# 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 - Optimized Flow**

**Phase 1: The Motivation**

- The mail sorting crisis
- Why rules don't work
- First mathematical neurons
- Rosenblatt's learning breakthrough

**Phase 2: Understanding Basics**

- How neurons calculate
- Activation functions (NEW placement!)
- Hand calculations and examples

**Phase 3: Crisis & Solution**

- The XOR problem
- Geometric intuition (NEW!)
- Hidden layers solution
- Backpropagation breakthrough

**Phase 4: Theory to Practice**

- Gradient landscape visualization
- LeNet (1998): First success
- AlexNet (2012): Deep learning explosion
- Modern architectures
- Real-world applications

**Phase 5: Your Turn**

- Building your first network
- Debugging tips
- Next steps & resources

**Key Changes from Original:**

- Activation functions BEFORE XOR
- Geometric intuition bridge added
- Function approx AFTER theory

## 1950s: The Mail Sorting Crisis

**The Challenge:**

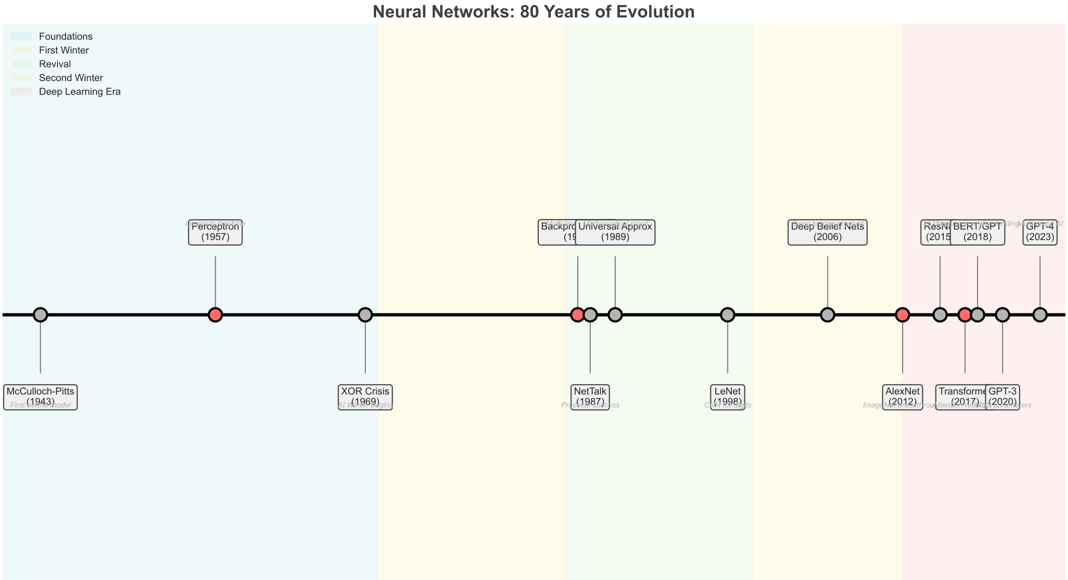
- 150 million letters per day
- Hand-written addresses
- Human sorters: slow, expensive, error-prone
- Traditional programming: useless

**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?

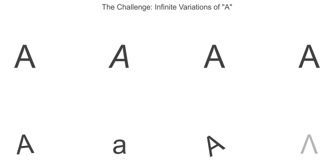
**Problem: Recognize the Letter "A"**

## 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?




## The Insight:

- We need pattern recognition, not rules
- System must learn from examples
- Similar to how humans learn

---

**This realization launched the field of machine learning**

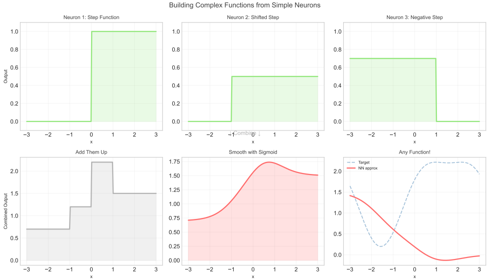
# 1943: The First Mathematical Neuron

**McCulloch & Pitts:**

- Neurophysiologists studying brain
- Asked: Can neurons be modeled mathematically?
- Created first artificial neuron

**The Model:**

- Multiple inputs (dendrites)
- Weighted sum (cell body)
- Threshold activation (axon)
- Binary output (fire or not)




**The Limitation:**

- Weights were fixed
- No learning mechanism
- Programmer had to set weights manually

**Revolutionary idea, but missing the key ingredient: learning**

**Frank Rosenblatt's Insight:**

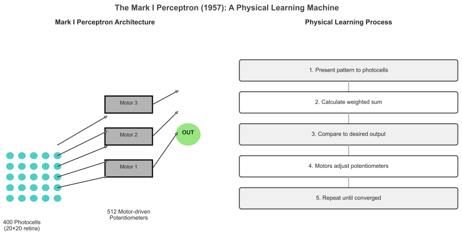
> *"What if the machine could adjust its own weights based on mistakes?"*

**The Perceptron Learning Rule:**

1. Make a prediction
2. Check if wrong
3. If wrong: adjust weights
4. Repeat until correct

**Historic Demo (1958):**

- Mark I Perceptron machine
- Learned to recognize simple shapes
- Press coverage: "Thinking machine!"




**Why Revolutionary:**

- First machine that could learn
- Weights adjusted automatically
- Learned from examples, not rules
- Mathematically proven to converge

**This was the birth of machine learning**

## 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**

# Making It Concrete: Teaching OR Logic

**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**

## 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**

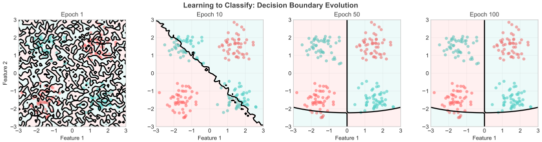
**Teaching a Network to Separate Red from Blue Points**

**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**

# 1969: The Problem That Killed AI

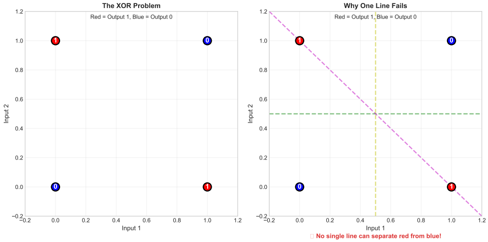
# The XOR Problem: Why Single Neurons Fail

**XOR (Exclusive OR):**

- Output 1 if inputs are different
- Output 0 if inputs are same

**Truth Table:**

| $x_1$ | $x_2$ | Output |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |




**The Challenge:**

- Try drawing ONE straight line
- That separates 1's from 0's
- Impossible!

**Why It Matters:**

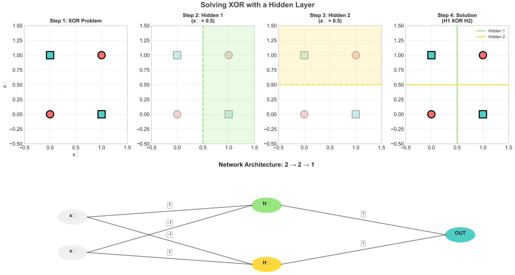
- Perceptrons can only draw straight lines
- XOR requires curved boundary
- This is the simplest non-linear problem
- Minsky & Papert proved this mathematically

**This proof triggered the first AI Winter (1970-1980)**

# NEW: Geometric Intuition - Why We Need Two Boundaries

**The Key Insight: XOR Needs TWO Lines, Not One**

**Visualization:**




**What We See:**

- Red line: Separates (0,0) from others
- Blue line: Separates (1,1) from others
- Green region: Intersection of both
- Points (0,1) and (1,0) in green = Output 1!

**This geometric intuition explains WHY hidden layers work**

**The Breakthrough Idea:**

1. Use TWO neurons (not one)
2. Each neuron creates one boundary
3. Combine their outputs
4. Intersection solves XOR!

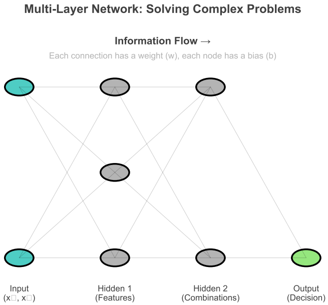
**This Requires:**

- Hidden layer with 2 neurons
- Output layer combines results
- This is a 2-layer network
- First layer: create boundaries
- Second layer: find intersection

**Key Insight:** Hidden layers let us combine simple boundaries into complex decision regions!

# The Solution: Hidden Layers

**Architecture with Hidden Layer:**




**How It Works:**

- Input layer: 2 neurons ($x_1$, $x_2$)
- Hidden layer: 2 neurons (two boundaries)
- Output layer: 1 neuron (combines)

**Forward Pass for XOR:**

Given weights:

- Hidden 1: $w = [1, 1], b = -0.5$
- Hidden 2: $w = [1, 1], b = -1.5$
- Output: $w = [1, -1], b = 0$

For input $(1, 0)$:

$$\begin{aligned} h_1 &= \sigma(1 \cdot 1 + 1 \cdot 0 - 0.5) \\ &= \sigma(0.5) \approx 0.62 \\ h_2 &= \sigma(1 \cdot 1 + 1 \cdot 0 - 1.5) \\ &= \sigma(-0.5) \approx 0.38 \\ y &= \sigma(1 \cdot 0.62 - 1 \cdot 0.38) \\ &= \sigma(0.24) \approx 0.56 \text{ (close to 1!)} \end{aligned}$$

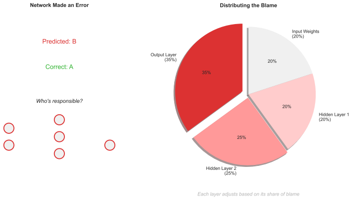
**Hidden layers unlock non-linear patterns**

# The New Challenge: How to Train Hidden Layers?

**The Credit Assignment Problem:**

- Output is wrong - we know the error
- But which hidden neuron caused it?
- How much should each weight change?
- Perceptron rule only works for output layer

**Why It's Hard:**




**Early Failed Attempts (1970s):**

- Random weight adjustment
- Genetic algorithms
- Simulated annealing
- All too slow or unreliable

**What We Needed:**

- Systematic way to assign blame
- Efficient computation
- Guaranteed to improve
- Works for many layers

**The solution would come from calculus...**

**This problem was solved in 1986 with backpropagation**

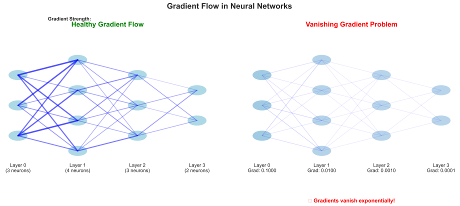
# 1986: Backpropagation - The Breakthrough Algorithm

**The Algorithm (Rumelhart et al.):**

1. **Forward pass**: Compute output
2. **Compute error**: Compare to target
3. **Backward pass**: Use chain rule to compute gradients
4. **Update weights**: Gradient descent

**The Key Insight:**

- Use calculus (chain rule)
- Error flows backward through network
- Each layer gets its share of blame
- Weights adjusted proportionally




**Mathematical Foundation:**

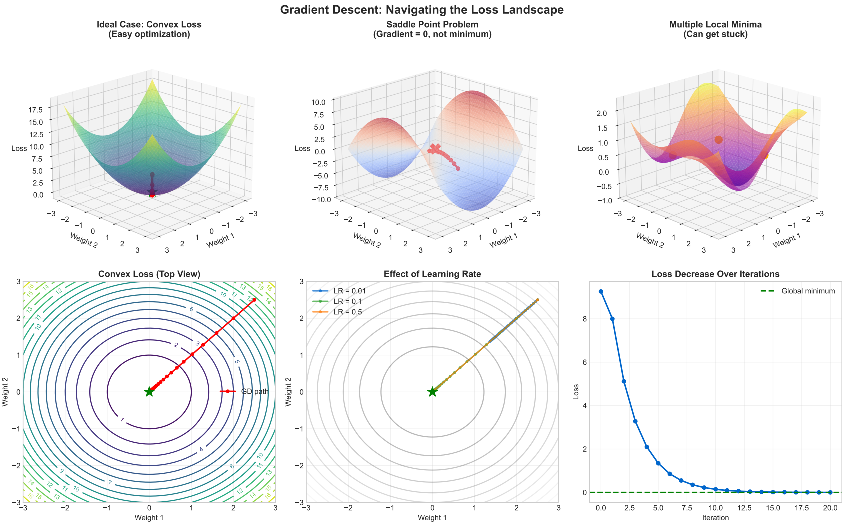
$$\frac{\partial L}{\partial w_{ij}} = \frac{\partial L}{\partial a_j} \cdot \frac{\partial a_j}{\partial z_j} \cdot \frac{\partial z_j}{\partial w_{ij}} = \delta_j \cdot a_i$$

**Why Revolutionary:**

- Efficient: One backward pass
- General: Works for any architecture
- Automatic: No manual tuning

**This paper revived neural networks and enabled modern deep learning**

# Visualizing Learning: The Gradient Landscape



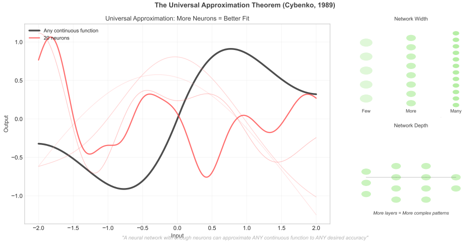

**Gradient descent finds the valley where error is minimized**

**Cybenko's Universal Approximation Theorem:**

> *A neural network with one hidden layer and finite neurons can approximate ANY continuous function to ANY desired accuracy*

**What This Means:**

- Mathematical proof
- Not just XOR - ANY pattern!
- Theoretical justification
- Explains why NNs are so powerful




**Caveats:**

- Guarantees existence, not learning
- May need exponential neurons
- Deep networks often more efficient
- Still need good training algorithm

**Theory meets practice: NNs CAN learn any pattern, backprop shows us HOW**

# 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:**

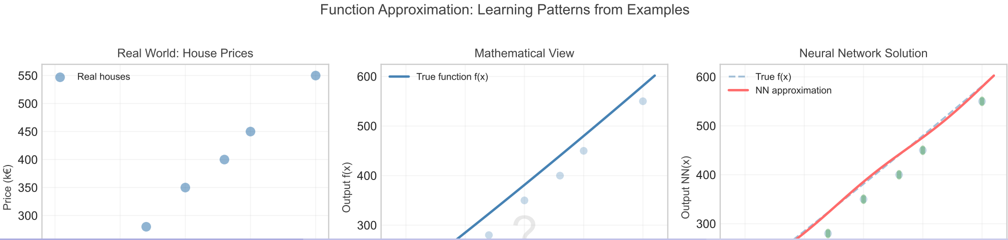
- Guess the formula
- Write explicit rules
- Hope it works
- Problem: Real world is too complex!

*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!



**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

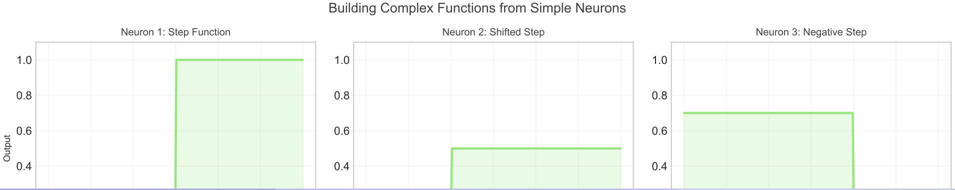
**Real-World Analogy:**

*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:*

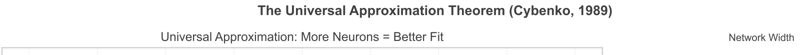
1. With 4 pixels: Very blocky image
2. With 100 pixels: Recognizable
3. With 10,000 pixels: Photo-realistic
4. With infinite pixels: Perfect!

*Same with neurons:*

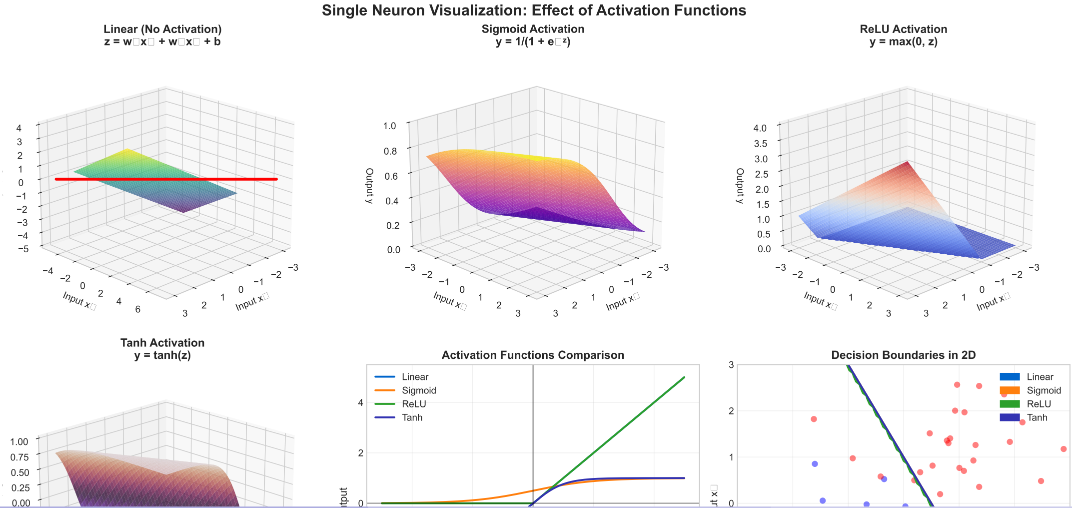
1. Few neurons: Rough approximation
2. More neurons: Better fit
3. Many neurons: Nearly perfect
4. Infinite neurons: Exact function!

**Why This Matters:**

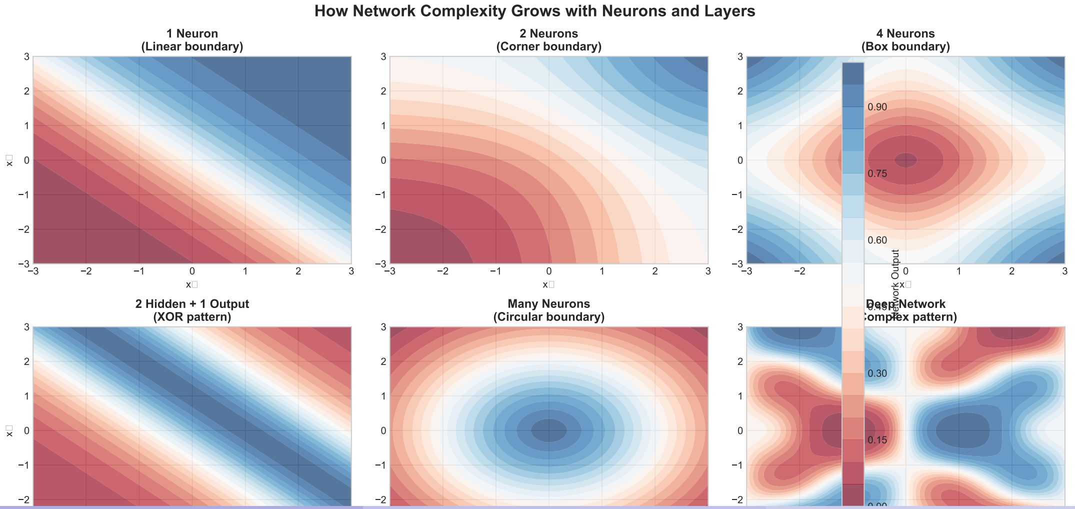
We don't need different architectures for different problems - just one universal tool that adapts!



## Visualizing How Activation Functions Transform the Output Space

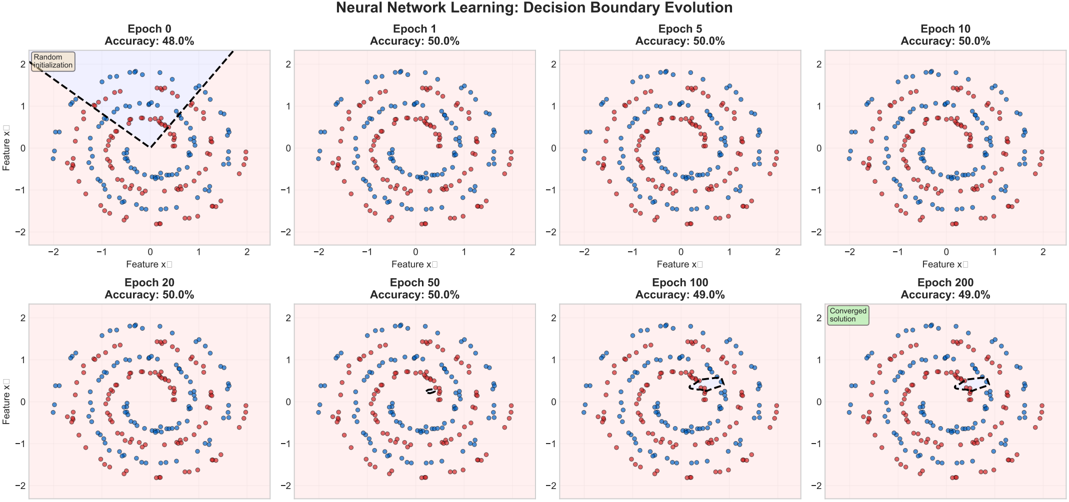


## How More Neurons Enable More Complex Decision Boundaries

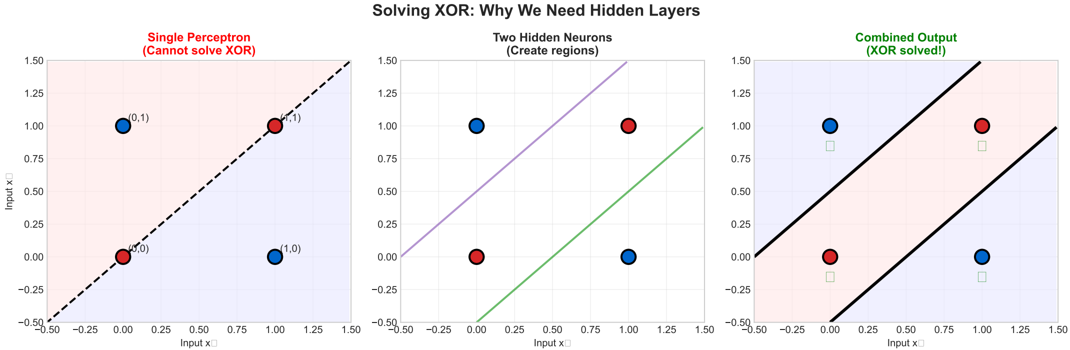


# The Learning Process: Frame by Frame

## Watching Decision Boundaries Evolve During Training

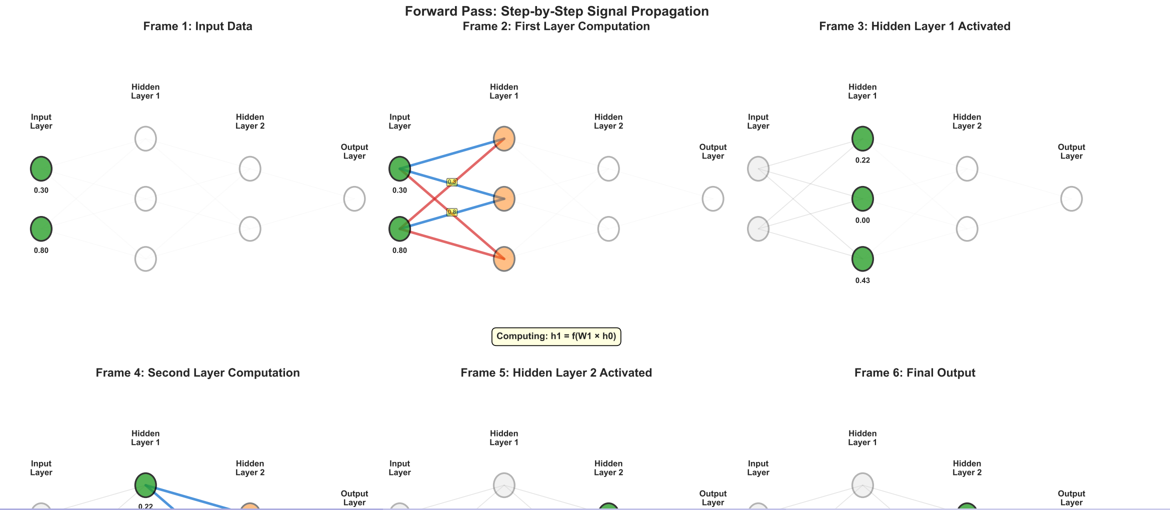


## Why We Need Hidden Layers: The XOR Solution

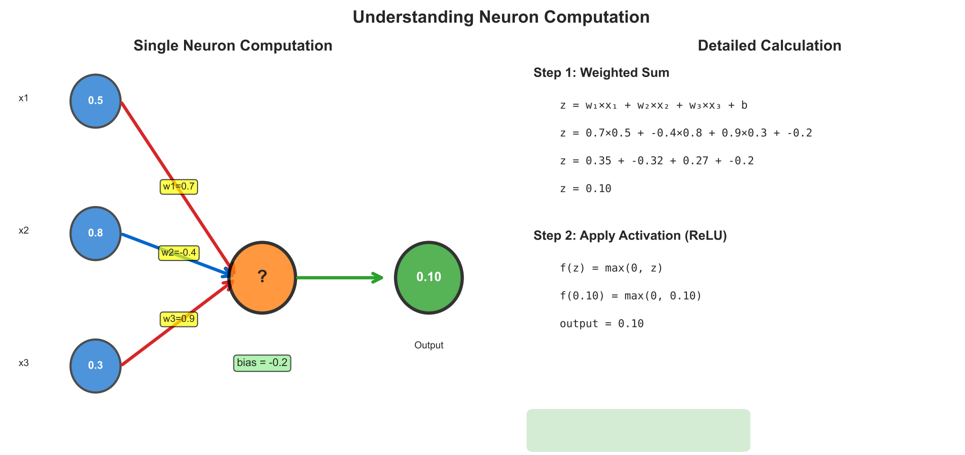



**Two hidden neurons working together can solve what one neuron cannot**

## Following Data as it Flows Through the Network



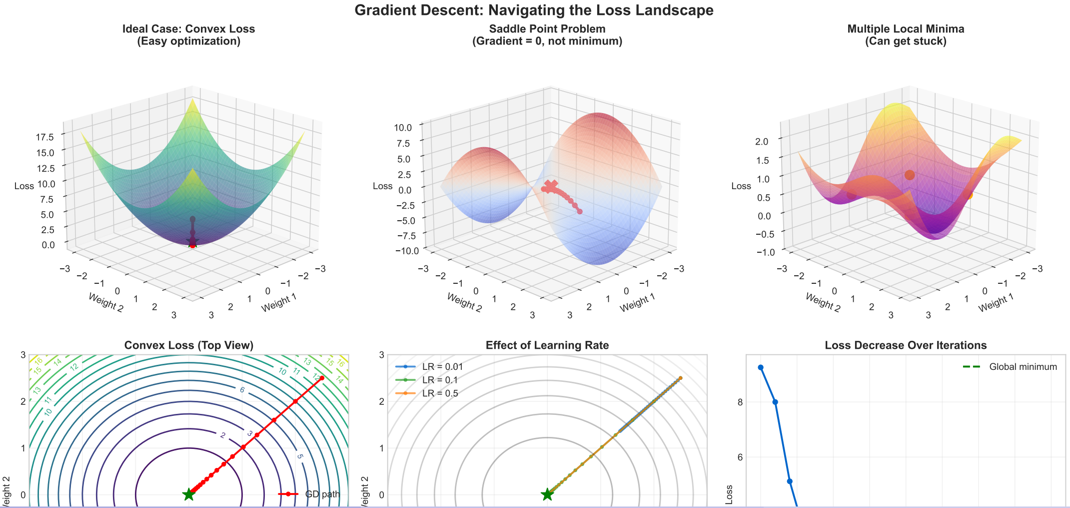
## The Math Behind a Single Neuron's Calculation



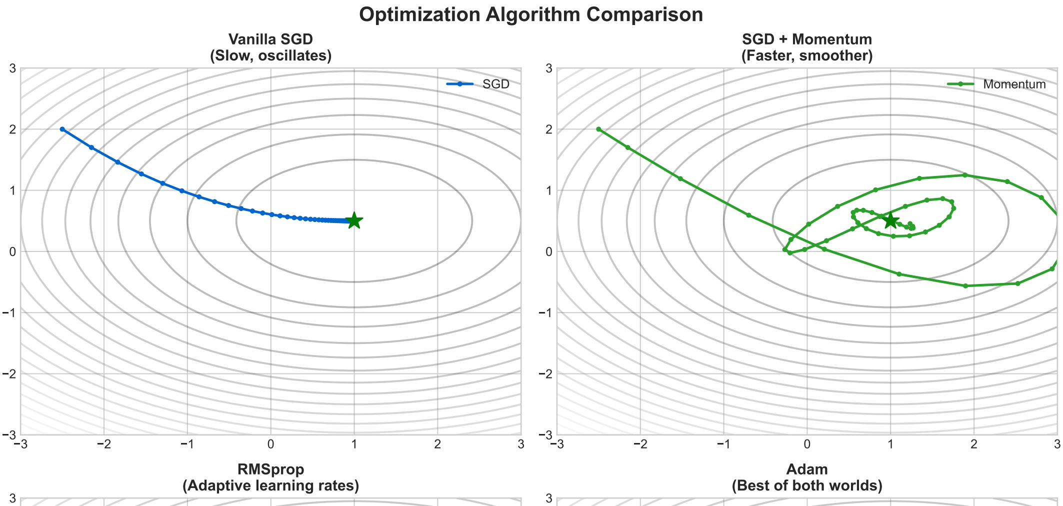

**Try It Yourself:** Follow along: multiply each input by its weight, add them up, add bias, apply activation!

**This calculation happens millions of times per second in modern networks**

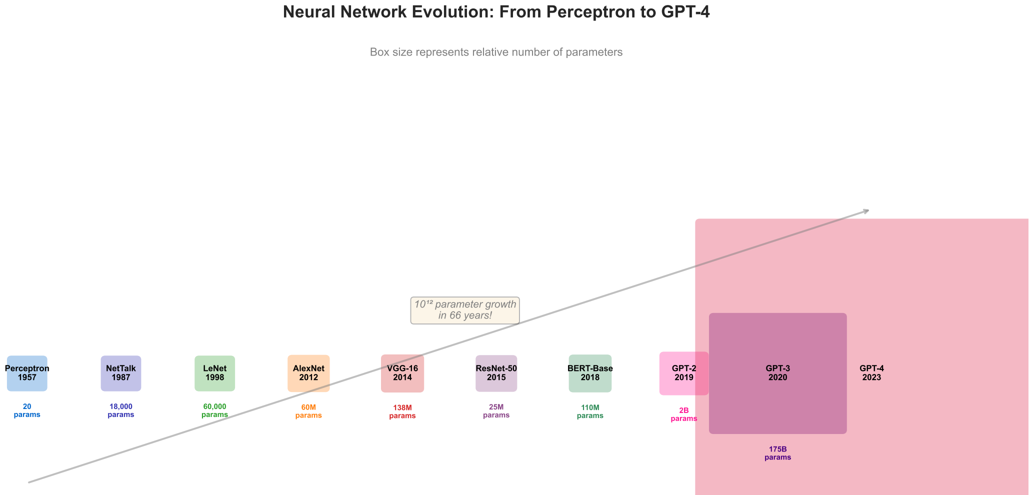
## Gradient Descent: Finding the Valley in 3D Space



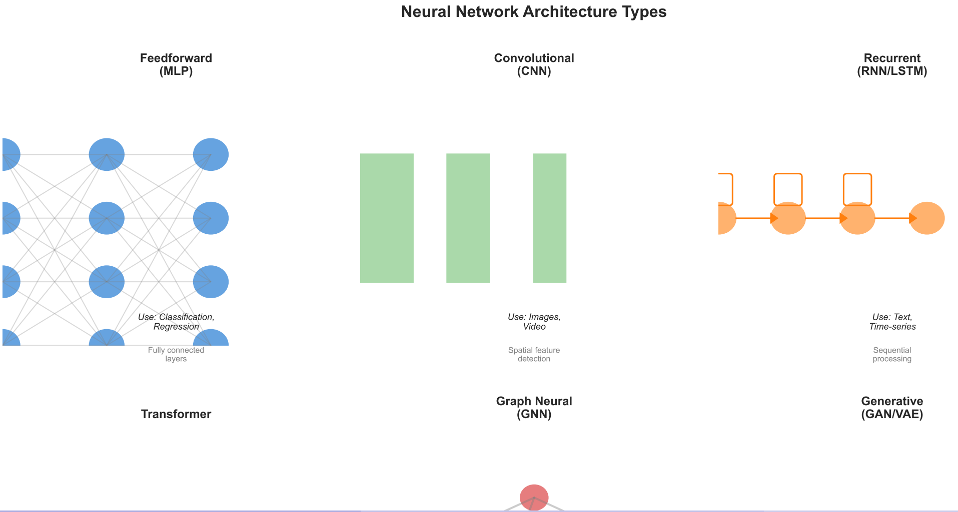
## Why Adam Outperforms Simple Gradient Descent



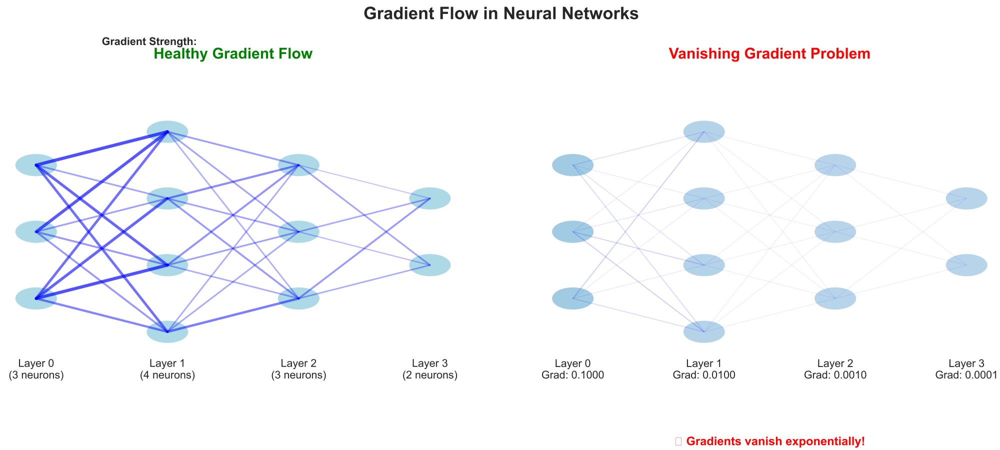
## From 20 Parameters to 1.8 Trillion: The Growth of Neural Networks



## Different Architectures for Different Problems



## Why Deep Networks Were Hard Before ReLU




**Left: Healthy gradient flow, Right: Vanishing gradients - this problem limited networks to 2-3 layers for decades**

**From Theory to Working Systems**

## 1998-2012: From Digits to ImageNet

**1998 - LeNet: First Success**

- Yann LeCun's CNN for digits
- $32 \times 32$ 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**

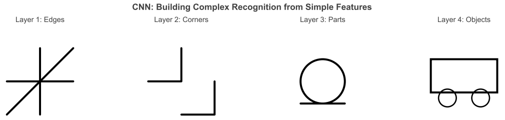
**How We Actually Recognize Objects**

**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**

**Finding the Best Weights: Like Hiking Down a Mountain**

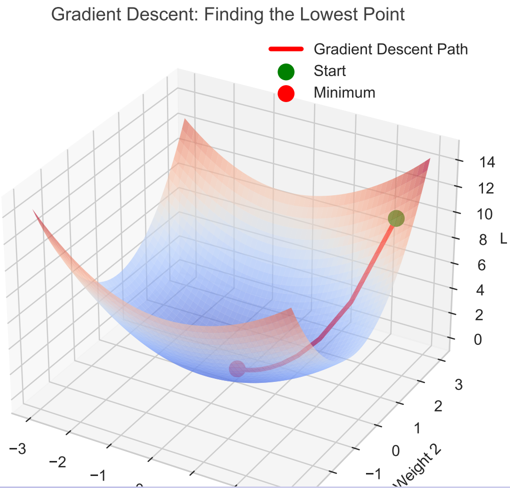
**The Optimization Problem:**

- 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**

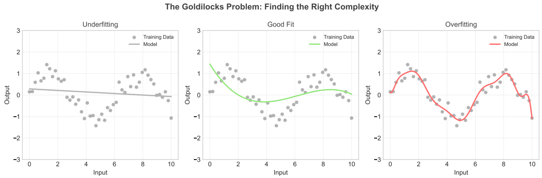
## Memorization vs. Understanding

**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



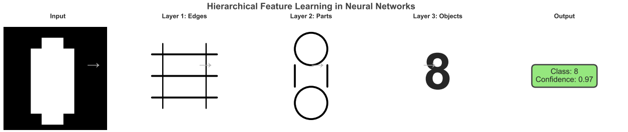

**Solutions:**

- **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

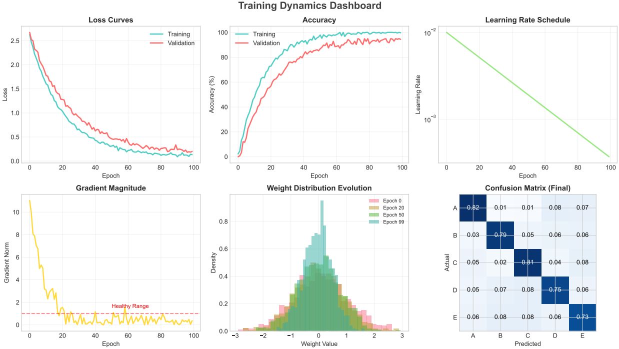
**Discovered Automatically:**

- 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**

# Training Dynamics: Watching Networks Learn

## 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**

**The Explosion of Modern AI**

## 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:**

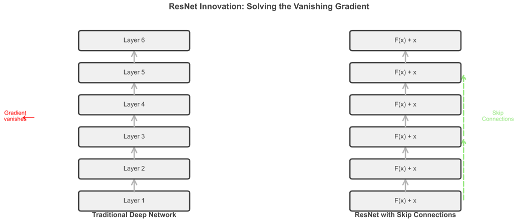
- **Vision:** Self-driving cars
- **Language:** Google Translate
- **Speech:** Siri, Alexa
- **Medicine:** Disease diagnosis
- **Science:** Protein folding

**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**

# Batch Normalization: Keeping Networks Stable

**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**

## 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

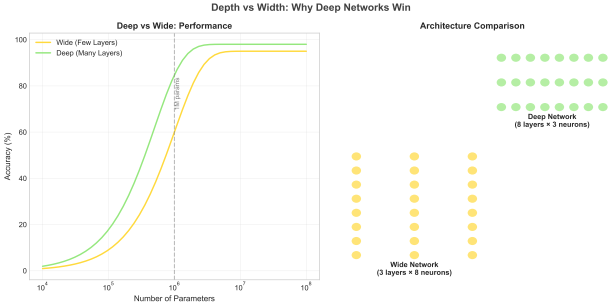
**Implications:**

- 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)

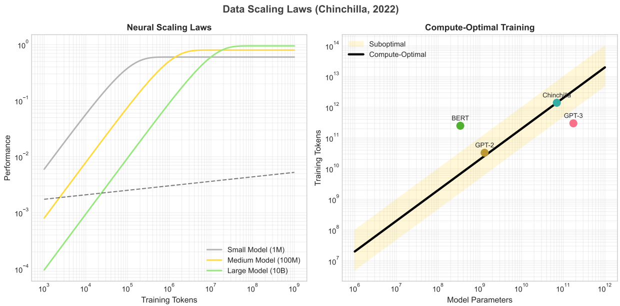
**Modern Insights:**

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

**Scaling Laws:**

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

## 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

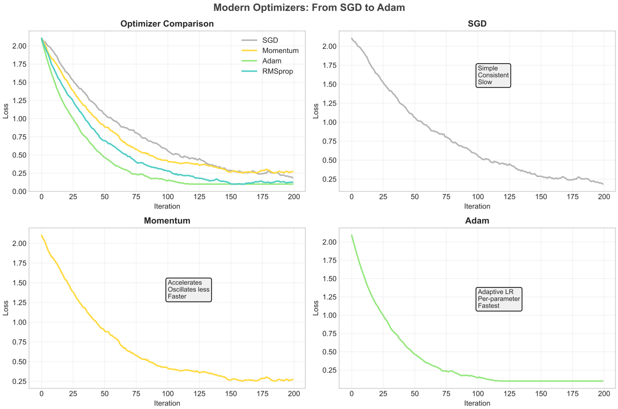
**Data Efficiency Tricks:**

- 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

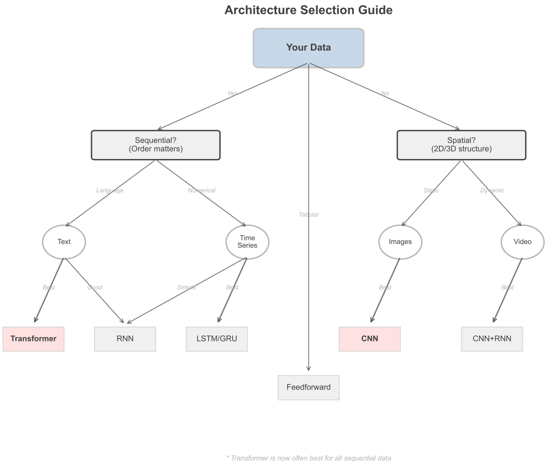
**Modern Variants:**

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

**Choosing an Optimizer:**

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

## 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 → CNN
- Text → Transformer/RNN
- Tabular → Feedforward
- Audio → CNN or RNN
- Video → CNN + RNN

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

## 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**

## Let's Build Something Real!

### Complete MNIST Classifier:

```python
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
```

```python
# 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!**

**Additional Material for Deep Dive**

- Appendix A: Advanced Topics
- Appendix B: Extended History

# Deep Dives for the Curious

**This section contains advanced material that goes beyond the core BSc curriculum.**

Topics covered:

- The Lottery Ticket Hypothesis
- Inductive Biases in Neural Architectures
- Scaling Laws and Performance Prediction
- Deep vs Wide Network Architectures
- Emergent Abilities at Scale
- Advanced Optimization Algorithms

*These topics are valuable for understanding state-of-the-art research but not essential for getting started with neural networks.*

**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**

**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

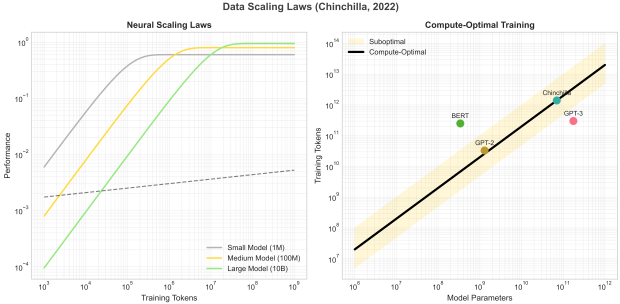
**The Tradeoff:**

- 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**

# 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:

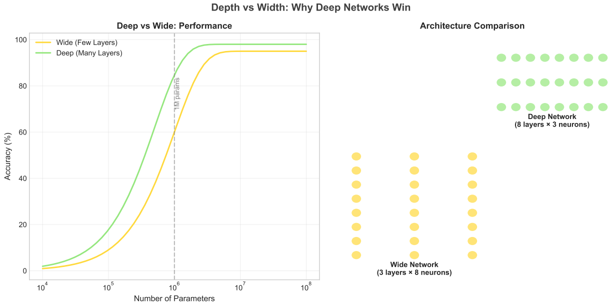
- Data augmentation
- Synthetic data generation
- Active learning
- Curriculum learning
- Multi-task training

Why it matters: These laws predict costs before training

## Current Limits:

# 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$^{0.5}$

## 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

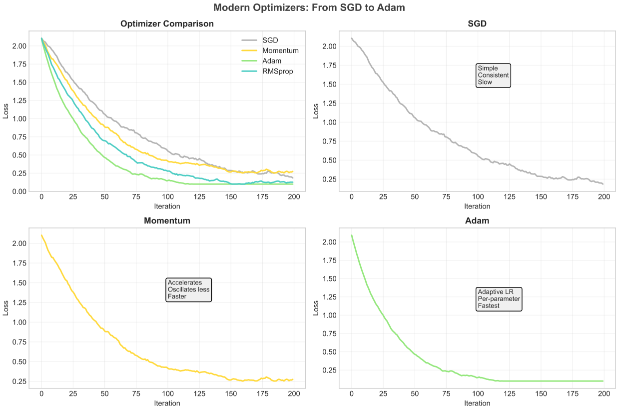
**Implications:**

- 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**

# Optimization Algorithms: How Networks Learn

## 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)

## The Full Story: Historical Deep Dives

**This section contains fascinating historical details that enrich the narrative but aren't essential for understanding the technical concepts.**

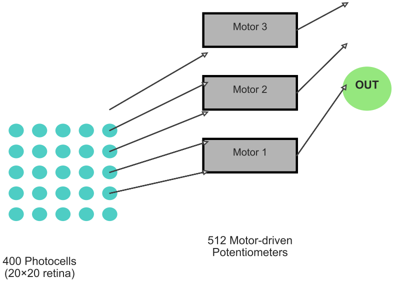
Topics covered:

- The Mark I Perceptron: Physical Hardware
- NetTalk: Networks Learn to Speak (1987)
- Batch Normalization: Keeping Networks Stable

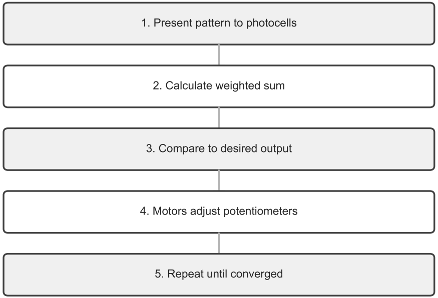
*These stories show how each breakthrough built on previous work and overcame specific limitations.*

## The Mark I Perceptron (1957): A Physical Learning Machine

**Mark I Perceptron Architecture**




**Physical Learning Process**




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

# 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:**

- $7 \times 29$ 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"**

## 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