Width** vs **Depth**: Different trade-offs
5. **Hidden layers learn features** at increasing abstraction
6. **Output layer design**:
   - $\rightarrow$ Binary: sigmoid (1 output)
   - $\rightarrow$ Multi-class: softmax (K outputs)
   - $\rightarrow$ Regression: linear (1 output)
7. **Parameters**: Weights + Biases per layer

Next: How does information flow forward through the network?

11. Forward Propagation

Forward Propagation: Data's Journey

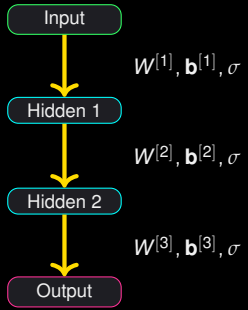
Fun Fact

"Forward propagation is like a game of telephone, except each person does math before passing the message."

The Big Picture:

1. Input enters the network
2. Each layer transforms it
3. Output emerges at the end

Direction: Input → Output

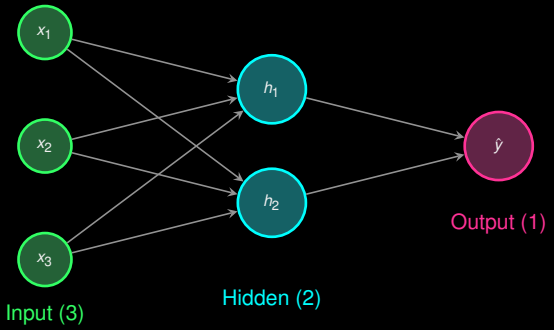


Concrete Example: A 2-Layer Network

Network Configuration

Architecture: $3 \to 2 \to 1$ (input $\to$ hidden $\to$ output)

- Input: $\mathbf{x} = (0.5, 0.8, -0.3)^T$
- Hidden layer: 2 neurons, ReLU activation
- Output layer: 1 neuron, sigmoid activation



Step 1: Hidden Layer Computation

Given weights and biases:

$$W^{[1]} = \begin{pmatrix} 0.2 & 0.4 & -0.5 \\ -0.3 & 0.1 & 0.2 \end{pmatrix}, \quad \mathbf{b}^{[1]} = \begin{pmatrix} 0.1 \\ -0.2 \end{pmatrix}$$

Compute pre-activation:

$$\begin{aligned} \mathbf{z}^{[1]} &= W^{[1]}\mathbf{x} + \mathbf{b}^{[1]} \\ &= \begin{pmatrix} 0.2 & 0.4 & -0.5 \\ -0.3 & 0.1 & 0.2 \end{pmatrix} \begin{pmatrix} 0.5 \\ 0.8 \\ -0.3 \end{pmatrix} + \begin{pmatrix} 0.1 \\ -0.2 \end{pmatrix} \\ &= \begin{pmatrix} 0.1 + 0.32 + 0.15 + 0.1 \\ -0.15 + 0.08 - 0.06 - 0.2 \end{pmatrix} = \begin{pmatrix} 0.67 \\ -0.33 \end{pmatrix} \end{aligned}$$

Step 1 (continued): Apply Activation

Pre-activation: $\mathbf{z}^{[1]} = (0.67, -0.33)^T$

Apply ReLU:

$$\mathbf{a}^{[1]} = \text{ReLU}(\mathbf{z}^{[1]}) = \begin{pmatrix} \max(0, 0.67) \\ \max(0, -0.33) \end{pmatrix} = \begin{pmatrix} 0.67 \\ 0 \end{pmatrix}$$

Key Takeaway

The negative value becomes 0!
ReLU “kills” negative activations.

Step 2: Output Layer Computation

Output layer weights and biases:

$$W^{[2]} = \begin{pmatrix} 0.6 & -0.4 \end{pmatrix}, \quad b^{[2]} = 0.3$$

Compute pre-activation:

$$\begin{aligned} z^{[2]} &= W^{[2]}\mathbf{a}^{[1]} + b^{[2]} \\ &= \begin{pmatrix} 0.6 & -0.4 \end{pmatrix} \begin{pmatrix} 0.67 \\ 0 \end{pmatrix} + 0.3 \\ &= 0.402 + 0 + 0.3 = 0.702 \end{aligned}$$

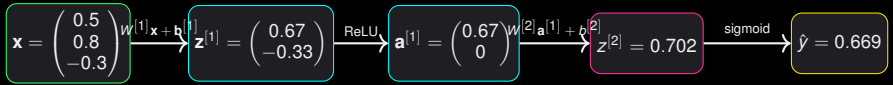
Apply sigmoid:

$$\hat{y} = \sigma(z^{[2]}) = \frac{1}{1 + e^{-0.702}} \approx \boxed{0.669}$$

Key Takeaway

Final prediction: 66.9% probability for class 1

Forward Pass Summary



Fun Fact

From 3 numbers in to 1 number out. That's forward propagation!
(All the matrix multiplication happens behind the scenes)

Batching: Multiple Samples at Once

Batch Forward Pass

Instead of one input $\mathbf{x}$, process m samples simultaneously:

$$X = \begin{pmatrix} | & | & & | \\ \mathbf{x}^{(1)} & \mathbf{x}^{(2)} & \cdots & \mathbf{x}^{(m)} \\ | & | & & | \end{pmatrix} \in \mathbb{R}^{n \times m}$$

Layer computation (vectorized):

$$Z^{[l]} = W^{[l]} A^{[l-1]} + \mathbf{b}^{[l]}, \quad A^{[l]} = \sigma(Z^{[l]})$$

Why batch?

- GPU parallelization
- Faster training
- Better gradient estimates

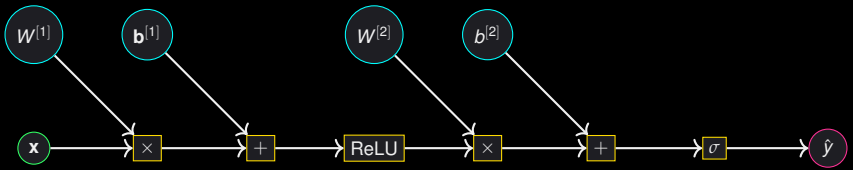
Typical batch sizes

32, 64, 128, 256...
(powers of 2 for efficiency)

Computational Graph

A **computational graph** represents the network as a directed graph where:

- Nodes = operations or variables
- Edges = data flow



Key Takeaway

PyTorch and TensorFlow build these graphs automatically!
Used for automatic differentiation (backprop).

Forward Pass in PyTorch

PyTorch Implementation

```
import torch
import torch.nn as nn

class SimpleNet(nn.Module):
def __init__(self):
super().__init__()
self.hidden = nn.Linear(3, 2)
self.output = nn.Linear(2, 1)
self.relu = nn.ReLU()
self.sigmoid = nn.Sigmoid()

def forward(self, x):
x = self.relu(self.hidden(x))
x = self.sigmoid(self.output(x))
return x

model = SimpleNet()
x = torch.tensor([0.5, 0.8, -0.3])
y_pred = model(x)
```

Key Takeaways: Forward Propagation

Key Takeaway

1. **Forward propagation** = data flows input $\rightarrow$ output
2. Each layer: $\mathbf{a}^{[l]} = \sigma(\boldsymbol{W}^{[l]}\mathbf{a}^{[l-1]} + \mathbf{b}^{[l]})$
3. Steps per layer:
   - $\rightarrow$ Linear: $\mathbf{z} = \boldsymbol{W}\mathbf{a} + \mathbf{b}$
   - $\rightarrow$ Activation: $\mathbf{a} = \sigma(\mathbf{z})$
4. **Batching**: Process multiple samples simultaneously
5. **Computational graph**: Framework builds automatically
6. Forward pass produces **prediction** $\hat{y}$

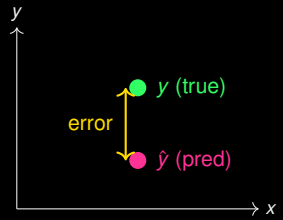
Next: How do we measure if the prediction is any good? Loss functions!

12. Loss Functions

Loss Functions: Measuring Mistakes

Fun Fact

"The loss function is like a brutally honest friend. It tells you exactly how wrong you are — numerically."



The Goal:

- Prediction: $\hat{y}$
- True value: y
- Loss: How far off is $\hat{y}$ from y?

Loss vs Cost

Loss Function

Loss $\mathcal{L}(\hat{y}, y)$ measures error for a **single sample**.
How wrong is this one prediction?

Cost Function

Cost J is the **average loss** over all training samples:

$$J = \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$$

Key Takeaway

Training goal: Minimize the cost function J
Lower cost = better predictions (on average)

Fun Fact

Loss is personal. Cost is the team average.

Mean Squared Error (MSE)

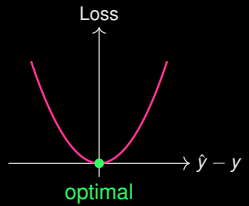
MSE for Regression

$$\mathcal{L}_{\text{MSE}}(\hat{y}, y) = (\hat{y} - y)^2$$

$$J_{\text{MSE}} = \frac{1}{N} \sum_{i=1}^{N} (\hat{y}^{(i)} - y^{(i)})^2$$

Properties:

- Always ≥ 0
- Zero when $\hat{y} = y$
- Penalizes large errors MORE (squared!)
- Smooth, differentiable



Parabola centered at 0

Mean Absolute Error (MAE)

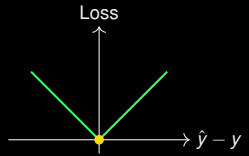
MAE for Regression

$$\mathcal{L}_{\text{MAE}}(\hat{y}, y) = |\hat{y} - y|$$

$$J_{\text{MAE}} = \frac{1}{N} \sum_{i=1}^{N} |\hat{y}^{(i)} - y^{(i)}|$$

MSE vs MAE:

	MSE	MAE
Outliers	Sensitive	Robust
Gradient	Smooth	Discontinuous at 0
Use	Default	Noisy data



V-shaped (not smooth at 0)

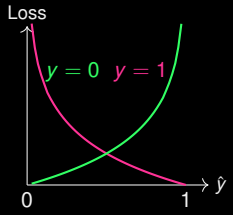
BCE for Binary Classification

For $y \in \{0, 1\}$ and $\hat{y} \in (0, 1)$ (probability):

$$\mathcal{L}_{\text{BCE}}(\hat{y}, y) = -\left[y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})\right]$$

Intuition:

- If $y = 1$: Loss $= -\log(\hat{y}) \rightarrow$ want $\hat{y} \rightarrow 1$
- If $y = 0$: Loss $= -\log(1 - \hat{y}) \rightarrow$ want $\hat{y} \rightarrow 0$



Why Cross-Entropy for Classification?

The Problem with MSE for Classification

Using MSE with sigmoid output:

- Gradient $\to 0$ when sigmoid saturates
- Very slow learning when $\hat{y} \approx 0$ or $\hat{y} \approx 1$
- "Vanishing gradient" problem

Cross-Entropy Solves This

The log in BCE cancels the exp in sigmoid!

$$\frac{\partial \mathcal{L}_{\text{BCE}}}{\partial z} = \hat{y} - y$$

Gradient is **linear** in error — no vanishing!

Key Takeaway

Rule of thumb: Use cross-entropy for classification, MSE for regression.

Categorical Cross-Entropy

For K classes with one-hot encoded $\mathbf{y}$ and softmax output $\hat{\mathbf{y}}$:

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

Since $\mathbf{y}$ is one-hot (only one $y_k = 1$):

$$\mathcal{L}_{\text{CE}} = -\log(\hat{y}_c) \quad \text{where } c \text{ is the true class}$$

Example: True class = 2 (out of 3)

$$\mathbf{y} = (0, 1, 0)$$
$$\hat{\mathbf{y}} = (0.1, 0.7, 0.2)$$
$$\mathcal{L} = -\log(0.7) \approx 0.357$$

Loss Function Summary

Task	Loss	Output Act.	Formula
Regression	MSE	None/Linear	$(\hat{y} - y)^2$
Regression	MAE	None/Linear	$\lvert\hat{y} - y\rvert$
Binary Class.	BCE	Sigmoid	$-y \log \hat{y} - (1 - y) \log(1 - \hat{y})$
Multi-class	CE	Softmax	$-\sum_k y_k \log \hat{y}_k$

Fun Fact

Choosing the right loss function is like choosing the right tool:

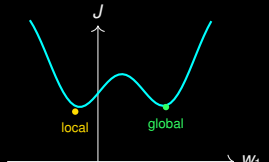
- Hammer (MSE) for nails (regression)
- Screwdriver (CE) for screws (classification)

The Loss Landscape

Visualization

The cost function $J(\mathbf{W})$ defines a "landscape" over parameter space.

- Valleys = good parameters
- Peaks = bad parameters
- Training = finding valleys



Non-convex landscape

Key Takeaway

Neural network loss landscapes are:

- High-dimensional (millions of parameters!)
- Non-convex (multiple minima)
- Surprisingly well-behaved (saddle points, not local minima)

Key Takeaways: Loss Functions

Key Takeaway

1. **Loss** = error for one sample, **Cost** = average over dataset
2. **MSE**: $(\hat{y} - y)^2$ — regression, penalizes outliers
3. **MAE**: $|\hat{y} - y|$ — regression, robust to outliers
4. **Binary Cross-Entropy**: $-y \log \hat{y} - (1 - y) \log(1 - \hat{y})$
   $\rightarrow$ Use with sigmoid output for binary classification
5. **Categorical Cross-Entropy**: $-\sum y_k \log \hat{y}_k$
   $\rightarrow$ Use with softmax output for multi-class
6. Match loss function to task and output activation!

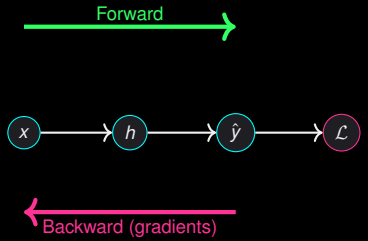
Next: How do we minimize the loss? Backpropagation!

13. Backpropagation

Backpropagation: Learning from Mistakes

Fun Fact

"Backprop is like tracing your steps back through a maze, but instead of finding where you went wrong, you're finding WHO is responsible for the mistake."



The Key Insight:

- Forward: Compute predictions
- Backward: Compute gradients
- Chain rule does ALL the work!

Gradient Descent: The Big Picture

Gradient Descent Update

To minimize $J(\mathbf{W})$, update parameters:

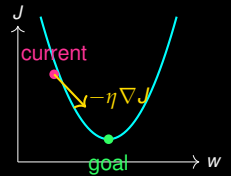
$$W_{new} = W_{old} - \eta \frac{\partial J}{\partial W}$$

where η is the **learning rate**.

We Need Gradients!

$$\frac{\partial J}{\partial W^{[l]}}, \quad \frac{\partial J}{\partial \mathbf{b}^{[l]}}$$

For EVERY parameter in EVERY layer.
How? **Backpropagation!**



The Chain Rule: Heart of Backprop

Chain Rule

If $y = f(g(x))$, then:

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

Multiply the derivatives along the path!

Example: $y = (2x + 1)^2$
Let $g = 2x + 1$, so $y = g^2$

$$\frac{dy}{dg} = 2g = 2(2x + 1)$$
$$\frac{dg}{dx} = 2$$
$$\frac{dy}{dx} = 2(2x + 1) \cdot 2 = 4(2x + 1)$$

Fun Fact

Chain rule = "blame propagation." Each step shares responsibility for the final error!

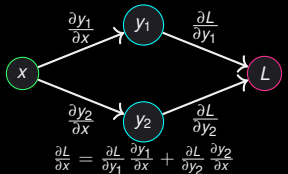
Multivariate Chain Rule

If $L = L(y)$ and $y = y(x_1, x_2, \ldots, x_n)$:

$$\frac{\partial L}{\partial x_i} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial x_i}$$

If there are multiple paths from x to L:

$$\frac{\partial L}{\partial x} = \sum_{\text{all paths}} \frac{\partial L}{\partial y_j} \cdot \frac{\partial y_j}{\partial x}$$



Backprop Example: Setup

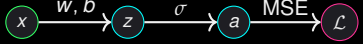
Simple network: $x \to$ hidden $\to$ output $\to$ loss

Forward equations:

$$z = wx + b$$

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

$$\mathcal{L} = (a - y)^2$$



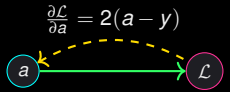
Goal

Compute: $\frac{\partial \mathcal{L}}{\partial w}$ and $\frac{\partial \mathcal{L}}{\partial b}$
Then update: $w \leftarrow w - \eta \frac{\partial \mathcal{L}}{\partial w}$

Backprop Step 1: Output Layer

Start at the end: $\mathcal{L} = (a - y)^2$

$$\frac{\partial \mathcal{L}}{\partial a} = 2(a - y)$$



Info

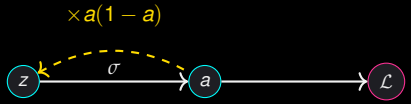
This gradient tells us: "How much does $\mathcal{L}$ change if we nudge a?"
If $a > y$: positive gradient (decrease a!)
If $a < y$: negative gradient (increase a!)

Backprop Step 2: Through Activation

Chain through sigmoid: $a = \sigma(z)$

Sigmoid derivative: $\frac{da}{dz} = \sigma(z)(1 - \sigma(z)) = a(1 - a)$

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



Key Takeaway

Multiply by local gradient $a(1 - a)$ at each step!

Backprop Step 3: To Parameters

Final step: $z = wx + b$

$$\frac{\partial z}{\partial w} = x$$

$$\frac{\partial z}{\partial b} = 1$$

Full gradients:

$$\boxed{\frac{\partial \mathcal{L}}{\partial w} = \frac{\partial \mathcal{L}}{\partial z} \cdot x = 2(a - y) \cdot a(1 - a) \cdot x}$$

$$\boxed{\frac{\partial \mathcal{L}}{\partial b} = \frac{\partial \mathcal{L}}{\partial z} \cdot 1 = 2(a - y) \cdot a(1 - a)}$$

Success

Done! We can now update w and b using gradient descent.

Backpropagation for L Layers

Initialize: $\delta^{[L]} = \nabla_a \mathcal{L} \odot \sigma'(\mathbf{z}^{[L]})$
For $l = L, L-1, \ldots, 1$:

$$\frac{\partial \mathcal{L}}{\partial W^{[l]}} = \delta^{[l]} (\mathbf{a}^{[l-1]})^T$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{[l]}} = \delta^{[l]}$$

$$\delta^{[l-1]} = (W^{[l]})^T \delta^{[l]} \odot \sigma'(\mathbf{z}^{[l-1]})$$

Fun Fact

$\delta^{[l]}$ = "error signal" at layer l
It flows backward, getting transformed at each layer!

Why Backprop is Efficient

Naive Approach

Compute $\frac{\partial \mathcal{L}}{\partial w}$ separately for each weight.
Cost: $O(W \cdot N)$ forward passes
For 1M weights: 1 million forward passes!

Backprop

One forward + one backward pass.
Cost: $O(2)$ passes total
Same cost regardless of # weights!

Key Takeaway

Backprop reuses intermediate computations!
The "error" $\delta^{[l]}$ computed once serves all weights in that layer.

PyTorch Does It For You!

PyTorch (and other frameworks) compute gradients automatically.

```
import torch
x = torch.tensor([0.5, 0.8, -0.3], requires_grad=True)
w = torch.tensor([0.2, 0.4, -0.5], requires_grad=True)
b = torch.tensor(0.1, requires_grad=True)
z = torch.dot(w, x) + b
a = torch.sigmoid(z)
y_true = torch.tensor(1.0)
loss = (a - y_true) ** 2
loss.backward()   # Backward pass - ONE LINE!
print(w.grad)   # dL/dw for each component
```

Key Takeaways: Backpropagation

Key Takeaway

1. **Backprop** computes $\frac{\partial \mathcal{L}}{\partial W}$ for all parameters
2. **Chain rule** is the key: multiply local gradients
3. Flow: Start at loss, work backward through network
4. Error signal $\delta^{[l]}$ propagates backward
5. **Efficient**: $O(1)$ passes, not $O(\text{params})$
6. **Frameworks handle it**: `loss.backward()` does everything!
7. Key derivatives to know:
   - → Sigmoid: $\sigma'(z) = \sigma(z)(1 - \sigma(z))$
   - → ReLU: $\text{ReLU}'(z) = \mathbf{1}_{z>0}$
   - → MSE: $\frac{\partial}{\partial \hat{y}}(\hat{y} - y)^2 = 2(\hat{y} - y)$

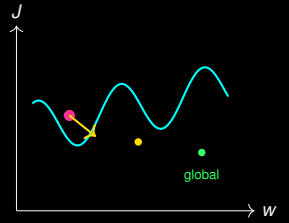
Next: Using gradients to optimize — Training algorithms!

14. Optimization Algorithms

Optimization: Finding the Best Weights

Fun Fact

"Gradient descent is like hiking down a mountain in the fog. You can only feel the slope beneath your feet and hope you're going the right way."



The Challenge:

- Loss landscape is high-dimensional
- Non-convex (many local minima)
- We only see local gradients

Vanilla Gradient Descent

Gradient Descent Update

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta J(\theta_t)$$

where η is the **learning rate**.

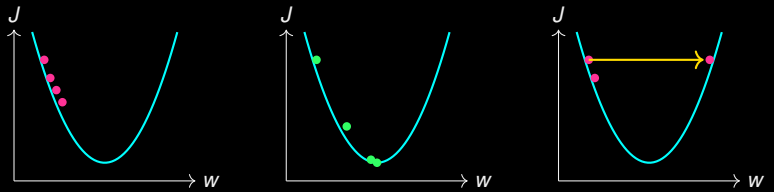
Types by batch size:

- **Batch GD**: All data at once
- **Stochastic GD**: One sample
- **Mini-batch GD**: Subset (common!)

Problems

- Learning rate too high $\rightarrow$ diverge
- Learning rate too low $\rightarrow$ slow
- Same rate for all parameters
- Gets stuck in saddle points

The Learning Rate Dilemma



η too small
(very slow)

η just right
(converges)

η too large
(oscillates/diverges)

Key Takeaway

Typical values: $\eta \in [10^{-4}, 10^{-1}]$
Start with 0.001 and adjust based on training curves.

Momentum: Building Speed

SGD with Momentum

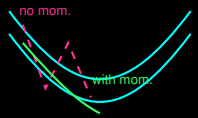
$$v_t = \beta v_{t-1} + \nabla_\theta J(\theta_t)$$
$$\theta_{t+1} = \theta_t - \eta v_t$$

Typical: $\beta = 0.9$

Intuition

Like a ball rolling downhill:

- Builds up velocity
- Carries through flat regions
- Dampens oscillations



Fun Fact

Momentum = memory of past gradients. “Keep going in the general direction!”

RMSprop: Adaptive Learning Rates

RMSprop (Hinton)

$$s_t = \beta s_{t-1} + (1-\beta)(\nabla_\theta J)^2$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{s_t + \epsilon}} \nabla_\theta J$$

Typical: $\beta = 0.9$, $\epsilon = 10^{-8}$

Key Insight

Divide by running average of gradient magnitudes.

- Large gradients $\rightarrow$ smaller steps
- Small gradients $\rightarrow$ larger steps

Info

Adapts per-parameter!
Different learning rates for different weights based on their gradient history.

Adam: The Best of Both Worlds

Adam (Adaptive Moment Estimation)

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla_\theta J \quad \text{(momentum)}$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2)(\nabla_\theta J)^2 \quad \text{(RMSprop)}$$
$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1-\beta_2^t} \quad \text{(bias correction)}$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t$$

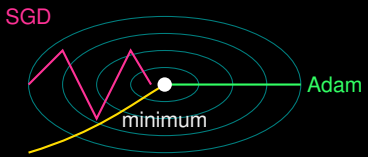
Default hyperparameters:

- $\eta = 0.001$
- $\beta_1 = 0.9$ (momentum)
- $\beta_2 = 0.999$ (RMSprop)
- $\epsilon = 10^{-8}$

Success

Adam is the default choice!
Works well out of the box for most problems.

Optimizer Comparison

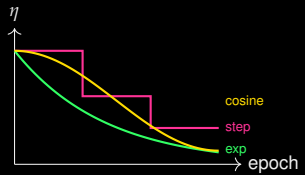


	Speed	Stability	Tuning
SGD	Medium	Low	Hard
Momentum	Fast	Medium	Medium
RMSprop	Fast	High	Easy
Adam	Fast	High	Easy

Learning Rate Decay

Decrease η during training for fine-tuning:

- **Step decay**: $\eta_t = \eta_0 \cdot \gamma^{\lfloor t/s \rfloor}$
- **Exponential**: $\eta_t = \eta_0 \cdot e^{-\lambda t}$
- **Cosine annealing**: $\eta_t = \eta_{\min} + \frac{1}{2}(\eta_0 - \eta_{\min})(1 + \cos(\frac{t\pi}{T}))$



Key Takeaway

Warmup: Start with small η, gradually increase, then decay.

Optimizers in PyTorch

PyTorch Implementation

```
import torch.optim as optim

model = MyNetwork()

optimizer = optim.Adam(model.parameters(), lr=0.001)
# or:  optim.SGD(model.parameters(), lr=0.01, momentum=0.9)

for epoch in range(num_epochs):
for x, y in dataloader:
optimizer.zero_grad()   # Clear gradients
y_pred = model(x)     # Forward pass
loss = criterion(y_pred, y)
loss.backward()       # Compute gradients
optimizer.step()       # Update parameters
```

Key Takeaways: Optimization

Key Takeaway

1. **Gradient Descent**: $\theta \leftarrow \theta - \eta \nabla J$
2. **Mini-batch**: Balance between speed and stability
3. **Momentum**: Accumulate velocity, smooth updates
4. **RMSprop**: Adapt learning rate per parameter
5. **Adam**: Combines momentum + RMSprop
   - $\rightarrow$ Default choice: $\eta = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$
6. **Learning rate schedule**: Start high, decay over time
7. Key workflow: `zero_grad()` → `forward` → `backward` → `step()`

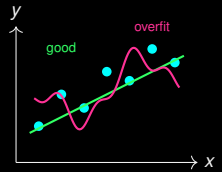
Next: Preventing overfitting with regularization!

15. Regularization Techniques

Regularization: Keeping Models Humble

Fun Fact

"A model that memorizes the training data is like a student who only memorizes answers — useless on a test with new questions!"



The Overfitting Problem:

- Training loss: LOW (YES)
- Test loss: HIGH (NO)
- Model memorized, didn't learn!

Bias-Variance Tradeoff

Decomposition of Error

$$\text{Error} = \text{Bias}^2 + \text{Variance} + \text{Irreducible Noise}$$

High Bias (Underfitting)

Model too simple

- Misses patterns
- Train error: HIGH
- Test error: HIGH

High Variance (Overfitting)

Model too complex

- Fits noise
- Train error: LOW
- Test error: HIGH

Key Takeaway

Regularization reduces variance at the cost of slightly higher bias.
Goal: Find the sweet spot!

L2 Regularization (Weight Decay)

L2 Regularization

Add penalty for large weights:

$$J_{\text{reg}} = J_{\text{original}} + \frac{\lambda}{2} \sum_l \| W^{[l]} \|_F^2$$

where $\| W \|_F^2 = \sum_{i,j} W_{ij}^2$ (Frobenius norm)

Effect on gradient:

$$\frac{\partial J_{\text{reg}}}{\partial W} = \frac{\partial J}{\partial W} + \lambda W$$

Update becomes:

$$W \leftarrow W(1 - \eta\lambda) - \eta \nabla_W J$$

Weights "decay" toward zero

Why it works

- Penalizes complex models
- Encourages small weights
- Smooths decision boundary
- Reduces sensitivity to noise

L1 Regularization (Lasso)

L1 Regularization

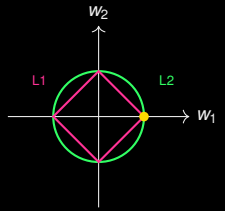
$$J_{\text{reg}} = J_{\text{original}} + \lambda \sum_{l} \|W^{[l]}\|_1$$

where $\|W\|_1 = \sum_{i,j} |W_{ij}|$

L1 vs L2

L1 promotes sparsity!

- Drives some weights to exactly 0
- Feature selection built-in
- Simpler models

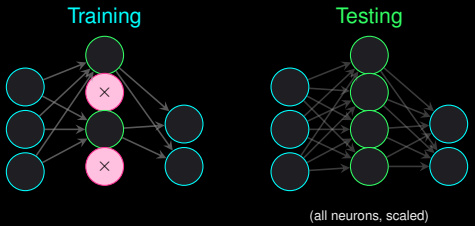


L1 tends to hit corners (sparse)

Dropout (Srivastava et al., 2014)

During training, randomly set each neuron's output to 0 with probability p:

$$\tilde{a}_i = \begin{cases} 0 & \text{with probability } p \\ \frac{a_i}{1-p} & \text{with probability } 1-p \end{cases}$$



Why Dropout Works

Intuitions

- **Ensemble effect**: Training many sub-networks
- **Redundancy**: Can't rely on any single feature
- **Co-adaptation prevention**: Features must be useful alone

Typical Values

- Input layer: $p = 0.2$
- Hidden layers: $p = 0.5$
- Output layer: No dropout

Higher p = more regularization

Fun Fact

Dropout is like studying for an exam knowing some of your brain cells will randomly fail. You learn to be robust!

Batch Normalization (Ioffe & Szegedy, 2015)

Normalize activations across the batch:

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

Then scale and shift:

$$y_i = \gamma \hat{x}_i + \beta$$

where γ and β are learnable parameters.

Benefits:

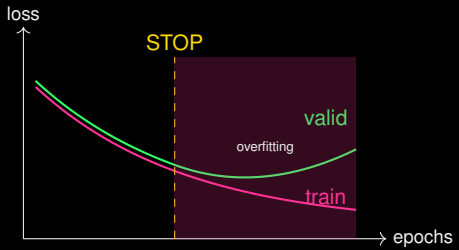
- Stabilizes training
- Allows higher learning rates
- Mild regularization effect
- Reduces internal covariate shift

Key Takeaway

Apply BatchNorm **before** or **after** activation.
Typical placement: after linear, before ReLU.

Early Stopping

Stop training when validation loss starts increasing.



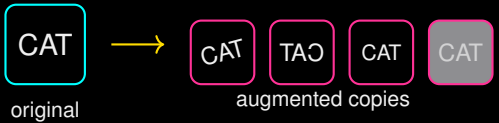
Key Takeaway

Patience: Wait k epochs before stopping (to avoid noise).
Save model at best validation loss!

Data Augmentation

Create more training data by applying transformations:

- Images: rotation, flip, crop, color jitter
- Text: synonym replacement, back-translation
- Audio: pitch shift, time stretch, noise injection



Fun Fact

The same cat is still a cat whether it's rotated, flipped, or darker!

Regularization Summary

Technique	Effect	When to Use
L2 (Weight Decay)	Shrinks weights	Almost always
L1 (Lasso)	Sparse weights	Feature selection
Dropout	Random deactivation	Deep networks
Batch Norm	Normalize activations	Deep networks
Early Stopping	Stop training early	Always
Data Augmentation	More training data	Images, audio

Key Takeaway

Best practice: Combine multiple techniques!

- L2 regularization (almost always)
- Dropout (hidden layers)
- BatchNorm (deep networks)
- Early stopping (monitor validation)
- Data augmentation (when possible)

Key Takeaways: Regularization

Key Takeaway

1. **Overfitting**: Low train error, high test error
2. **Regularization** trades bias for lower variance
3. **L2/L1**: Add weight penalty to loss
   - → L2: Shrinks all weights
   - → L1: Drives some to zero (sparse)
4. **Dropout**: Randomly disable neurons (training only!)
5. **BatchNorm**: Normalize activations per batch
6. **Early stopping**: Monitor validation loss
7. **Data augmentation**: Create more training data
8. Combine multiple techniques for best results

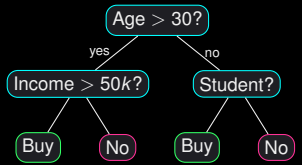
Next: Decision Trees — a different approach to learning!

16. Decision Trees

Decision Trees: Asking the Right Questions

Fun Fact

"Decision trees are like a game of 20 questions, except the computer picks the questions AND answers them!"



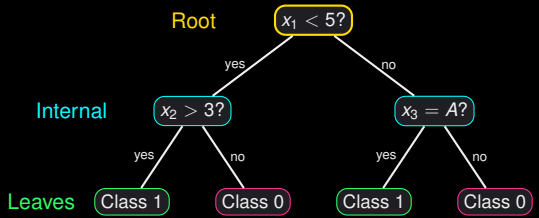
Key Ideas:

- Split data with yes/no questions
- Each split = one decision
- Leaves = final predictions

Anatomy of a Decision Tree

Decision Tree Components

- **Root**: Top node (first question)
- **Internal nodes**: Decision points (questions)
- **Branches**: Answers (yes/no or thresholds)
- **Leaves**: Final predictions



How to Choose Splits: Information Gain

Entropy

Measure of impurity/uncertainty:

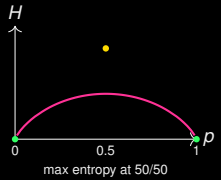
$$H(S) = -\sum_{c=1}^{C} p_c \log_2(p_c)$$

where p_c = proportion of class c in set S.

Information Gain

$$IG(S, A) = H(S) - \sum_{v \in A} \frac{|S_v|}{|S|} H(S_v)$$

Choose split that maximizes IG!



Key Takeaway

Low entropy = pure node. High entropy = mixed node.

Alternative: Gini Impurity

Gini Impurity

$$Gini(S) = 1 - \sum_{c=1}^{C} p_c^2$$

Example:
If S has 70% class A, 30% class B:

$$Gini = 1 - (0.7^2 + 0.3^2) = 1 - 0.58 = 0.42$$

If S is 100% class A:

$$Gini = 1 - 1^2 = 0$$

Entropy vs Gini

- Similar results in practice
- Gini slightly faster (no log)
- Gini: default in sklearn
- Entropy: information-theoretic

Fun Fact

Gini = probability of misclassifying a randomly chosen sample if it was randomly labeled.

Building a Decision Tree (ID3/CART)

Recursive Algorithm

BuildTree(S):

1. If all samples in S have same class: return Leaf(class)
2. If no features left: return Leaf(majority class)
3. Find best feature/threshold to split on (max IG or min Gini)
4. Split S into subsets $S_1, S_2, \ldots$
5. Recursively: BuildTree(S_1), BuildTree(S_2), ...

For continuous features:
Try all thresholds: $x_i < t$ vs $x_i \geq t$
Choose t with best split.

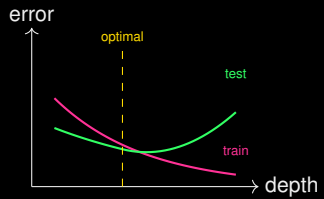
For categorical features:
Try all subsets or one-vs-rest splits.

Overfitting: The Tree's Curse

Deep Trees Overfit!

Without limits, a tree can have:

- One leaf per training sample
- 100% training accuracy
- Terrible test accuracy



Solution: Pruning

Limit tree growth:

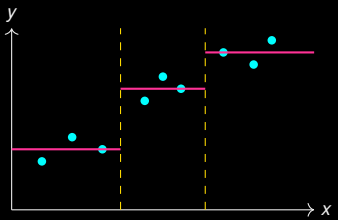
- Max depth
- Min samples per leaf
- Min samples to split

Regression Tree

For continuous targets, use **variance reduction**:

$$\text{Var}(S) = \frac{1}{|S|} \sum_{i \in S} (y_i - \bar{y})^2$$

Leaf prediction: mean of samples in that leaf.



Fun Fact

Regression trees = piecewise constant approximation. Square-ish, but it works!

Decision Trees: Pros and Cons

Advantages

- Easy to interpret
- No scaling needed
- Handles mixed features
- Built-in feature selection
- Fast to train
- No assumptions about data

Disadvantages

- Prone to overfitting
- Unstable (small data changes $\rightarrow$ different tree)
- Can't extrapolate
- Biased toward features with many values
- Greedy (local optima)

Key Takeaway

Single trees are limited, but they're the building blocks for powerful **ensemble methods**!

Decision Trees in Scikit-learn

Python Implementation

```
from sklearn.tree import DecisionTreeClassifier, plot_tree
import matplotlib.pyplot as plt
clf = DecisionTreeClassifier(
max_depth=3,      # Prevent overfitting
min_samples_leaf=5,  # Min samples per leaf
criterion='gini'    # or 'entropy'
)
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)
plt.figure(figsize=(12, 8))
plot_tree(clf, filled=True, feature_names=feature_names)
plt.show()
```

Key Takeaways: Decision Trees

Key Takeaway

1. **Decision trees** = recursive binary splits
2. **Split criteria**: Information gain (entropy) or Gini impurity
3. **Entropy**: $H(S) = -\sum p_c \log_2 p_c$
4. **Gini**: $G(S) = 1 - \sum p_c^2$
5. **Regression**: Use variance, predict mean
6. **Overfitting risk**: Always prune!
   $\rightarrow$ Limit depth, min samples per leaf
7. **Pros**: Interpretable, fast, no preprocessing
8. **Cons**: Unstable, overfit easily

Next: Combining trees into powerful ensembles!

17. Ensemble Methods & Boosting

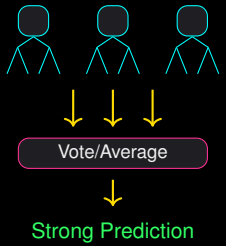
Ensemble Methods: Wisdom of the Crowd

Fun Fact

"One tree might be wrong, but a whole forest? Much harder to fool!"

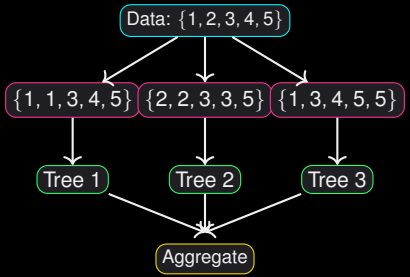
Core Idea:

- Train multiple "weak" models
- Combine their predictions
- Get a "strong" model!



Bagging (Breiman, 1996)

1. Create B bootstrap samples (sample with replacement)
2. Train one model on each bootstrap sample
3. Aggregate predictions:
   - → Classification: majority vote
   - → Regression: average



Random Forest: Bagging + Feature Randomness

Random Forest (Breiman, 2001)

Bagging + random feature selection at each split:

- At each node, consider only m random features
- Typical m: $\sqrt{p}$ for classification, $p/3$ for regression
- This **decorrelates** the trees!

Why Random Features?

Without it:

- All trees use same strong features
- Trees are correlated
- Averaging doesn't help much

With it:

- Trees are diverse
- Errors cancel out!

Hyperparameters

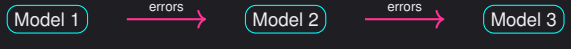
- `n_estimators`: # of trees
- `max_features`: m
- `max_depth`: Tree depth
- `min_samples_leaf`

More trees = better (diminishing returns)

Boosting: Sequential Learning

Boosting Intuition

Train models **sequentially**, each focusing on mistakes of previous ones.



Bagging vs Boosting

Bagging: Parallel, reduce variance
Boosting: Sequential, reduce bias

Fun Fact

Boosting: "You got this wrong? Let me send my friend who's good at exactly that!"

AdaBoost: Adaptive Boosting

AdaBoost (Freund & Schapire, 1997)

1. Initialize sample weights: $w_i = 1/N$
2. For $t = 1, \ldots, T$:
   - 2.1 Train weak learner on weighted data
   - 2.2 Compute weighted error: $\epsilon_t = \sum_{wrong} w_i$
   - 2.3 Compute model weight: $\alpha_t = \frac{1}{2} \ln\left(\frac{1-\epsilon_t}{\epsilon_t}\right)$
   - 2.4 Update sample weights: increase for wrong, decrease for correct
3. Final: $H(x) = \text{sign}\left(\sum_t \alpha_t h_t(x)\right)$

Key Takeaway

Better weak learners get higher α. Misclassified samples get higher weight.

Gradient Boosting: Boosting as Gradient Descent

Gradient Boosting

Each new model fits the **residuals** (gradient of loss):

1. Initialize: $F_0(x) = \arg\min_c \sum_i L(y_i, c)$
2. For $t = 1, \ldots, T$:
   - 2.1 Compute residuals: $r_i = -\frac{\partial L(y_i, F_{t-1}(x_i))}{\partial F_{t-1}}$
   - 2.2 Fit tree h_t to residuals
   - 2.3 Update: $F_t(x) = F_{t-1}(x) + \eta \cdot h_t(x)$

For MSE loss:
Residual = $y_i - F_{t-1}(x_i)$
(actual minus prediction)

Fun Fact

Each tree says: "Here's how much you're off by. Let me fix that!"

XGBoost: Gradient Boosting on Steroids

XGBoost (Chen & Guestrin, 2016)

Key improvements over vanilla gradient boosting:

- **Regularized objective**: $L = \sum_i l(y_i, \hat{y}_i) + \sum_k \Omega(f_k)$
- **Second-order approximation**: Uses Hessian
- **Efficient implementation**: Parallel, cache-aware
- **Handling missing values**: Built-in
- **Column subsampling**: Like random forest

Success

XGBoost often wins Kaggle competitions!
Also: **LightGBM** (Microsoft) and **CatBoost** (Yandex) are popular alternatives.

Ensemble Methods Comparison

	Random Forest	AdaBoost	XGBoost
Training	Parallel	Sequential	Sequential
Reduces	Variance	Bias	Both
Overfitting	Robust	Can overfit	Regularized
Tuning	Easy	Medium	Complex
Interpretable	Somewhat	Somewhat	Less
Speed	Fast	Medium	Fast

Rules of Thumb

- **Start with**: Random Forest (robust, few hyperparams)
- **For maximum performance**: XGBoost/LightGBM
- **For interpretability**: Single tree (with pruning)

Scikit-learn & XGBoost

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import GradientBoostingClassifier
import xgboost as xgb

# Random Forest
rf = RandomForestClassifier(n_estimators=100, max_depth=10)
rf.fit(X_train, y_train)

# Gradient Boosting
gb = GradientBoostingClassifier(n_estimators=100,
learning_rate=0.1)
gb.fit(X_train, y_train)

# XGBoost
xgb_model = xgb.XGBClassifier(n_estimators=100,
learning_rate=0.1, max_depth=6)
xgb_model.fit(X_train, y_train)
```

Key Takeaways: Ensemble Methods

Key Takeaway

1. **Ensemble** = combine multiple weak learners
2. **Bagging**: Train in parallel on bootstrap samples
   → Reduces variance
3. **Random Forest**: Bagging + random feature subsets
   → Decorrelates trees
4. **Boosting**: Train sequentially, focus on errors
   → Reduces bias
5. **AdaBoost**: Weight samples and models
6. **Gradient Boosting**: Fit residuals (gradients)
7. **XGBoost**: Regularized, efficient GB

Next: Statistical Learning Theory — why does any of this work?

18. Statistical Learning Theory

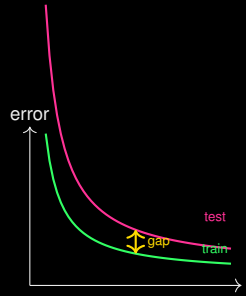
Statistical Learning Theory: Why ML Works

Fun Fact

"We've been fitting models and hoping they work. Let's understand WHY they work (or don't)!"

The Big Questions:

- When can we learn from data?
- How much data do we need?
- Why does training error $\neq$ test error?



Generalization gap decreases with more data

PAC Learning: Probably Approximately Correct

PAC Learning (Valiant, 1984)

A concept class $\mathcal{C}$ is **PAC-learnable** if there exists an algorithm that:
For any distribution $\mathcal{D}$, any $\epsilon > 0$, any $\delta > 0$:
Given m samples from $\mathcal{D}$, the algorithm outputs h such that:

$$P\left[\text{error}(h) \leq \epsilon\right] \geq 1 - \delta$$

with $m = \text{poly}(1/\epsilon, 1/\delta, n, \text{size}(c))$

Translation

Probably: With high probability $(1 - \delta)$
Approximately: Low error $(\leq \epsilon)$
Correct: On new data!

Fun Fact

"Give me enough data, and I'll *probably* give you something *approximately* right!"

Generalization Error Decomposition

Error Types

- **Training error**: Error on training data

$$\hat{R}(h) = \frac{1}{n}\sum_{i=1}^{n} \mathbf{1}[h(x_i) \neq y_i]$$

- **Generalization error**: Expected error on new data

$$R(h) = \mathbb{E}_{(x,y)\sim\mathcal{D}}[\mathbf{1}[h(x) \neq y]]$$

The Gap

$$\underbrace{R(h)}_{\text{test error}} = \underbrace{\hat{R}(h)}_{\text{train error}} + \underbrace{(R(h) - \hat{R}(h))}_{\text{generalization gap}}$$

Generalization Bound (Finite Hypothesis Class)

Finite Hypothesis Bound

For a finite hypothesis class $|\mathcal{H}|$, with probability $\geq 1 - \delta$:

$$R(h) \leq \hat{R}(h) + \sqrt{\frac{\ln |\mathcal{H}| + \ln(1/\delta)}{2n}}$$

Implications:

- More hypotheses $\rightarrow$ larger gap
- More data $n \rightarrow$ smaller gap
- Higher confidence $1/\delta \rightarrow$ larger gap

Problem

Neural networks have infinite hypothesis classes! This bound doesn't directly apply...

VC Dimension: Measuring Model Complexity

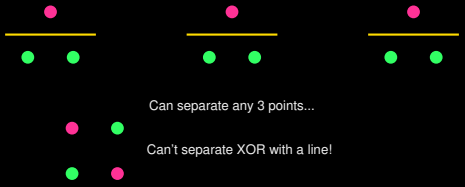
Shattering

A hypothesis class $\mathcal{H}$ **shatters** a set of points if it can realize ALL possible labelings of those points.

VC Dimension

The **VC dimension** of $\mathcal{H}$ is the maximum number of points that can be shattered by $\mathcal{H}$.

Linear classifiers in 2D: VC = 3



VC Generalization Bound

VC Bound

For hypothesis class with VC dimension d, with probability $\geq 1 - \delta$:

$$R(h) \leq \hat{R}(h) + \sqrt{\frac{d(\ln(2n/d) + 1) + \ln(4/\delta)}{n}}$$

Implications

- Higher VC dim $\rightarrow$ worse generalization
- Need $n \gg d$ for good bounds
- Model complexity matters!

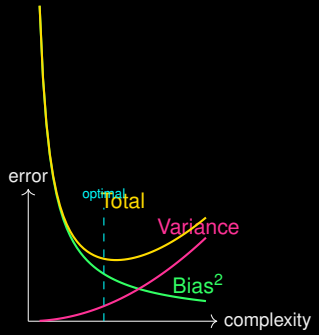
VC Examples

- Linear in $\mathbb{R}^d$: VC = $d + 1$
- Intervals on $\mathbb{R}$: VC = 2
- Axis-aligned rectangles: VC = 4
- Neural nets: Depends on depth/width

MSE Decomposition

For any estimator $\hat{f}$:

$$\mathbb{E}[(y - \hat{f}(x))^2] = \underbrace{\text{Bias}[\hat{f}(x)]^2}_{\text{systematic error}} + \underbrace{\text{Var}[\hat{f}(x)]}_{\text{sensitivity to training}} + \underbrace{\sigma^2}_{\text{irreducible}}$$



No Free Lunch Theorem

No Free Lunch (Wolpert & Macready)

Averaged over ALL possible problems, every learning algorithm performs equally!

$$\sum_f P(d_m|f, A_1) = \sum_f P(d_m|f, A_2)$$

for any algorithms A_1, A_2 and any performance measure.

Fun Fact

"No algorithm is universally better. Some are just better for YOUR problem!"

Implications

- Domain knowledge matters
- Try multiple algorithms
- Inductive bias is crucial

Sample Complexity: How Much Data?

Sample Complexity

Minimum samples m needed to achieve error $\leq \epsilon$ with probability $\geq 1 - \delta$.

For PAC-learnable classes:

$$m \geq \frac{1}{\epsilon}\left(d \ln \frac{1}{\epsilon} + \ln \frac{1}{\delta}\right)$$

where d is VC dimension.

Rule of Thumb

For neural nets, people often use:

$$n \geq 10 \times (\text{\# parameters})$$

(Very rough heuristic!)

Modern Paradox

Deep learning often works with:

$$n \ll (\text{\# parameters})$$

Theory is still catching up!

Key Takeaways: Statistical Learning Theory

Key Takeaway

1. **PAC learning**: Probably Approximately Correct framework
2. **Generalization gap**: Test error - Train error
3. **VC dimension**: Measure of model complexity
   - $\rightarrow$ Max points a model can shatter
4. **Generalization bounds**: Test error bounded by train error + complexity term
5. **Bias-Variance tradeoff**:
   - $\rightarrow$ Simple models: high bias, low variance
   - $\rightarrow$ Complex models: low bias, high variance
6. **No Free Lunch**: No universal best algorithm
7. **Sample complexity**: More complex model $\rightarrow$ more data needed

Next: KL Divergence and Information Theory!

19. KL Divergence & Information Theory

Information Theory: The Math of Surprise

Fun Fact

"Information theory tells us: rare events are surprising, common events are boring. Just like plot twists!"

Why for ML?

- Cross-entropy loss
- Measuring distribution similarity
- Understanding predictions

Information Content

The **information** (surprise) of event with probability p:

$$I(x) = -\log_2(p(x)) \text{ bits}$$

- $p = 1$ (certain): 0 bits
- $p = 0.5$: 1 bit
- $p = 0.01$: 6.64 bits

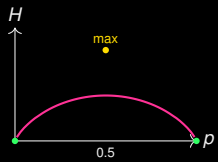
Entropy: Average Surprise

Shannon Entropy

The **entropy** of a distribution P is the expected information:

$$H(P) = -\sum_x p(x) \log p(x) = \mathbb{E}_{x \sim P}[-\log p(x)]$$

Binary entropy:



Properties

- $H \geq 0$ always
- Max when uniform
- Min (= 0) when deterministic
- Units: bits ($\log_2$) or nats (ln)

Cross-Entropy: Comparing Distributions

Cross-Entropy

The **cross-entropy** between true distribution P and predicted distribution Q:

$$H(P, Q) = -\sum_x p(x) \log q(x) = \mathbb{E}_{x \sim P}[-\log q(x)]$$

Interpretation

Average bits needed to encode data from P using code optimized for Q.

If $Q \neq P$: We're using a "wrong" code!

In ML

- P = true labels (one-hot)
- Q = predicted probabilities
- $H(P, Q)$ = cross-entropy loss!

$$\mathcal{L}_{CE} = -\sum_c y_c \log(\hat{y}_c) = H(\mathbf{y}, \hat{\mathbf{y}})$$

KL Divergence: How Different Are Two Distributions?

Kullback-Leibler Divergence

The **KL divergence** from Q to P:

$$D_{KL}(P\|Q) = \sum_x p(x) \log \frac{p(x)}{q(x)} = \mathbb{E}_{x\sim P}\left[\log \frac{p(x)}{q(x)}\right]$$

Key Relationship

$$D_{KL}(P\|Q) = H(P,Q) - H(P)$$

Cross-entropy = Entropy + KL divergence

When minimizing cross-entropy with fixed P, we're minimizing KL divergence!

Properties of KL Divergence

Gibbs' Inequality

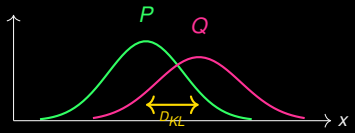
$$D_{KL}(P\|Q) \geq 0$$

with equality iff $P = Q$ (almost everywhere)

Not Symmetric!

$$D_{KL}(P\|Q) \neq D_{KL}(Q\|P)$$

KL divergence is NOT a distance metric!



Fun Fact

$D_{KL}(P\|Q)$ asks: "How surprised is Q by data from P?"

Forward vs Reverse KL

Forward KL: $D_{KL}(P\|Q)$

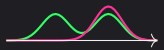
Minimize: $\mathbb{E}_P[\log P - \log Q]$

- Q must cover all of P
- **Mean-seeking**
- Used in variational inference (ELBO)

Reverse KL: $D_{KL}(Q\|P)$

Minimize: $\mathbb{E}_Q[\log Q - \log P]$

- Q can ignore parts of P
- **Mode-seeking**
- Used in policy gradient, GANs



KL Divergence in Machine Learning

1. **Cross-entropy loss**: Minimizes $D_{KL}(\text{true} \| \text{pred})$
2. **Variational Autoencoders (VAE)**:
$$\mathcal{L}_{VAE} = \text{reconstruction} + D_{KL}(q(z|x) \| p(z))$$
3. **Knowledge Distillation**:
$$\mathcal{L}_{KD} = D_{KL}(\text{teacher} \| \text{student})$$
4. **Reinforcement Learning**: Policy optimization bounds
5. **Bayesian inference**: Variational approximations

Key Takeaway

KL divergence is everywhere in modern ML — it measures how "different" one distribution is from another!

Mutual Information: Shared Information

Mutual Information

$$I(X;Y) = D_{KL}(P(X,Y) \| P(X)P(Y)) = H(X) - H(X|Y)$$

How much knowing Y tells us about X.

Properties

- $I(X;Y) \geq 0$
- $I(X;Y) = I(Y;X)$ (symmetric!)
- $I(X;Y) = 0$ iff independent
- $I(X;X) = H(X)$

Info

Used in: feature selection, information bottleneck, representation learning

Information Theory Summary

Quantity	Formula	Meaning
Information	$-\log p(x)$	Surprise of event
Entropy $H(P)$	$\mathbb{E}_P[-\log p]$	Average surprise
Cross-entropy	$\mathbb{E}_P[-\log q]$	Bits using wrong code
KL divergence	$\mathbb{E}_P[\log p/q]$	Distribution difference
Mutual info	$D_{KL}(P_{XY} \| P_X P_Y)$	Shared information

Key Relationships

$$H(P, Q) = H(P) + D_{KL}(P \| Q)$$
$$I(X; Y) = H(X) + H(Y) - H(X, Y)$$
$$D_{KL}(P \| Q) \geq 0 \text{ with equality iff } P = Q$$

Key Takeaways: Information Theory

Key Takeaway

1. **Information**: $-\log p$ — rare events have more info
2. **Entropy**: Average information = uncertainty measure
3. **Cross-entropy**: Using "wrong" distribution for encoding
   - → This is our classification loss!
4. **KL divergence**: Measures distribution difference
   - → Not symmetric, not a true distance
   - → Forward vs reverse have different behaviors
5. **Mutual information**: Shared information between variables
6. Minimizing cross-entropy = minimizing KL from true distribution

Next: The classic MNIST dataset!

20. MNIST Dataset

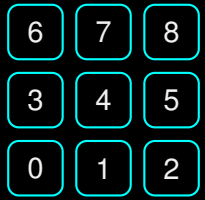
MNIST: The “Hello World” of ML

Fun Fact

“Every ML journey starts with MNIST. It’s tradition. Like ‘Hello World’ for programmers, but with more pixels.”

What is MNIST?

- Handwritten digit images
- Collected from Census Bureau
- Created by Yann LeCun, 1998
- The benchmark for decades



28×28 grayscale images

MNIST Dataset Details

MNIST Specifications

- **Training set**: 60,000 images
- **Test set**: 10,000 images
- **Image size**: 28 × 28 pixels
- **Channels**: 1 (grayscale)
- **Pixel values**: 0-255 (usually normalized)
- **Classes**: 10 (digits 0-9)
- **Input dimension**: 784 (28×28)
- **Output**: One-hot or class label
- **Format**: IDX file format
- **Size**: 12 MB total

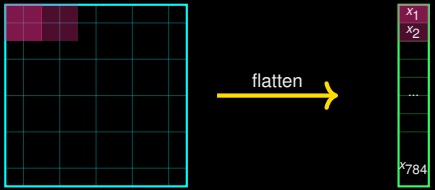
Key Takeaway

As a vector: Each image $\rightarrow$ 784-dimensional vector
Input shape: $(N, 1, 28, 28)$ or $(N, 784)$ depending on model

Image as Data: Flattening

28×28 Image

784-d Vector



Info

Pixel $(i, j) \to$ vector index $28 \cdot i + j$
Row-major ordering: read left-to-right, top-to-bottom

Preprocessing MNIST

Standard Preprocessing

1. **Normalization**: Scale to $[0, 1]$ or standardize

$$x' = \frac{x}{255} \quad \text{or} \quad x' = \frac{x - \mu}{\sigma}$$

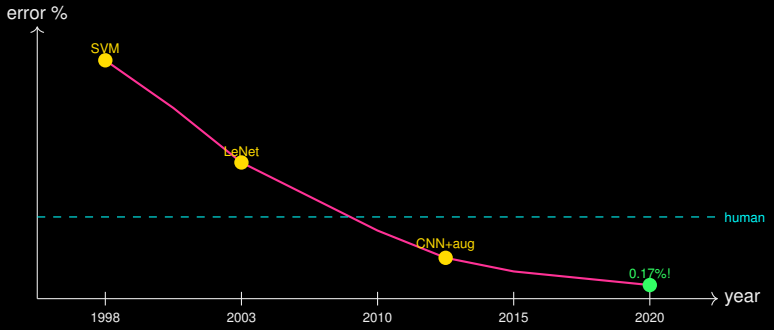
2. **Reshaping**:
   - $\rightarrow$ MLP: $(N, 784)$ — flat vector
   - $\rightarrow$ CNN: $(N, 1, 28, 28)$ — keep spatial structure
3. **Labels**: One-hot encode for cross-entropy

$$y = 3 \rightarrow \mathbf{y} = (0, 0, 0, 1, 0, 0, 0, 0, 0, 0)$$

Fun Fact

MNIST statistics: $\mu \approx 0.1307$, $\sigma \approx 0.3081$
(These numbers are ML folk knowledge at this point!)

MNIST Performance Through History



Info

State-of-art: 0.17% error (99.83% accuracy)
Human performance: 1-2% error (some digits are genuinely ambiguous!)

Why Start with MNIST?

Advantages

- Small and fast to train
- Easy to visualize
- Well-studied baseline
- Works on CPU
- Great for learning
- Available everywhere

Limitations

- “Too easy” for modern methods
- Not representative of real problems
- Grayscale, centered, clean
- Limited variability
- “MNIST-easy” is a term!

Next Steps After MNIST

- **Fashion-MNIST**: Clothes instead of digits (same format)
- **CIFAR-10/100**: 32×32 color images, 10/100 classes
- **ImageNet**: 1000 classes, real images

Loading MNIST in PyTorch

PyTorch Implementation

```
import torch
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

transform = transforms.Compose([
transforms.ToTensor(),
transforms.Normalize((0.1307,), (0.3081,))
])

train_data = datasets.MNIST(root='./data', train=True,
download=True, transform=transform)
test_data = datasets.MNIST(root='./data', train=False,
download=True, transform=transform)

train_loader = DataLoader(train_data, batch_size=64, shuffle=True)
test_loader = DataLoader(test_data, batch_size=64, shuffle=False)
```

A Simple MNIST Classifier

MLP for MNIST

```
import torch.nn as nn

class MNISTClassifier(nn.Module):
def __init__(self):
super().__init__()
self.flatten = nn.Flatten()
self.layers = nn.Sequential(
nn.Linear(784, 256), nn.ReLU(), nn.Dropout(0.2),
nn.Linear(256, 128), nn.ReLU(), nn.Dropout(0.2),
nn.Linear(128, 10)
)
def forward(self, x):
x = self.flatten(x)
return self.layers(x)
# Typical accuracy:  approx 98% with this simple model!
```

Key Takeaways: MNIST

Key Takeaway

1. **MNIST**: 70,000 handwritten digits (60k train, 10k test)
2. **Format**: 28×28 grayscale → 784-dim vector
3. **Preprocessing**:
   - → Normalize: divide by 255 or use $\mu = 0.1307, \sigma = 0.3081$
   - → One-hot encode labels for cross-entropy
4. **Baseline performance**:
   - → Simple MLP: 98%
   - → CNN: 99%
   - → State-of-art: 99.8%+
5. **Perfect for learning** but not representative of real challenges
6. **Graduate to**: Fashion-MNIST, CIFAR-10, ImageNet

Next: Putting it all together — Training Pipeline!

21. Training Pipeline

The Complete Training Pipeline

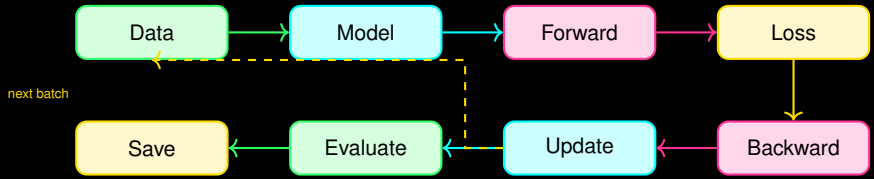
Fun Fact

“We’ve learned all the ingredients. Now it’s time to bake the neural network cake! CAKE”

The Training Recipe

1. **Data**: Load, preprocess, batch
2. **Model**: Define architecture
3. **Loss**: Choose objective function
4. **Optimizer**: Select update rule
5. **Train**: Forward $\rightarrow$ Loss $\rightarrow$ Backward $\rightarrow$ Update
6. **Evaluate**: Test set performance
7. **Save**: Checkpoint best model

Training Pipeline Overview



Info

Epoch: One pass through entire training set
Iteration/Step: One batch update

The Training Loop

PyTorch Training Loop

```
def train_epoch(model, loader, criterion, optimizer, device):
model.train() # Set to training mode
total_loss = 0
correct = 0
for batch_idx, (data, target) in enumerate(loader):
data, target = data.to(device), target.to(device)
optimizer.zero_grad() # Clear gradients
output = model(data) # Forward pass
loss = criterion(output, target)
loss.backward() # Backward pass
optimizer.step() # Update weights
total_loss += loss.item()
pred = output.argmax(dim=1)
correct += (pred == target).sum().item()
return total_loss / len(loader), correct / len(loader.dataset)
```

Training vs Evaluation Mode

`model.train()`

- Dropout **active**
- BatchNorm uses **batch** statistics
- Gradients computed

Use during: **Training**

`model.eval()`

- Dropout **disabled**
- BatchNorm uses **running** statistics
- Can disable gradient computation

Use during: **Validation/Test**

Common Mistake!

Forgetting to switch between train/eval mode is a very common bug!
Always call `model.eval()` before testing and `model.train()` before training.

The Evaluation Loop

PyTorch Evaluation

```
@torch.no_grad() # Disable gradient computation
def evaluate(model, loader, criterion, device):
model.eval() # Set to evaluation mode
total_loss = 0
correct = 0
for data, target in loader:
data, target = data.to(device), target.to(device)
output = model(data)
loss = criterion(output, target)
total_loss += loss.item()
pred = output.argmax(dim=1)
correct += (pred == target).sum().item()
accuracy = correct / len(loader.dataset)
avg_loss = total_loss / len(loader)
return avg_loss, accuracy
```

Complete Training Script

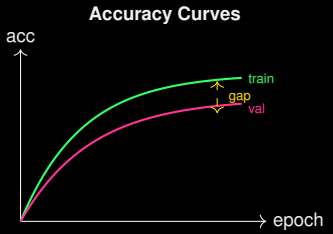
Full Training Pipeline

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = MNISTClassifier().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=5, gamma=0.5)

best_acc = 0
for epoch in range(num_epochs):
train_loss, train_acc = train_epoch(model, train_loader,
criterion, optimizer, device)
val_loss, val_acc = evaluate(model, val_loader, criterion, device)
scheduler.step()
print(f'Epoch {epoch}:  Train Loss={train_loss:.4f}, Val Acc={val_acc:.4f}')

if val_acc > best_acc:
best_acc = val_acc
torch.save(model.state_dict(), 'best_model.pth')
```

Monitoring Training Progress



Watch For

- Val loss increasing → overfitting
- Loss plateau → lower LR
- Loss spikes → LR too high

Gap

Large train-val gap means model memorizing, not learning!

Saving and Loading Models

Model Checkpointing

```
# Save model weights only
torch.save(model.state_dict(), 'model_weights.pth')

# Load weights
model = MNISTClassifier()
model.load_state_dict(torch.load('model_weights.pth'))

# Save complete checkpoint (for resuming training)
checkpoint = {
'epoch': epoch,
'model_state_dict': model.state_dict(),
'optimizer_state_dict': optimizer.state_dict(),
'loss': loss,
'best_acc': best_acc
}
torch.save(checkpoint, 'checkpoint.pth')

# Load and resume
checkpoint = torch.load('checkpoint.pth')
model.load_state_dict(checkpoint['model_state_dict'])
optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
```

Training Best Practices

Tips for Successful Training

1. **Start simple**: Small model, verify it can overfit one batch
2. **Monitor everything**: Loss, accuracy, gradients, learning rate
3. **Use validation set**: Never tune on test set!
4. **Checkpoint regularly**: Save best model by validation metric
5. **Reproducibility**: Set random seeds
6. **Normalize data**: Huge impact on training stability
7. **Learning rate**: Often the most important hyperparameter

Fun Fact

“If in doubt, lower the learning rate.” — Ancient ML Wisdom

Debugging Training Issues

Common Problems

- **Loss = NaN**
  - → LR too high
  - → Divide by zero
  - → Log of zero
- **Loss doesn't decrease**
  - → LR too low
  - → Bug in forward pass
  - → Wrong loss function
- **Overfitting**
  - → Add regularization
  - → More data
  - → Smaller model

Debugging Checklist

1. Can model overfit 1 batch?
2. Are labels correct?
3. Is data normalized?
4. Are gradients flowing?
5. Is loss function appropriate?
6. Is train/eval mode correct?
7. Are shapes matching?

Key Takeaways: Training Pipeline

Key Takeaway

1. **Training loop**: Forward → Loss → Backward → Update
2. **Modes**: Always toggle train/eval appropriately
3. **Evaluation**: Use `@torch.no_grad()` for efficiency
4. **Monitoring**: Track train & val loss/accuracy
5. **Checkpointing**: Save best model, enable resuming
6. **Best practices**:
   - → Start simple, verify overfitting
   - → Never tune on test set
   - → Learning rate is crucial

Congratulations! You've completed Day 0! PARTY

Next: Day 1 — Hands-on implementation and advanced topics!

Day 0 Summary: The Journey So Far

Mathematical Foundations

- Numbers & counting
- Real & complex numbers
- Functions & parameters
- Limits & continuity
- Differentiation
- Integration
- Graph theory

Machine Learning

- Neural network basics
- Forward propagation
- Loss functions
- Backpropagation
- Optimization algorithms
- Regularization
- Decision trees & boosting
- Statistical learning
- Information theory

Success

You now have the foundation to understand, implement, and debug neural networks from scratch!

Thank You!

Questions?