Teaching Machines to See Patterns

A Neural Networks Primer: Why We Needed Each Piece of the Puzzle

NLP Course 2025

From the 1950s mail sorting crisis to ChatGPT: How humanity taught machines to think

Your Journey Through Neural Networks

Where We're Going Today

Act I: The Problem (1943-1969)

- The mail sorting crisis
- First mathematical neurons
- The perceptron revolution
- The XOR catastrophe

Intermission: Understanding the Basics

- How neurons calculate
- Why we need layers
- Following the forward pass

Act II: The Struggles (1980s-1990s)

- Hidden layers save the day
- Backpropagation breakthrough
- Universal approximation proof

Act III: The Revolution (2000s-Present)

- Deep learning explosion
- Modern architectures
- Real-world impact

Epilogue: Your Turn

- Build your first network
- Next steps

Each act builds on the previous - no jumping ahead!

The Core Idea: Neural Networks are Function Approximators

What does this actually mean?

The Problem:

- We have inputs (x)
- We want outputs (y)
- But we don't know the formula!
- Examples:
  - Size → Price
  - Image → Label
  - Text → Sentiment

Traditional Approach:

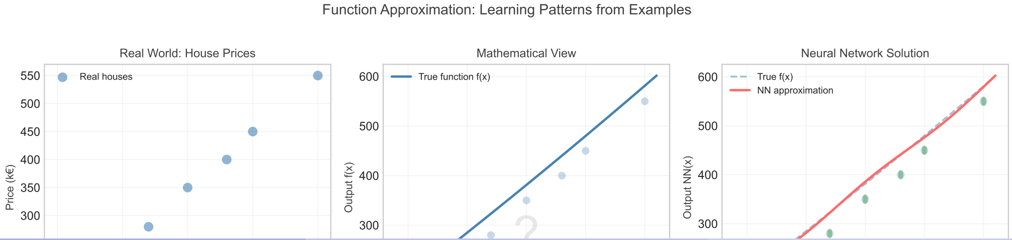
- Guess the formula
- Write explicit rules
- Hope it works
- Problem: Real world is too complex!

Example:
Price = a × Size + b
(Too simple for real data!)

Neural Network Approach:

- Learn from examples
- Build the formula automatically
- Adjust until it fits
- Works for ANY pattern!

Magic:
NN learns: f(x) ≈ y
No formula needed!



How NNs Build Complex Functions from Simple Pieces

The LEGO Principle: Combine Simple Parts to Build Anything

The Building Blocks:

1. **Individual Neurons:**
   - Each neuron = simple decision
   - "Is input ¿ threshold?"
   - Outputs: on/off (or smooth version)
2. **Combine Neurons:**
   - Add their outputs
   - Weight their importance
   - Create complex shapes
3. **Stack Layers:**
   - First layer: simple features
   - Next layer: combinations
   - Final layer: complete function

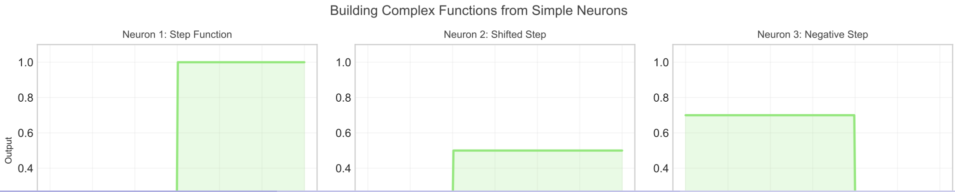
Real-World Analogy:

Making a Cake (Complex) from Ingredients (Simple):

- Flour → Basic structure
- Sugar → Sweetness level
- Eggs → Binding agent
- Mix right amounts → Perfect cake!

In Neural Networks:

- Neuron 1 → Detects edges
- Neuron 2 → Detects curves
- Neuron 3 → Detects colors
- Combine all → Recognize faces!



The Most Important Theorem in Deep Learning (Cybenko, 1989)

The Theorem (Plain English):

> *"A neural network with enough neurons can approximate* **ANY** *continuous function to* **ANY** *desired accuracy"*

What This Means:

- Universal: Works for any smooth pattern
- Guaranteed: Not hoping, but proving
- Practical: Just add more neurons!

The Catch:

- How many neurons? Could be millions
- How to find weights? That's training
- How long to train? That's the art

Intuitive Proof:

Think of it like pixel art:

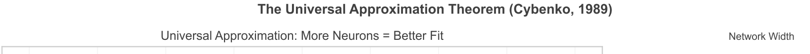
1. With 4 pixels: Very blocky image
2. With 100 pixels: Recognizable
3. With 10,000 pixels: Photo-realistic
4. With infinite pixels: Perfect!

Same with neurons:

1. Few neurons: Rough approximation
2. More neurons: Better fit
3. Many neurons: Nearly perfect
4. Infinite neurons: Exact function!

Why This Matters:

We don't need different architectures for different problems - just one universal tool that adapts!



1950s: The Mail Sorting Crisis

The Challenge:

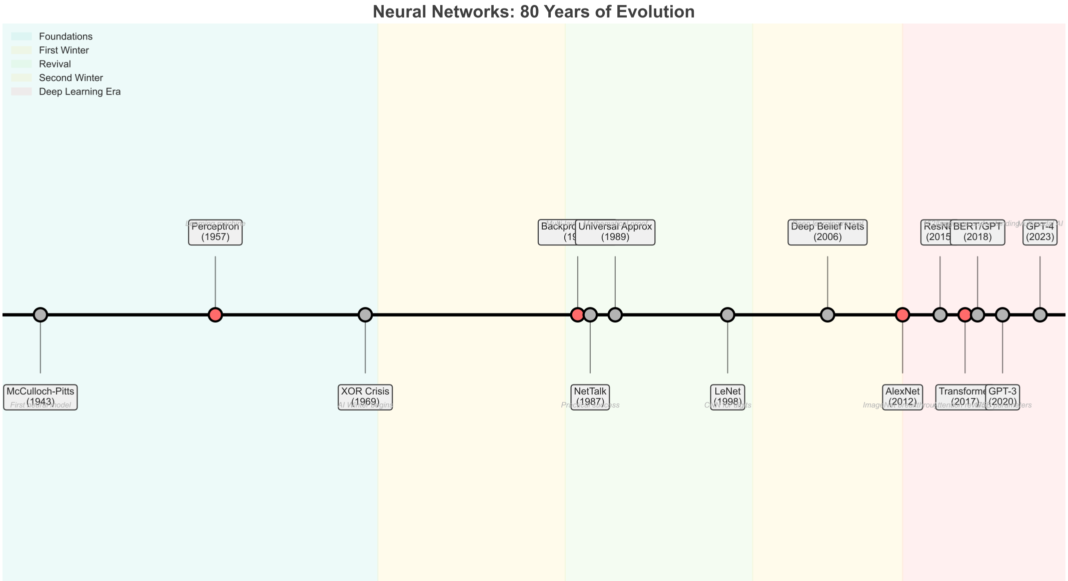
- 150 million letters per day
- Hand-written addresses
- Human sorters: slow, expensive, error-prone
- Traditional programming: useless

Why Traditional Code Failed:

- Can't write rules for every handwriting style
- Too many variations of each letter
- Context matters: "I" vs "l" vs "1"
- This wasn't computation—it was pattern recognition

This problem would take 40 years to solve properly

80 Years of Neural Networks: The Complete Journey



Why Can't We Just Write Rules?

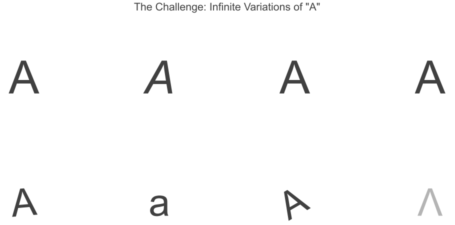
Problem: Recognize the Letter "A"

Traditional Approach (Failed):

```
if (has_triangle_top AND
    has_horizontal_bar AND
    two_diagonal_lines) {
  return "A"
}
```

But what about...

- Handwritten A's?
- Different fonts?
- Rotated A's?
- Partial A's?



Just for the letter "A", we'd need thousands of rules!

The breakthrough: What if machines could learn patterns like children do?

1943: The First Spark - McCulloch & Pitts

The Birth of Computational Neuroscience

The Revolutionary Paper:

- "A Logical Calculus of Ideas Immanent in Nervous Activity"
- First mathematical model of neurons
- Proved: Networks can compute ANY logical function
- Inspired von Neumann's computer architecture

Key Insight:

- Neurons = Logic gates
- Brain = Computing machine
- Thinking = Computation

The Model:

- Binary neurons (0 or 1)
- Threshold activation
- Fixed connections
- No learning yet!

Historical Impact:

- Founded field of neural networks
- Influenced cybernetics movement
- Set stage for AI research
- "The brain is a computer" metaphor

14 years later, Rosenblatt would add the missing piece: learning

1957: The First Learning Machine - The Perceptron

Frank Rosenblatt's Radical Idea: Neurons That Learn

Beyond McCulloch-Pitts:

- Adjustable weights (not fixed!)
- Learning from mistakes
- Physical machine built (Mark I)
- Could recognize simple patterns

The Hardware:

- 400 photocells (20×20 "retina")
- 512 motor-driven potentiometers
- Weights adjusted by electric motors
- Took 5 minutes to learn patterns

Mathematical Model:

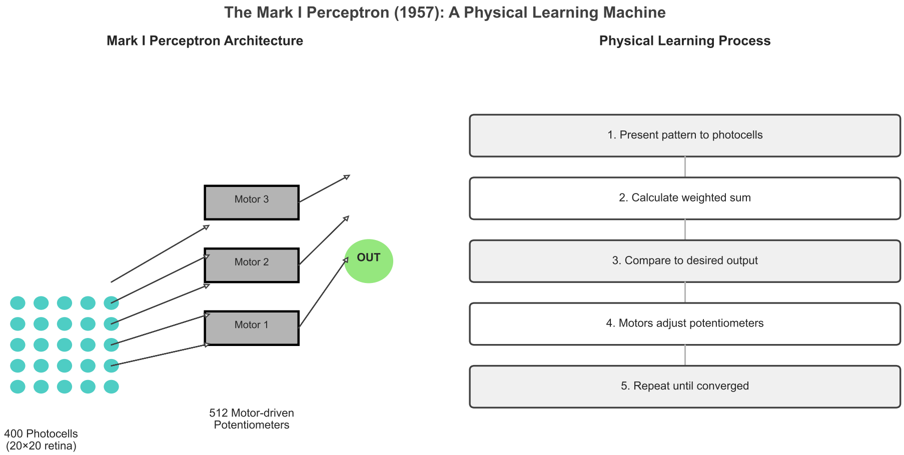
- Inputs: $x_1, x_2, ..., x_n$
- Weights: $w_1, w_2, ..., w_n$
- Sum: $z = \sum_{i=1}^{n} w_i x_i + b$
- Output: $y = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if } z \leq 0 \end{cases}$

In plain words: Each input gets a vote (weight). We add up all votes plus a bias. If total is positive, output 1; otherwise 0.

Learning Rule: If wrong: $w_i = w_i + \eta \cdot error \cdot x_i$

The New York Times, 1958: "The Navy revealed the embryo of an electronic computer that will be able to walk, talk, see, write, reproduce itself and be conscious of its existence."

The Mark I Perceptron: A Physical Learning Machine



The first neural network wasn't software—it was a room-sized machine with motors and photocells

Let's Understand How This Actually Works

We've Seen the History...

- McCulloch-Pitts invented the neuron
- Rosenblatt made it learn
- The perceptron was born

Now Let's See the Science:

- How does a neuron calculate?
- What does learning mean?
- Why was XOR so hard?

Next 5 slides: Hands-on calculations and exercises
Get your pencil ready - we're going to work through real examples!

Don't worry - we'll return to the story once you understand the basics

Understanding Check: Can You Answer These?

Let's Make Sure We're Together

Quick Questions:

1. Why couldn't traditional programming solve mail sorting?
2. What does a weight represent in simple terms?
3. Why do we need the bias term?
4. What was revolutionary about Rosenblatt's perceptron?

Think About It:

- A weight is like the importance/trust we give to each input
- Bias shifts our decision threshold
- Learning = adjusting these weights
- The perceptron was the first machine that could learn!

Try It Yourself: Draw a simple perceptron with 2 inputs. Label the weights, bias, and output. What would the weights be to compute AND logic?

If any of these are unclear, revisit the previous slides before continuing

Problem: Learn OR function (output 1 if ANY input is 1)

Training Data:

x_1	x_2	Output
0	0	0
0	1	1
1	0	1
1	1	1

The Perceptron:

$$z = w_1 \cdot x_1 + w_2 \cdot x_2 + b$$

$$\text{output} = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if } z \leq 0 \end{cases}$$

In plain words: Multiply first input by first weight, second input by second weight, add bias, check if positive

Learning Process:

1. Start with random weights
2. For each example:
   - Calculate output
   - If wrong: adjust weights
   - If correct: keep weights
3. Repeat until all correct

Final Solution: $w_1 = 1$, $w_2 = 1$, $b = -0.5$

Success! But this was just the beginning...

Let's Calculate Together: Is This Email Spam?

A Real Perceptron Calculation You Can Follow

The Email:

"FREE money! Click here NOW for amazing offer!!!"

Our Features (Inputs):

- x_1 = Has "FREE"? = 1
- x_2 = Has "money"? = 1
- x_3 = Many "!"? = 1
- x_4 = From friend? = 0

Learned Weights:

- w_1 = +3 (FREE is very spammy)
- w_2 = +2 (money is suspicious)
- w_3 = +2 (!!! is aggressive)
- w_4 = -5 (friends are trusted)
- b = -2 (threshold)

Let's Calculate:

$$
\begin{aligned}
z &= w_1 \cdot x_1 + w_2 \cdot x_2 + w_3 \cdot x_3 + w_4 \cdot x_4 + b \\
&= 3 \cdot 1 + 2 \cdot 1 + 2 \cdot 1 + (-5) \cdot 0 + (-2) \\
&= 3 + 2 + 2 + 0 - 2 \\
&= 5
\end{aligned}
$$

Decision:

- $z = 5 > 0$
- Output = 1 = SPAM!

Try It Yourself: What if this email WAS from a friend ($x_4 = 1$)? Recalculate! Would it still be spam?

Answer: $z = 5 - 5 = 0$, borderline!

This is exactly how early spam filters worked - and why they failed on clever spam

Breaking Down the Math Symbols

Inputs and Weights:

- x_i = input value (what we see)
- w_i = weight (importance/strength)
- b = bias (threshold adjuster)

The Computation:

$$z = \sum_{i=1}^{n} w_i x_i + b$$

This means:

- Multiply each input by its weight
- Add them all up
- Add the bias

Real Example:

Should I go outside?

Factor	Value	Weight
Sunny?	1	+2
Raining?	0	-3
Weekend?	1	+1

$$z = (1 \times 2) + (0 \times -3) + (1 \times 1) = 3$$

Decision: $z > 0$, so YES!

This simple math would evolve into deep learning

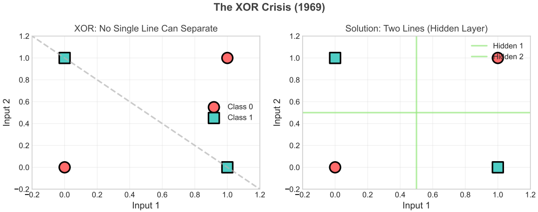
Minsky & Papert's Devastating Discovery

XOR (Exclusive OR):

x_1	x_2	Output
0	0	0
0	1	1
1	0	1
1	1	0

The Problem:

- Can't draw a single line to separate
- Perceptron only learns linear boundaries
- Real-world problems are non-linear!



Impact:

- Funding dried up
- "AI Winter" begins
- Neural networks abandoned

The field would be dormant for over a decade...

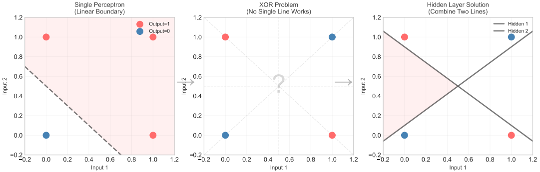
Let's See Why We Need Hidden Layers

Problem 1: Spam Detection (Easy)

- Has many spam words? $\rightarrow$ SPAM
- Has few spam words? $\rightarrow$ NOT SPAM
- One line (threshold) works!

Problem 2: Cat or Dog Photo (Hard)

- Small + fluffy? Could be either!
- Large + smooth? Could be either!
- Pointy ears + whiskers? $\rightarrow$ Cat
- Floppy ears + wet nose? $\rightarrow$ Dog
- Need multiple feature detectors!



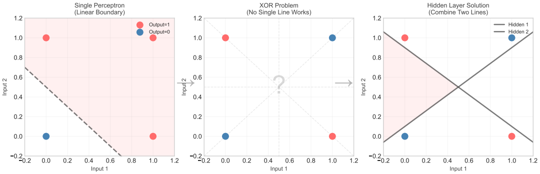
The Solution:

1. First layer: Multiple detectors
   - Detector 1: "Has cat features?"
   - Detector 2: "Has dog features?"
2. Second layer: Combine detections
   - If cat features ¿ dog features $\rightarrow$ Cat

This is why deep learning works: each layer builds more complex detectors from simpler ones

Visual Bridge: Why XOR Needs Hidden Layers

From Simple Lines to Complex Boundaries



Single Perceptron = One Line:

- Can only draw straight boundaries
- Works for OR, AND
- Fails for XOR, real problems

Hidden Layers = Multiple Lines:

- Each hidden neuron draws a line
- Output combines these lines
- Can create any shape!

Common Confusion: Hidden layers don't "hide" anything - they're called hidden because we don't directly set their values. They learn what features to detect!

This insight took 13 years to discover and implement properly

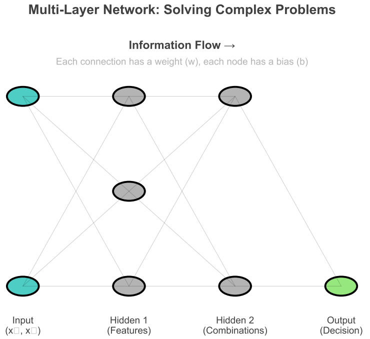
1980s: The Hidden Layer Revolution

The Insight:

- Stack multiple layers!
- First layer: detect simple features
- Hidden layer: combine features
- Output layer: final decision

Solving XOR:

- Hidden neuron 1: Is it (0,1)?
- Hidden neuron 2: Is it (1,0)?
- Output: OR of hidden neurons



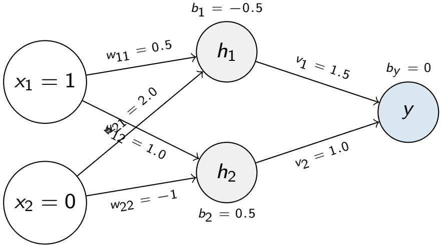
New Architecture:

- Input layer: raw data
- Hidden layer(s): feature extraction

Forward Pass Playground: Let's Calculate Through a Network!

Follow the Numbers Step by Step

Simple 2-Layer Network:



Your Task: Calculate the output!

Try It Yourself: Fill in the blanks as we go:

- $h_1 = ?$
- $h_2 = ?$
- $y = ?$

This is exactly what happens millions of times per second in deep learning

Step 1: Calculate Hidden Neurons

$$\begin{aligned} h_1 &= \text{ReLU}(1 \cdot 0.5 + 0 \cdot 2.0 - 0.5) \\ &= \text{ReLU}(0.5 + 0 - 0.5) \\ &= \text{ReLU}(0) = 0 \end{aligned}$$

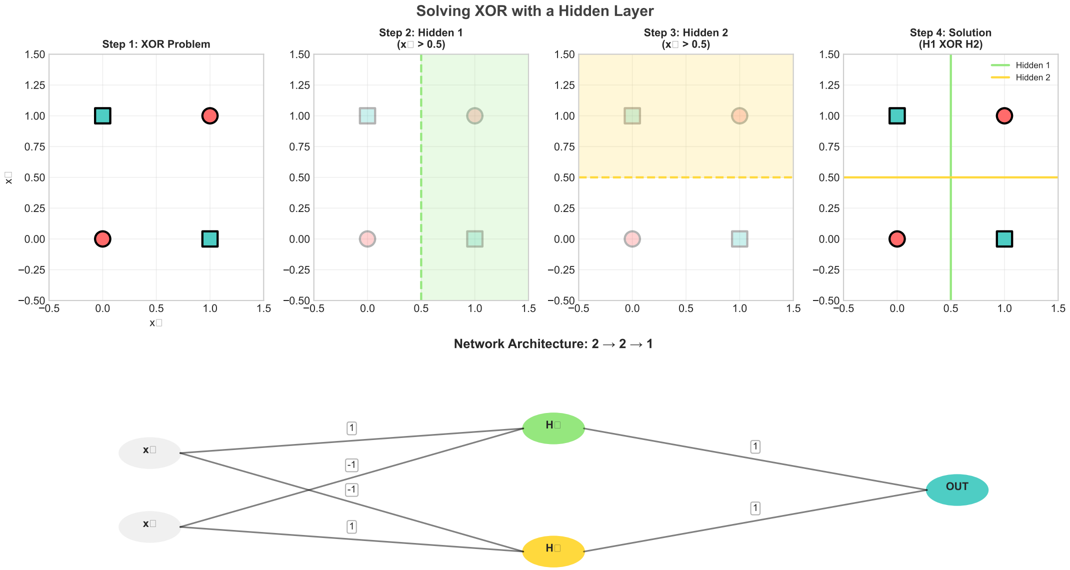
$$\begin{aligned} h_2 &= \text{ReLU}(1 \cdot 1.0 + 0 \cdot (-1) + 0.5) \\ &= \text{ReLU}(1.0 + 0 + 0.5) \\ &= \text{ReLU}(1.5) = 1.5 \end{aligned}$$

Step 2: Calculate Output

$$\begin{aligned} y &= 0 \cdot 1.5 + 1.5 \cdot 1.0 + 0 \\ &= 0 + 1.5 + 0 = 1.5 \end{aligned}$$

The network output is 1.5!

Solving XOR: Step-by-Step with Hidden Layers



Learning in Action: Weight Evolution

../figures/weight_evolution.pdf

1986: Backpropagation - Teaching Networks to Learn

Rumelhart, Hinton, Williams: The Learning Algorithm

The Problem:

- Perceptron learning only works for 1 layer
- How to adjust hidden layer weights?
- No direct error signal for hidden neurons

The Solution:

- Propagate error backwards!
- Each layer gets blame for output error
- Use calculus (chain rule) to distribute blame

The Algorithm:

1. Forward: Calculate output
2. Compare: Find error
3. Backward: Distribute blame
4. Update: Adjust all weights

In plain words: Like a teacher marking an essay: finds the final error, then traces back to see which paragraphs, sentences, and words caused it

Impact:

- Finally could train deep networks!
- Neural networks reborn
- Foundation of all modern AI

This algorithm runs billions of times to train ChatGPT

1987: NetTalk - Networks Learn to Speak

Sejnowski & Rosenberg: The First Viral NN Demo

The Challenge:

- Convert written text to speech
- English is irregular (tough, though, through)
- Rule-based systems had 1000s of exceptions

The Network:

- 7×29 input (7-letter window)
- 80 hidden neurons
- 26 output phonemes
- Trained overnight on DEC workstation

The Magic:

- Started: Random babbling
- Hour 1: Vowel-consonant patterns
- Hour 5: Recognizable words
- Hour 10: 95% accuracy!

Hidden Neurons Learned:

- Vowel detectors
- Consonant clusters
- Word boundaries
- Nobody programmed these!

Common Confusion: The network discovered linguistic concepts on its own - features linguists took centuries to identify!

Media sensation: "Computer teaches itself to read aloud overnight"

Cybenko, Hornik: The Ultimate Proof

The Theorem:

A feedforward network with:

- One hidden layer
- Enough neurons
- Non-linear activation

Can approximate ANY continuous function to arbitrary accuracy!

What This Means:

- Neural networks are universal computers
- No function is too complex
- Just need enough neurons and data

Real-World Analogy:

LEGO blocks can build anything:

- Few blocks = rough shape
- Many blocks = detailed model
- Infinite blocks = perfect replica

Same with neurons:

- Few neurons = rough approximation
- Many neurons = good function
- Infinite neurons = exact function

Try It Yourself: Think of any pattern or function. This theorem guarantees a neural network can learn it!

This gave theoretical backing to the neural network revolution

The Vanishing Gradient Problem: Why Deep Was Hard

../figures/vanishing_gradient.pdf

A Simple Example You Can Calculate

Tiny Network:

- Input: $x = 1$
- Weight: $w = 2$
- Output: $y = w \times x = 2$
- Target: $t = 3$
- Error: $E = (y - t)^2 = (2 - 3)^2 = 1$

Question: How should we change w?

Gradient Calculation:

$$\begin{aligned} \frac{\partial E}{\partial w} &= \frac{\partial}{\partial w}(w \cdot x - t)^2 \\ &= 2(w \cdot x - t) \cdot x \\ &= 2(2 - 3) \cdot 1 = -2 \end{aligned}$$

Weight Update:

- Gradient $= -2$ (negative means increase w)
- Learning rate $= 0.1$
- New weight $= w - 0.1 \times (-2) = 2.2$

Check Our Work:

- New output $= 2.2 \times 1 = 2.2$
- New error $= (2.2 - 3)^2 = 0.64$
- Error decreased from 1 to 0.64!

Try It Yourself: What would happen if we used learning rate $= 1$? Calculate the new weight and error. What goes wrong?

This simple calculation, repeated millions of times, trains all neural networks

The Need for Non-Linearity

Problem with Linear:

- Stack of linear layers = still linear!
- $f(g(x)) = (wx + b_1)w' + b_2 = w'wx + \ldots$
- Can't learn complex patterns

Solution: Activation Functions

- Add non-linearity after each layer
- Allows learning complex boundaries
- Different functions for different needs

Common Activation Functions:

- **Sigmoid:** $\sigma(x) = \frac{1}{1+e^{-x}}$
  - Smooth, outputs 0-1
  - Good for probabilities

In plain words: Squashes any input to range 0-1. Large positive becomes 1, large negative becomes 0

- **ReLU:** $f(x) = \max(0, x)$
  - Simple, fast
  - Solves vanishing gradient
- **Tanh:** $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
  - Outputs -1 to 1
  - Zero-centered

ReLU's simplicity revolutionized deep learning in 2011

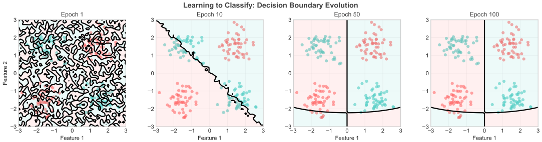
Teaching a Network to Separate Red from Blue Points

The Setup:

- Input: (x, y) coordinates
- Output: Red or Blue class
- Network: $2 \rightarrow 4 \rightarrow 2$ neurons

Training Process:

1. Epoch 1: Random boundary
2. Epoch 10: Rough separation
3. Epoch 50: Good boundary
4. Epoch 100: Perfect fit



What Each Layer Learns:

- Layer 1: Simple boundaries
- Hidden: Combine boundaries
- Output: Final decision

This same principle scales to millions of parameters

1998-2012: From Digits to ImageNet

1998 - LeNet: First Success

- Yann LeCun's CNN for digits
- 32×32 pixels $\rightarrow$ 10 classes
- 60,000 parameters
- Banks adopt for check reading

Key Innovation: Convolutions

- Share weights across image
- Detect features anywhere
- Build complexity layer by layer

2012 - AlexNet: The Revolution

- 1000 ImageNet classes
- 60 million parameters
- GPUs enable training
- Error rate: 26% $\rightarrow$ 16%

What Changed:

- Big Data (millions of images)
- GPU computing (100x faster)
- ReLU activation
- Dropout regularization

This victory ended the second AI winter permanently

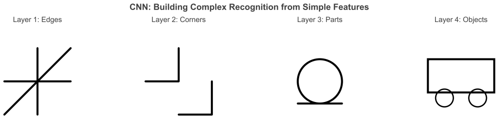
How We Actually Recognize Objects

Human Vision Process:

1. Detect edges
2. Find shapes
3. Identify parts
4. Recognize object

CNN Mimics This:

- Layer 1: Edge detectors
- Layer 2: Corner/curve detectors
- Layer 3: Part detectors
- Layer 4: Object detectors



Key Insight:

- A "wheel detector" works anywhere in image
- Share the same detector across positions
- Reduces parameters dramatically
- Makes network translation-invariant

This is why CNNs dominate computer vision

The Mathematics of Learning: Gradient Descent

Finding the Best Weights: Like Hiking Down a Mountain

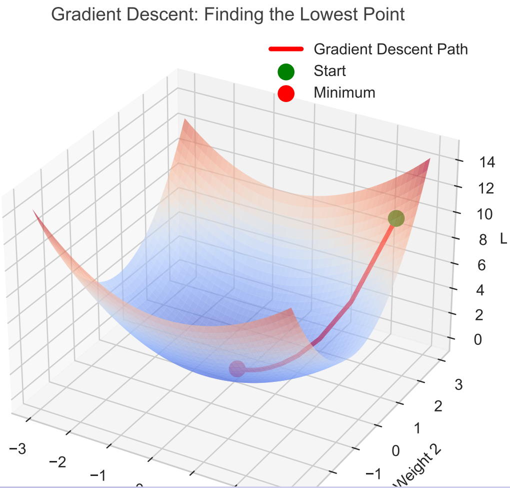
The Optimization Problem:

- Millions of weights to adjust
- Each affects the error
- Need to find best combination

Gradient Descent:

1. Calculate error (loss)
2. Find slope (gradient) for each weight
3. Step downhill: $w = w - \alpha \cdot \nabla L$

In plain words: New weight = old weight - (step size times slope)



Types of Learning: Different Problems, Different Approaches

Supervised Learning:

- Have input-output pairs
- Learn mapping function
- Examples: Classification, Regression

Unsupervised Learning:

- Only have inputs
- Find patterns/structure
- Examples: Clustering, Compression

Reinforcement Learning:

- Learn through trial/error
- Maximize reward signal
- Examples: Games, Robotics

Self-Supervised (Modern):

- Create labels from data itself
- Predict next word, masked words
- Examples: GPT, BERT

Self-supervised learning powers all modern language models

Check Your Understanding: Learning Types

Can You Match These Examples?

Try It Yourself: Match each scenario to a learning type: Supervised, Unsupervised, Reinforcement, Self-Supervised

Scenarios:

1. Teaching a robot to walk by giving rewards for standing
2. Showing 1000 cat photos labeled "cat"
3. Giving GPT text with words masked out
4. Finding groups in customer data

Answers:

1. Reinforcement (trial and error)
2. Supervised (labeled examples)
3. Self-supervised (creates own labels)
4. Unsupervised (finds patterns)

Common Confusion: Self-supervised IS supervised learning - we just create the labels automatically from the data itself!

Understanding these differences helps you choose the right approach

The Overfitting Problem: When Learning Goes Too Far

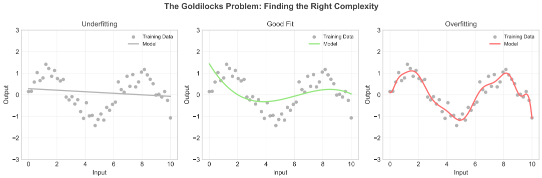
Memorization vs. Understanding

The Problem:

- Network memorizes training data
- Fails on new, unseen data
- Like student memorizing answers

Signs of Overfitting:

- Training accuracy: 99%
- Test accuracy: 60%
- Complex decision boundaries
- High variance



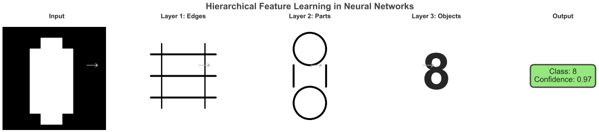
Solutions:

- **More data:** Can't memorize everything
- **Dropout:** Randomly disable neurons
- **Regularization:** Penalize complexity
- **Early stopping:** Stop before overfitting

"With four parameters I can fit an elephant, with five I can make him wiggle his trunk" - von Neumann

How Deep Networks See: Building Features Layer by Layer

From Pixels to Concepts: The Hierarchy of Understanding



What Each Layer Learns:

- **Layer 1:** Edges, colors, gradients
- **Layer 2:** Corners, textures, curves
- **Layer 3:** Parts (eyes, wheels, patterns)
- **Layer 4:** Objects (faces, cars, scenes)
- **Layer 5:** Concepts (identity, style, context)

Why Hierarchy Matters:

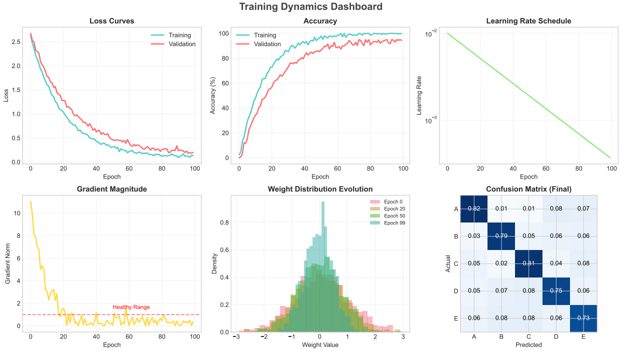
- Reusable features
- Efficient representation
- Transfer learning works
- Mimics visual cortex

Discovered Automatically:

- No manual feature engineering
- Emerges from data
- Different tasks, same hierarchy
- Universal pattern

Each layer combines features from the previous layer into more abstract concepts

Real-Time Monitoring: The Training Dashboard



Key Metrics to Track:

- **Loss Curves:** Training vs validation
- **Accuracy:** How often we're right
- **Learning Rate:** Speed of updates
- **Gradient Norm:** Update magnitude

Warning Signs:

- Gap = Overfitting
- Flat = Learning stopped
- Spikes = Instability
- NaN = Numerical issues

Healthy Training:

- Smooth decrease
- Val follows train
- Gradients stable
- LR decays properly

When to Stop:

- Validation plateaus
- Gap increasing
- Diminishing returns

Modern training requires constant monitoring - it's more art than science

2014-Present: Networks That Changed the World

The Depth Revolution:

- 2014 - VGGNet: 19 layers
- 2015 - ResNet: 152 layers
- 2017 - Transformers: Attention
- 2020 - GPT-3: 175B parameters

Why Depth Matters:

- Each layer = abstraction level
- Deep = complex reasoning
- Hierarchical feature learning

Real-World Impact:

- **Vision:** Self-driving cars
- **Language:** Google Translate
- **Speech:** Siri, Alexa
- **Medicine:** Disease diagnosis
- **Science:** Protein folding

The Scale:

- Billions of parameters
- Trained on internet-scale data
- Months of GPU time
- Emergent abilities appear

We went from recognizing digits to passing the bar exam in 25 years

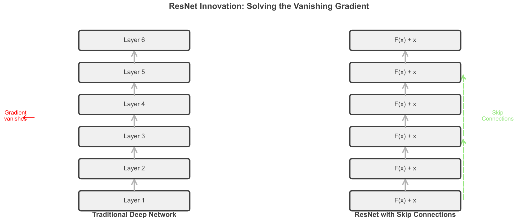
Problem: Networks Couldn't Get Deeper

The Vanishing Gradient:

- Gradients multiply through layers
- Become exponentially small
- Deep layers stop learning
- 20 layers was the limit

The Breakthrough: Skip Connections

- Add input directly to output
- $F(x) + x$ instead of just $F(x)$
- Gradients flow directly backward
- Can train 1000+ layers!



Why It Works:

- Learn residual (difference) only
- Identity mapping is easy default
- Gradients have direct path
- Each layer refines previous result

This simple trick enabled the deep learning revolution

The Internal Covariate Shift Problem

The Issue:

- Each layer's input distribution changes
- As previous layers update
- Makes learning unstable
- Requires tiny learning rates

The Solution:

- Normalize inputs to each layer
- Mean $= 0$, Variance $= 1$
- Learn scale and shift parameters
- Apply during training and testing

BatchNorm Algorithm:

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i$$

$$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2$$

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

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

In plain words: 1) Find average, 2) Find spread, 3) Normalize to standard range, 4) Scale and shift as needed

Benefits:

- 10x faster training
- Higher learning rates OK
- Less sensitive to initialization

2019: The Lottery Ticket Hypothesis

Most Network Weights Don't Matter!

The Discovery:

- Networks contain "winning tickets"
- Subnetworks that train well alone
- 90-95% of weights can be removed
- Performance stays the same!

The Hypothesis: "Dense networks succeed because they contain sparse subnetworks that are capable of training effectively"

Implications:

- We massively overparameterize
- Training finds the needle in haystack
- Future: Train small from start?
- Mobile deployment possible

Why It Matters:

- Explains why big networks train better
- Pruning after training works
- Efficiency revolution starting
- Changes how we think about learning

A 1 billion parameter model might only need 50 million

Inductive Biases: Building in Assumptions

The Right Architecture for the Right Problem

What Are Inductive Biases?

- Assumptions built into architecture
- Guide learning toward solutions
- Trade flexibility for efficiency
- "Priors" about the problem

Examples:

- **CNN:** Spatial locality matters
- **RNN:** Order/time matters
- **GNN:** Graph structure matters
- **Transformer:** All positions can interact

Why They Matter:

- Reduce search space
- Faster convergence
- Better generalization
- Less data needed

The Tradeoff:

- Right bias = 10x better
- Wrong bias = 10x worse
- General architectures = safe but slow
- Specialized = fast but limited

Choosing the right inductive bias is still an art

Capabilities That Appear Suddenly with Scale

The Phenomenon:

- Small models: Can't do task at all
- Medium models: Still can't
- Large models: Suddenly can!
- No gradual improvement

Examples:

- 3-digit arithmetic (¿10B params)
- Chain-of-thought reasoning (¿50B)
- Code generation (¿20B)
- Multilingual translation (¿100B)

Why It Happens:

- Complex patterns need capacity
- Phase transitions in learning
- Composition of simpler abilities
- "Grokking" - sudden understanding

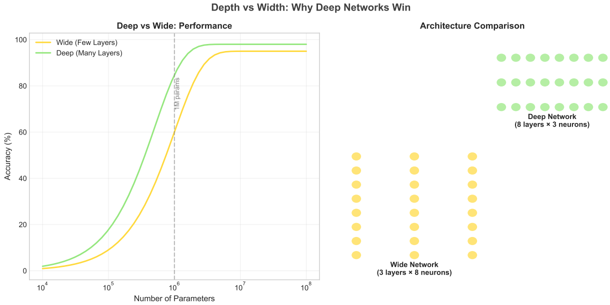
Implications:

- We can't predict what's next
- Scaling might unlock AGI
- Or hit fundamental limits
- Active area of research

GPT-3 showed abilities nobody expected or programmed

Architecture Choices: Deep vs Wide Networks

The Fundamental Tradeoff in Neural Architecture



Deep Networks (Many Layers):

- Complex hierarchical features
- Exponential expressiveness growth
- Harder to train (vanishing gradients)
- Better for vision, NLP

Wide Networks (Many Neurons):

The Sweet Spot:

- Vision: Deep (100+ layers)
- Language: Very deep (24-96 layers)
- Tabular: Wide and shallow (2-4 layers)
- Time series: Moderate (5-10 layers)

Modern Insights:

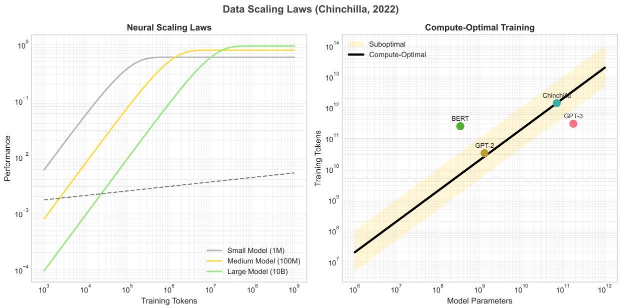
- Depth beats width for same parameters
- Skip connections enable extreme depth
- Width helps with memorization
- Depth helps with generalization

Scaling Laws:

- Performance $\propto$ depth$^{0.8}$
- Performance $\propto$ width$^{0.5}$

Scaling Laws: How Performance Grows with Data

The Predictable Relationship Between Data, Model Size, and Performance



The Chinchilla Law (2022):

- Optimal ratio: 20 tokens per parameter
- 10B model needs 200B tokens
- Most models are undertrained
- Data quality matters more than quantity

Power Law Scaling:

Practical Implications:

- 10x data → 2x performance
- 10x parameters → 1.7x performance
- 10x compute → 3x performance
- Diminishing returns always

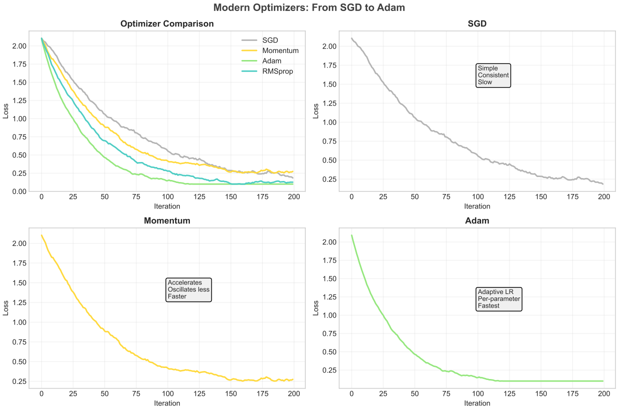
Data Efficiency Tricks:

- Data augmentation
- Synthetic data generation
- Active learning
- Curriculum learning
- Multi-task training

Why it matters: These laws predict costs before training

Current Limits:

The Evolution of Gradient Descent



SGD (1951):

- Basic gradient descent
- Learning rate: Fixed

Adam (2014):

- Adaptive learning rates per parameter
- Combines momentum + RMSprop
- De facto standard
- Works out-of-the-box

Modern Variants:

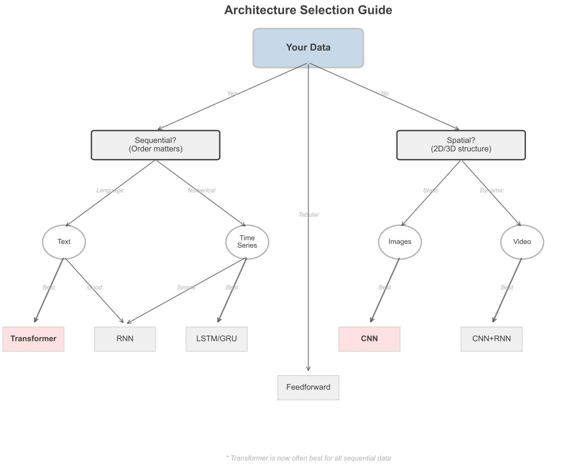
- **AdamW:** Decoupled weight decay
- **RAdam:** Rectified Adam
- **LAMB:** Large batch training
- **Sophia:** 2nd-order approximation

Choosing an Optimizer:

- Start with Adam (lr=3e-4)

Quick Guide: Choosing Your Architecture

Which Network Should You Use?



Decision Questions:

1. Is your data sequential?
2. Does position matter?
3. Is it images/spatial?
4. Fixed or variable size?

Quick Rules:

- Images $\rightarrow$ CNN
- Text $\rightarrow$ Transformer/RNN
- Tabular $\rightarrow$ Feedforward
- Audio $\rightarrow$ CNN or RNN
- Video $\rightarrow$ CNN + RNN

Common Confusion: Transformers now dominate most tasks, but specialized architectures still win for specific problems!

Neural Network Architectures: Right Tool for Right Job

Feedforward Networks:

- Information flows forward only
- Fixed-size input and output
- Good for: Classification, regression

Convolutional (CNN):

- Spatial feature detection
- Translation invariance
- Good for: Images, video

Recurrent (RNN):

- Process sequences
- Maintain memory/state
- Good for: Text, time-series

Transformer:

- Attention mechanism
- Parallel processing
- Good for: Language, everything else

Each architecture encodes different assumptions about the data

Modern Training: Standing on Shoulders of Giants

Transfer Learning:

- Start with pre-trained network
- Fine-tune on your task
- 100x less data needed
- Days $\rightarrow$ Hours training

Data Augmentation:

- Create variations of training data
- Rotations, crops, color shifts
- Prevents overfitting
- Free performance boost

Advanced Optimizers:

- **SGD:** Basic gradient descent
- **Momentum:** Remember past gradients
- **Adam:** Adaptive learning rates
- **AdamW:** With weight decay

Mixed Precision:

- Use 16-bit floats where possible
- Keep 32-bit for critical ops
- 2-3x speedup
- Same accuracy

These techniques make deep learning practical for everyone

Common Mental Models That Are WRONG

Misconceptions That Will Confuse You

WRONG: "Neurons are like brain neurons"

- Brain neurons: Complex, chemical, adaptive
- Artificial neurons: Simple math functions
- Just multiply and add!
- No biology involved

WRONG: "Networks understand concepts"

- What you think: "It knows what a cat is"
- Reality: It found statistical patterns
- No understanding, just correlation
- Can be fooled by tiny changes

WRONG: "More layers = always better"

- Too deep: Vanishing gradients
- Too deep: Overfitting
- Right depth: Depends on problem complexity
- Simple problems need shallow networks

WRONG: "It learns like humans"

- Humans: Learn from few examples
- Humans: Transfer knowledge easily
- Networks: Need thousands of examples
- Networks: Struggle with new situations

Remember: Neural networks are just fancy pattern matchers.
They don't think, understand, or reason - they find correlations in data.

Understanding these limits helps you use neural networks effectively

Why Deep Learning Exploded Now: The Perfect Storm

1. **Data Explosion:**
   - Internet = infinite training data
   - ImageNet: 14M labeled images
   - Common Crawl: 300TB of text
   - YouTube: 500 hours/minute
2. **Hardware Revolution:**
   - GPUs: 100x faster than CPUs
   - TPUs: Built for neural nets
   - Cloud computing: Rent supercomputers
   - Mobile chips with NPUs
3. **Algorithm Breakthroughs:**
   - ReLU activation (2011)
   - Batch normalization (2015)
   - Skip connections (2015)
   - Attention mechanism (2017)
4. **Open Source Culture:**
   - TensorFlow, PyTorch free
   - Pre-trained models shared
   - Papers with code
   - Collaborative research

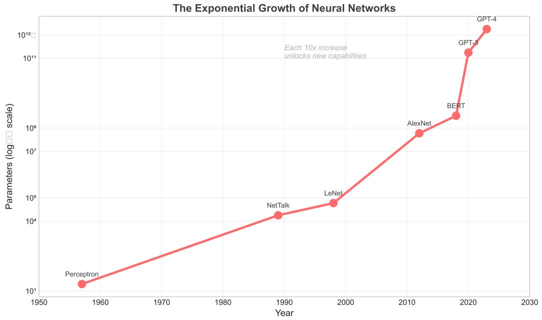
The same ideas from 1980s finally had the resources to work

Understanding Scale: From Perceptron to GPT-4

The Exponential Growth of Neural Networks

Parameter Growth:

- 1957 Perceptron: 20 weights
- 1987 NetTalk: 18,000
- 1998 LeNet: 60,000
- 2012 AlexNet: 60 million
- 2018 BERT: 340 million
- 2020 GPT-3: 175 billion
- 2023 GPT-4: 1.8 trillion



What Scale Brings:

- Emergent abilities
- Zero-shot learning
- Multi-task capability
- Common sense reasoning

Building a Digit Classifier in 10 Lines

PyTorch Implementation:

```python
import torch
import torch.nn as nn

class SimpleNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(784, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        return self.fc2(x)

# Train
model = SimpleNet()
optimizer = torch.optim.Adam(model.parameters())
criterion = nn.CrossEntropyLoss()
```

What This Does:

- Input: 28×28 pixel image
- Hidden: 128 neurons
- Output: 10 digit classes
- Activation: ReLU
- Training: Adam optimizer

Training Loop:

- Forward pass
- Calculate loss
- Backward pass
- Update weights
- Repeat

This simple network achieves 97% accuracy on MNIST

When Things Go Wrong (They Always Do)

Gradient Issues:

- **Exploding:** Gradients → infinity
  - Solution: Gradient clipping
- **Vanishing:** Gradients → 0
  - Solution: Better initialization, ReLU
- **Dead ReLU:** Neurons never activate
  - Solution: LeakyReLU, smaller learning rate

Debugging Tools:

- TensorBoard: Visualize training
- Gradient histograms
- Activation distributions
- Weight evolution plots

Common Failure Modes:

- Loss not decreasing: Learning rate
- Loss NaN: Numerical instability
- Oscillating loss: LR too high
- Plateau: Local minimum or LR too small

Sanity Checks:

1. Overfit single batch first
2. Check gradient flow
3. Visualize first layer filters
4. Plot loss curves
5. Test on toy problem

"If it's not working, it's always the learning rate" - Andrej Karpathy

Your Debugging Checklist: When Things Go Wrong

Systematic Debugging Saves Hours

Try It Yourself: Save this checklist - you'll need it for every project!

Step 1: Sanity Checks

- ☐ Can you overfit a single batch?
- ☐ Are inputs normalized?
- ☐ Is output layer correct?
- ☐ Loss function matches task?

Step 2: Data Checks

- ☐ Plot sample inputs
- ☐ Check label distribution
- ☐ Verify train/val split
- ☐ Look for data leakage

Step 3: Training Checks

- ☐ Plot loss curves
- ☐ Check gradient norms
- ☐ Monitor weight updates
- ☐ Try different learning rates

Step 4: Architecture

- ☐ Start with known working model
- ☐ Add complexity gradually
- ☐ Check activation distributions
- ☐ Verify dimensions match

Common Confusion: 90% of bugs are in data preprocessing, not the model!

Print this slide and keep it handy

Common Pitfalls: Learn from Others' Mistakes

Data Problems:

- Not enough data
- Unbalanced classes
- Data leakage
- No validation set

Architecture Issues:

- Too deep without skip connections
- Wrong activation functions
- Incorrect output layer
- Bad initialization

Training Mistakes:

- Learning rate too high/low
- No normalization
- Overfitting ignored
- Wrong loss function

Debugging Tips:

- Start simple, add complexity
- Overfit single batch first
- Monitor gradients
- Visualize predictions

"It's not working" usually means one of these issues

The Future: What's Next?

Current Frontiers:

- Multimodal models (text+image+audio)
- Efficient models for phones
- Neuromorphic hardware
- Quantum neural networks

Unsolved Problems:

- True reasoning ability
- Learning from few examples
- Explaining decisions
- Energy efficiency

Next Breakthroughs?

- Models that update continuously
- Networks that program themselves
- Biological-digital hybrids
- AGI (Artificial General Intelligence)?

Your Role:

- This field is 70 years young
- Major breakthroughs every 2-3 years
- Anyone can contribute
- The best is yet to come

"We're still in the steam engine era of AI" - Geoffrey Hinton

Final Check: Can You Explain These to a Friend?

Test Your Understanding

Core Concepts:

1. Why do we need activation functions?
2. What's backpropagation in one sentence?
3. Why did deep learning explode after 2012?
4. What's the vanishing gradient problem?
5. Why do CNNs work for images?

Try It Yourself: Write one-sentence answers for each. Compare with a classmate!

Key Answers:

- Without them, stacked layers = still linear
- Distributing error backwards through network
- GPUs + Big Data + ReLU converged
- Gradients shrink through many layers
- They detect features regardless of position

If You're Stuck:

- Review activation functions slide
- Re-read backprop section
- Check AlexNet breakthrough
- Look at gradient flow diagram
- Study convolution hierarchy

Understanding these concepts prepares you for everything that follows

The Journey So Far

Core Concepts:

1. **Neurons:** $y = f(\sum w_i x_i + b)$
2. **Learning:** Adjust weights to minimize error
3. **Depth:** Each layer adds abstraction
4. **Backpropagation:** Distribute error backwards
5. **Non-linearity:** Enables complex functions

Historical Lessons:

1. Every limitation spawned innovation
2. Simple ideas + scale = revolution
3. Biology inspires but doesn't limit
4. Persistence through AI winters
5. Theory + engineering = breakthroughs

You now understand the fundamentals that power all modern AI

Epilogue: Your First Neural Network in 5 Minutes

Let's Build Something Real!

Complete MNIST Classifier:

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

# 1. Define Network
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(784, 128)
        self.fc2 = nn.Linear(128, 64)
        self.fc3 = nn.Linear(64, 10)

    def forward(self, x):
        x = x.view(-1, 784)  # Flatten
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        return self.fc3(x)

# 2. Load Data
transform = transforms.ToTensor()
train_data = datasets.MNIST('.', train=True,
                            download=True,
                            transform=transform)
train_loader = DataLoader(train_data,
                          batch_size=64,
                          shuffle=True)

# 3. Setup Training
```

```
# 4. Training Loop
for epoch in range(3):
    for batch_idx, (data, target) in
            enumerate(train_loader):
        # Forward pass
        output = model(data)
        loss = criterion(output, target)

        # Backward pass
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        # Print progress
        if batch_idx % 100 == 0:
            print(f'Epoch {epoch}: {loss:.4f}')

# 5. Test One Example
model.eval()
test_image = train_data[0][0]
prediction = model(test_image.unsqueeze(0))
print(f"Predicted: {prediction.argmax()}")
```

What You Just Built:

- 3-layer neural network
- 60K training images
- 97% accuracy in 3 epochs

Where to Go Next: Your Learning Path

Continue Your Neural Network Journey

Next Topics to Learn:

1. **CNNs:** Computer vision
2. **RNNs/LSTMs:** Sequences
3. **Transformers:** Modern NLP
4. **GANs:** Generation
5. **RL:** Decision making

Practical Projects:

- Image classifier for your photos
- Sentiment analysis of tweets
- Chatbot for customer service
- Style transfer for art
- Game-playing agent

Resources:

- **Fast.ai:** Practical deep learning
- **PyTorch Tutorials:** Official guides
- **Papers with Code:** Latest research
- **Kaggle:** Competitions and datasets
- **3Blue1Brown:** Visual explanations

Remember:

- Start simple, build up
- Theory + practice together
- Join communities
- Build projects you care about
- Share what you learn

You've learned how humanity taught machines to think.
Now it's your turn to push the boundaries!

The future of AI is being written now - be part of it!