Module 1: The Birth of Neural Computing

From Biological Inspiration to the Perceptron (1943-1969)

Neural Networks for Finance

BSc Lecture Series

November 28, 2025

The Investment Committee

How Does a Committee Make Decisions?
Imagine an investment committee evaluating a stock:

- **Analyst A**: "Strong earnings growth" (+1 vote)
- **Analyst B**: "High debt levels" (-1 vote)
- **Analyst C**: "Good momentum" (+1 vote)
- **Senior Partner**: "Market risk is elevated" (-2 votes)

The Decision Process:

1. Gather evidence from each analyst
2. Weight opinions by seniority/expertise
3. Sum the weighted votes
4. If total $>$ threshold: **Buy**

Weighted Voting

Analyst	Vote	Weight
Analyst A	+1	1.0
Analyst B	-1	1.0
Analyst C	+1	1.0
Senior Partner	-1	2.0
Weighted Sum		**-1.0**

Decision: Don't Buy

Finance Hook: This is exactly how a perceptron works!

What If Machines Could Decide?

The Central Question
In 1943, scientists asked:

"Can we build a machine that learns to make decisions like a brain?"

Why This Matters for Finance:

- Humans are slow and biased
- Markets process millions of data points
- Pattern recognition at scale
- Consistent, emotionless decisions

The Promise
If we could capture how neurons compute:

- Automatic stock screening
- Risk assessment at scale
- Pattern detection in market data
- Learning from historical decisions

The Challenge
How do we translate biological processes into mathematical operations?

This module tells the story of how scientists attempted this translation.

The fundamental question that started neural network research

Module 1 Roadmap

The Complete Journey (4 Modules)

1. **The Perceptron (Today)**
   - Single neuron foundations
   - 1943-1969 history
2. Multi-Layer Perceptrons
   - Stacking layers, activation functions
3. Training Neural Networks
   - Backpropagation, optimization
4. Applications in Finance
   - Stock prediction case study

Today's Module Structure

1. **Historical Context** (1943-1969)
   - McCulloch-Pitts, Hebb, Rosenblatt
2. **Biological Inspiration**
   - From neurons to mathematics
3. **The Perceptron**
   - Intuition, then math
4. **Learning Algorithm**
   - How it adjusts weights
5. **Limitations**
   - XOR problem, AI Winter

Your journey through neural network fundamentals

Learning Objectives

By the end of this module, you will be able to:

1. **Understand biological inspiration**
   - How real neurons inspired artificial ones
   - What we kept and what we simplified
2. **Master the perceptron model**
   - Inputs, weights, sum, activation
   - The decision-making unit
3. **Interpret decision boundaries**
   - Geometric meaning of weights
   - Linear separability concept
4. **Apply the learning algorithm**
   - Weight update rule
   - Convergence conditions
5. **Recognize limitations**
   - XOR problem
   - Why single layers are not enough

Finance Connection: Throughout, we'll use stock classification as our running example.

By the end of this module, you will be able to...

1943: The Mathematical Neuron

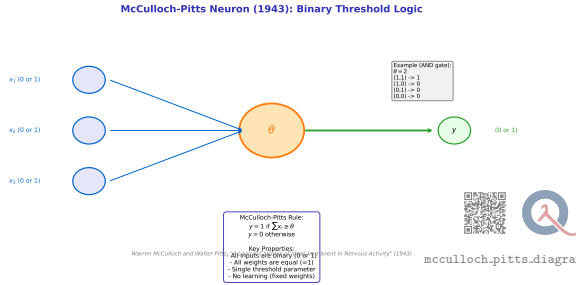
Warren McCulloch & Walter Pitts
In 1943, a neurophysiologist and a logician asked:

"Can we describe what neurons do using mathematics?"

Their paper: "A Logical Calculus of Ideas Immanent in Nervous Activity"

Key Insight:

- Neurons have binary states (fire or not)
- This is like TRUE/FALSE in logic
- Networks of neurons can compute any logical function



Warren McCulloch and Walter Pitts: "A Logical Calculus of Ideas Immanent in Nervous Activity"

The Big Idea: Computation in the Brain

What McCulloch & Pitts Proposed

The brain performs computation through:

1. **Binary Signals**
   - Neurons either fire (1) or don't (0)
   - Like bits in a computer
2. **Threshold Logic**
   - Sum of inputs exceeds threshold → fire
   - Otherwise → stay quiet
3. **Network Composition**
   - Complex behaviors from simple units
   - AND, OR, NOT gates from neurons

Logical Operations with Neurons

AND Gate (threshold = 2):

- Both inputs = 1 → output = 1
- Otherwise → output = 0

OR Gate (threshold = 1):

- Any input = 1 → output = 1
- All inputs = 0 → output = 0

Implication: If neurons compute logic, and computers compute logic, then we can build artificial brains!

If neurons compute, can we build artificial ones?

1949: Hebbian Learning

Donald Hebb's Insight
McCulloch-Pitts neurons were fixed. But how does the brain *learn*?

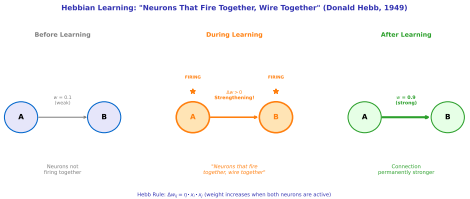
Hebb's Rule (1949):

> *"Neurons that fire together, wire together."*

In Plain Terms:

- If neuron A consistently activates neuron B
- The connection A $\rightarrow$ B grows stronger
- Repeated patterns reinforce pathways

Finance Analogy:
An analyst who repeatedly identifies winning stocks gains more influence in the committee.



hebb_learning_visualizatio

Donald Hebb: "Neurons that fire together, wire together"

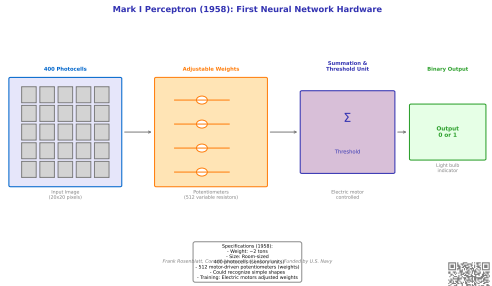
1958: The Perceptron is Born

Frank Rosenblatt at Cornell
Combined McCulloch-Pitts neurons with Hebbian learning into a machine that could *learn from examples*.

The Perceptron:

- A single artificial neuron
- Adjustable connection weights
- Learns to classify patterns
- Implemented in hardware (Mark I)

Key Innovation:
Not just fixed logic gates, but a system that **learns** the right weights from training data.



The Mark I Perceptron used 400 photocells connected to a single layer of neurons with adjustable weights.

Frank Rosenblatt creates a machine that can learn

The New York Times Headline

July 8, 1958 - The New York Times

"New Navy Device Learns By Doing; Psychologist Shows Embryo of Computer Designed to Read and Grow Wiser"

The Promises Made:

- Machines that recognize faces
- Automatic translation of languages
- Systems that "perceive" like humans
- The Navy predicted: walking, talking, self-reproducing machines

The Reality:
The perceptron could classify simple patterns, but the gap between promise and capability was vast.

Lessons for Today

Sound Familiar?

- "AI will replace all jobs"
- "Machines will be smarter than humans by 20XX"
- "This changes everything"

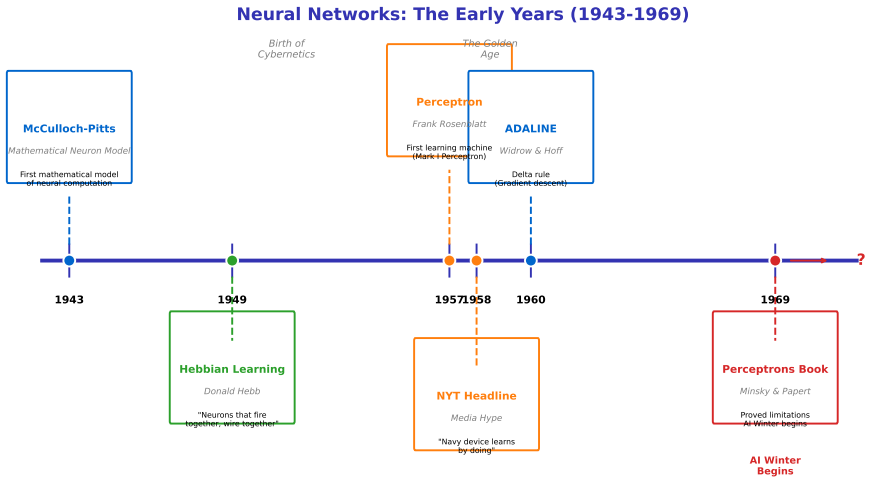
Pattern:

1. Genuine breakthrough
2. Media amplification
3. Overpromising
4. Disappointment
5. "AI Winter"

History repeats...

"New Navy Device Learns By Doing" - The hype cycle begins

Timeline: The Early Years



From mathematical model to practical machine to fundamental limitations

timeline_1943_196

Discussion Question

"The perceptron was funded by the US Navy for military applications. How does funding source shape research direction? Are there parallels in modern AI development?"

Consider:

- Military vs. commercial vs. academic funding
- What problems get prioritized?
- Open vs. closed research
- Today: Tech giants fund most AI research
- Government initiatives (CHIPS Act, etc.)
- Startup ecosystem influence

Think-Pair-Share: 3 minutes

The Biological Neuron

Anatomy of a Real Neuron

1. **Dendrites** (Input)
   - Tree-like branches
   - Receive signals from other neurons
   - Thousands of connections
2. **Cell Body (Soma)** (Processing)
   - Integrates incoming signals
   - Contains the nucleus
   - Determines if neuron fires
3. **Axon** (Output)
   - Long fiber carrying output signal
   - Connects to other neurons
   - All-or-nothing signal

How It Works

1. Signals arrive at dendrites
2. Soma sums the inputs
3. If sum exceeds threshold: neuron **fires**
4. Action potential travels down axon
5. Signal reaches next neurons

Key Numbers:

- Human brain: $\sim$86 billion neurons
- Each neuron: $\sim$7,000 connections
- Total synapses: $\sim$100 trillion

Dendrites receive, soma processes, axon transmits

The Artificial Neuron

Mathematical Abstraction

1. **Inputs** $(x_1, x_2, \ldots, x_n)$
   - Numerical values (features)
   - Replace dendrites
2. **Weights** $(w_1, w_2, \ldots, w_n)$
   - Importance of each input
   - Replace synapse strength
3. **Weighted Sum**
   - $z = \sum_{i=1}^{n} w_i x_i + b$
   - Replace soma integration
4. **Activation Function**
   - $y = f(z)$
   - Replace firing decision

The Complete Model

$$y = f\left(\sum_{i=1}^{n} w_i x_i + b\right)$$

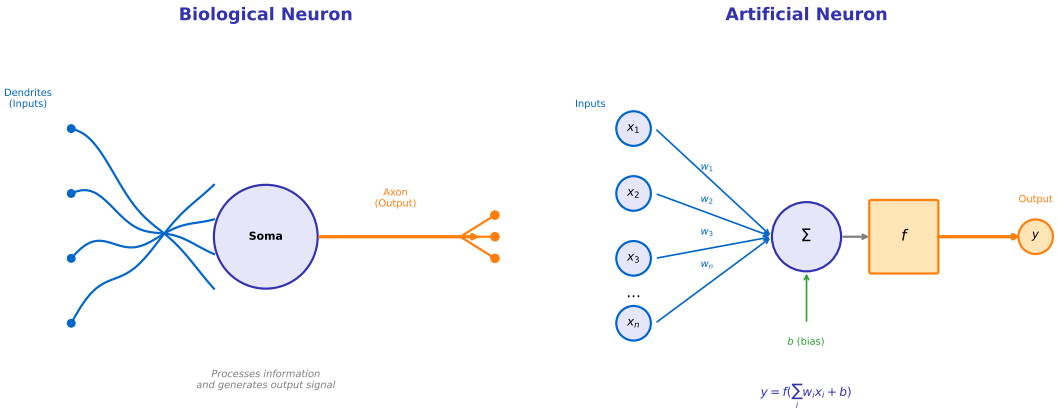
Components:

- x_i: Input features
- w_i: Learnable weights
- b: Bias (threshold adjustment)
- f: Activation function
- y: Output (prediction)

Key Point: The weights are what the network *learns*.

From biology to mathematics: the abstraction trade-off

Biological vs. Artificial: Side by Side



biological_vs_artificial_neuro

What did we keep? What did we simplify?

Finance Analogy: The Analyst

A Financial Analyst as a Neuron

Biology	Finance
Dendrites	Market data feeds
Synapses	Data reliability weights
Soma	Analyst's judgment
Threshold	Conviction level
Axon	"Buy" recommendation

The Process:

1. Receive multiple data points
2. Weight by source quality
3. Aggregate into overall view
4. If conviction $>$ threshold: recommend

Example: Stock Screening

Inputs (Data):

- x_1: P/E ratio $= 15$
- x_2: Revenue growth $= 20\%$
- x_3: Debt/Equity $= 0.5$

Weights (Importance):

- $w_1 = 0.3$ (value focus)
- $w_2 = 0.5$ (growth priority)
- $w_3 = -0.2$ (debt penalty)

Decision:

$$z = 0.3(15) + 0.5(20) - 0.2(0.5) = 14.4$$

If $z > 10$: **Buy**

Inputs (data) -¿ Weights (importance) -¿ Decision (output)

What We Gained from Abstraction

Benefits of Simplification

1. **Mathematical Tractability**
   - We can write equations
   - Analyze behavior formally
   - Prove theorems
2. **Computability**
   - Easy to implement in code
   - Fast computation
   - Scales to millions of units
3. **Trainability**
   - Can adjust weights systematically
   - Gradient-based optimization
   - Learn from data

What We Can Now Do

- Define learning algorithms
- Compute exact outputs
- Train on historical data
- Make predictions on new data
- Analyze decision boundaries

Scale Comparison:

	Brain	**GPU**
Operations/sec	10^{16}	10^{15}
Power	20W	300W
Training time	Years	Hours

Different trade-offs, different capabilities.

Simplification enables computation

What We Lost from Abstraction

Biological Complexity We Ignored

1. **Temporal Dynamics**
   - Real neurons have timing
   - Spike patterns carry information
   - We use static activations
2. **Structural Complexity**
   - Dendrites have local computation
   - Different neuron types
   - We use uniform units
3. **Neurochemistry**
   - Neurotransmitters vary
   - Modulatory systems
   - We use simple multiplication

Implications

What ANNs Cannot Do (Well):

- Energy efficiency of brain
- One-shot learning
- Continuous adaptation
- Common sense reasoning

The Trade-off:

Tractability ↑ Realism ↓

Artificial neurons are inspired by biology, not copies of it.

The brain does far more than our models capture

The Simplest Decision Maker

What is a Perceptron?
The simplest possible neural network:

- One artificial neuron
- Multiple inputs, one output
- Binary decision: Yes or No

Think of it as:

- A filter for data
- A simple classifier
- A linear decision maker

Finance Application:
Stock screener that outputs "Buy" or "Don't Buy" based on financial metrics.

Real-World Examples

Email Spam Filter:

- Inputs: word frequencies
- Output: spam or not spam

Loan Approval:

- Inputs: income, credit score, debt
- Output: approve or reject

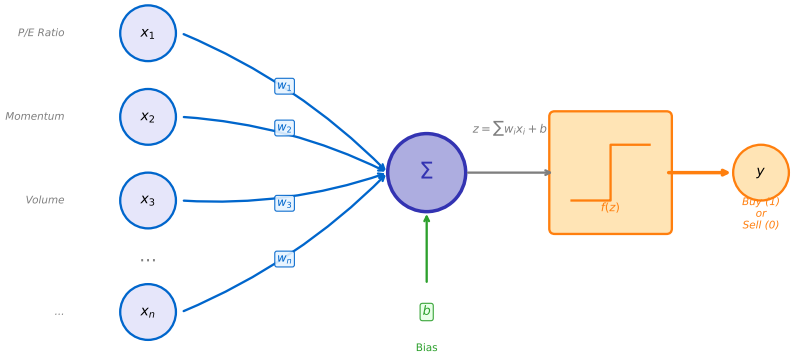
Stock Screening:

- Inputs: P/E, momentum, volume
- Output: buy or pass

All these are binary classification problems that a perceptron can solve (if the data is linearly separable).

A single perceptron is a stock screening filter

The Perceptron: Architecture



Finance Scenario: Buy or Sell?

Problem Setup
You want to build a simple stock screener:

- **Goal**: Decide Buy or Pass
- **Data**: Historical financial metrics
- **Method**: Perceptron classifier

Available Features:

1. P/E Ratio (valuation)
2. 6-month momentum (%)
3. Average daily volume
4. Debt-to-Equity ratio
5. Earnings surprise (%)

The Question
Given these features for a new stock, should we add it to our portfolio?

Example Stock:

- P/E = 18
- Momentum = +12%
- Volume = 2M shares
- D/E = 0.8
- Surprise = +5%

Traditional Approach:
Analyst manually weighs factors and decides.

Perceptron Approach:
Learn the weights from historical winners/losers.

Given financial indicators, should we buy this stock?

Inputs: The Raw Data

What Are Inputs?
Each input x_i is a numerical feature:

- A measurement
- A statistic
- A signal

In Finance:

- Price-based: returns, volatility
- Fundamental: P/E, ROE, debt ratios
- Technical: RSI, moving averages
- Sentiment: news scores, analyst ratings

Key Requirement:
All inputs must be **numerical**. Categorical data needs encoding.

Notation

For a stock with n features:

$$\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}$$

Example (n=3):

$$\mathbf{x} = \begin{pmatrix} 18 \\ 0.12 \\ 0.8 \end{pmatrix} = \begin{pmatrix} \text{P/E} \\ \text{Momentum} \\ \text{D/E} \end{pmatrix}$$

Note: Features often need **normalization** (covered in Module 3).

What data feeds into our decision?

Weights: The Importance Factors

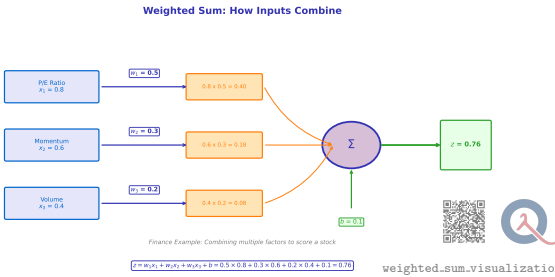
What Are Weights?

Each weight w_i represents:

- Importance of input x_i
- Direction of influence
- Learned from data

Interpretation:

- $w_i > 0$: Higher x_i pushes toward "Buy"
- $w_i < 0$: Higher x_i pushes toward "Sell"
- $|w_i|$ large: Strong influence
- $|w_i|$ small: Weak influence



"Not all data is equally important" - weights encode importance

Discussion Question

"If you could only look at 3 metrics for a stock, which would you choose and why? How would you weight them?"

Consider:

Value Investor Might Choose:

- P/E ratio ($w = 0.5$)
- Book value ($w = 0.3$)
- Dividend yield ($w = 0.2$)

Growth Investor Might Choose:

- Revenue growth ($w = 0.5$)
- Momentum ($w = 0.3$)
- Market share ($w = 0.2$)

Key Insight: Different investors would assign different weights. The perceptron *learns* these weights from historical performance.

Think-Pair-Share: 3 minutes

The Weighted Sum: Adding Up Evidence

Computing the Weighted Sum

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

What This Means:

- Multiply each input by its weight
- Sum all the products
- Add the bias term b
- Result: a single "score"

The Bias b:

- Shifts the decision threshold
- Like a "base rate" or prior
- Can be thought of as $w_0 \cdot x_0$ where $x_0 = 1$

Worked Example

Inputs:

- $x_1 = 0.8$ (normalized P/E)
- $x_2 = 0.6$ (normalized momentum)

Weights:

- $w_1 = 0.5$
- $w_2 = 0.7$
- $b = -0.3$

Calculation:

$$\begin{aligned} z &= w_1 x_1 + w_2 x_2 + b \\ &= (0.5)(0.8) + (0.7)(0.6) + (-0.3) \\ &= 0.4 + 0.42 - 0.3 \\ &= \mathbf{0.52} \end{aligned}$$

Combine all weighted inputs into a single score

Analogy: The Voting Committee

The Perceptron as a Committee

Member	Vote	Weight	Contribution
P/E analyst	+1	0.5	+0.5
Momentum	+1	0.7	+0.7
Bias (skeptic)	-1	0.3	-0.3
Total			**+0.9**

If Total $>$ 0: Committee recommends **Buy**

Key Insight:
The perceptron is just a weighted voting system where the weights are learned from data.

Why This Works

Traditional Committee:

- Human experts set weights
- Based on experience/intuition
- May have biases
- Hard to scale

Perceptron Committee:

- Weights learned from data
- Based on historical performance
- Consistent application
- Scales to any volume

Trade-off: Data-driven weights may not capture regime changes or rare events.

Some votes count more than others

The Threshold: Making the Call

The Activation Function

After computing z, we need a final decision.

Step Function:

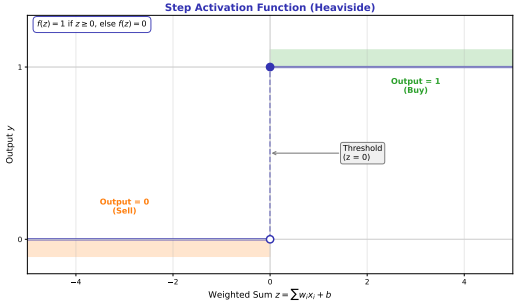
$$f(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}$$

Interpretation:

- $z \geq 0$: Evidence favors "Buy" $\rightarrow$ output 1
- $z < 0$: Evidence favors "Sell" $\rightarrow$ output 0

Why Step Function?

- Binary classification needs binary output
- Mimics neuron firing (all-or-nothing)
- Simple to implement



step_functio

Above threshold = Buy, Below threshold = Sell

The Complete Perceptron Flow

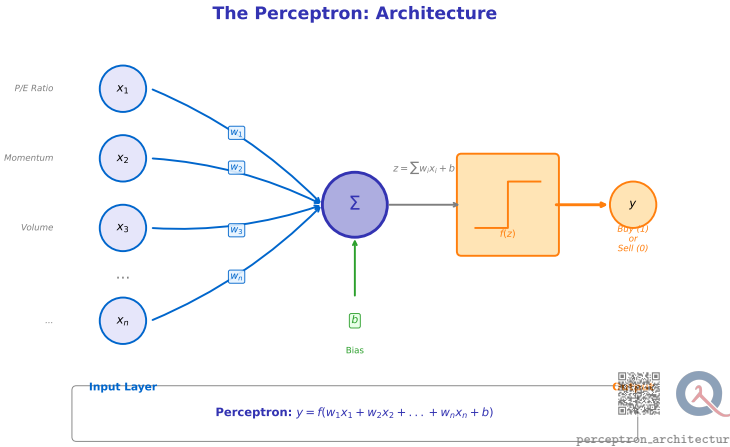
The Pipeline

1. **Input**: Receive features **x**
2. **Weight**: Multiply by **w**
3. **Sum**: Add all products + bias
4. **Activate**: Apply step function
5. **Output**: Return prediction

Compact Notation:

$$y = f(\mathbf{w}^T\mathbf{x} + b)$$

where $\mathbf{w}^T\mathbf{x} = \sum_i w_i x_i$



Inputs -¿ Weights -¿ Sum -¿ Threshold -¿ Decision

Now Let's Formalize

What You Already Know

From the intuition section:

- Inputs are weighted
- Weights encode importance
- Sum is compared to threshold
- Output is binary

What's Next

- Precise mathematical notation
- Geometric interpretation
- Foundation for learning algorithm

Why Math Matters

Without Math:

- "The network kind of learns"
- "Adjust weights somehow"
- "It works, probably"

With Math:

- Precise learning rules
- Convergence guarantees
- Understanding of limitations

The next 8 slides formalize what you already understand intuitively.

You understand the intuition. Let's write it precisely.

The Perceptron Equation

Scalar Form

$$y = f\left(\sum_{i=1}^{n} w_i x_i + b\right)$$

where f is the step function:

$$f(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

Vector Form

$$y = f(\mathbf{w}^T \mathbf{x} + b)$$

where:

- $\mathbf{w} = (w_1, \ldots, w_n)^T$
- $\mathbf{x} = (x_1, \ldots, x_n)^T$

Alternative Notation

We can absorb the bias into weights:

$$\tilde{\mathbf{w}} = \begin{pmatrix} b \\ w_1 \\ \vdots \\ w_n \end{pmatrix}, \quad \tilde{\mathbf{x}} = \begin{pmatrix} 1 \\ x_1 \\ \vdots \\ x_n \end{pmatrix}$$

Then:

$$y = f(\tilde{\mathbf{w}}^T \tilde{\mathbf{x}})$$

Note: This "bias trick" simplifies notation but they are equivalent.

The complete mathematical model

Unpacking the Mathematics

Term by Term

Symbol	Meaning
x_i	Input feature i
w_i	Weight for feature i
b	Bias (threshold shift)
z	Weighted sum (pre-activation)
f	Activation function
y	Output prediction
n	Number of features

Dimensions:

- $\mathbf{x} \in \mathbb{R}^n$
- $\mathbf{w} \in \mathbb{R}^n$
- $b, z, y \in \mathbb{R}$

What Gets Learned?

Learned (trainable):

- Weights $w_1, \ldots, w_n$
- Bias b

Fixed (architecture):

- Number of inputs n
- Activation function f

Given (data):

- Input values $x_1, \ldots, x_n$
- Target labels (for training)

Total Parameters: $n + 1$
(For a 3-feature perceptron: 4 parameters)

Each symbol has a meaning

The Bias Term

What Does Bias Do?
Without bias ($b = 0$):

$$z = \mathbf{w}^T \mathbf{x}$$

The decision boundary passes through origin.

With bias ($b \neq 0$):

$$z = \mathbf{w}^T \mathbf{x} + b$$

The decision boundary can be anywhere.

Interpretation:

- $b > 0$: Default toward "Buy"
- $b < 0$: Default toward "Sell"
- Like a prior belief

Finance Analogy

Without Bias:
"I have no opinion until I see data"

With Positive Bias:
"I'm generally bullish; you need to convince me to sell"

With Negative Bias:
"I'm skeptical by default; you need strong evidence to buy"

Key Point: Bias shifts the "bar" that evidence must clear. It's learned from data just like weights.

Bias shifts the decision threshold

The Step Activation Function

Formal Definition
The Heaviside step function:

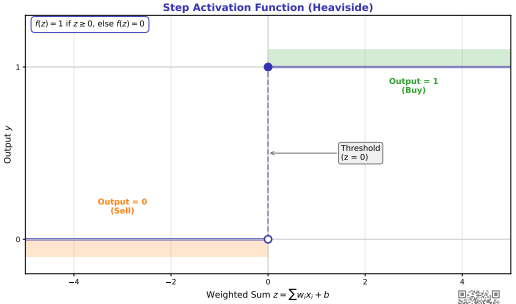
$$f(z) = \mathbf{1}_{z \geq 0} = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}$$

Properties:

- Output $\in \{0, 1\}$
- Discontinuous at $z = 0$
- Not differentiable (problem for gradient-based learning!)

Variants:

- Sign function: outputs $\{-1, +1\}$
- Same idea, different labels



Preview: The non-differentiability of the step function is why we'll need smoother activations (sigmoid, ReLU) in later modules.

Binary output: yes or no

Geometric Interpretation: The Decision Boundary

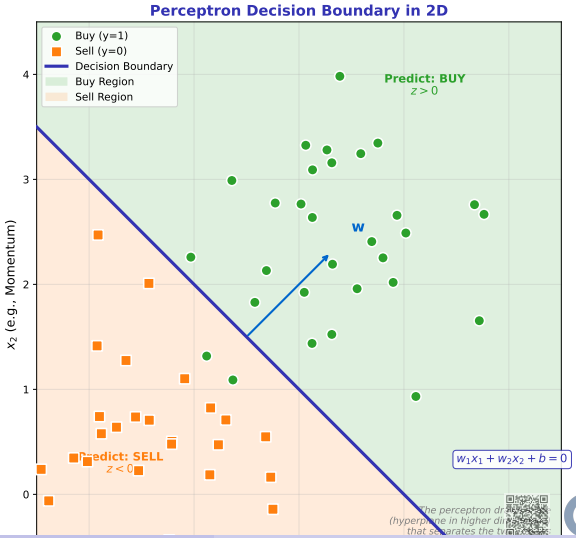
The Perceptron as a Hyperplane
The equation $\mathbf{w}^T\mathbf{x} + b = 0$ defines a hyperplane:

- In 2D: a line
- In 3D: a plane
- In nD: a hyperplane

Regions:

- $\mathbf{w}^T\mathbf{x} + b > 0$: Class 1 (Buy)
- $\mathbf{w}^T\mathbf{x} + b < 0$: Class 0 (Sell)
- $\mathbf{w}^T\mathbf{x} + b = 0$: Decision boundary

Weight Vector Direction:
$\mathbf{w}$ is perpendicular to the decision boundary, pointing toward the positive class.



Finance Example: Classifying Stocks

Two-Feature Stock Screener
Features:

- x_1: P/E ratio (normalized)
- x_2: 6-month momentum (%)

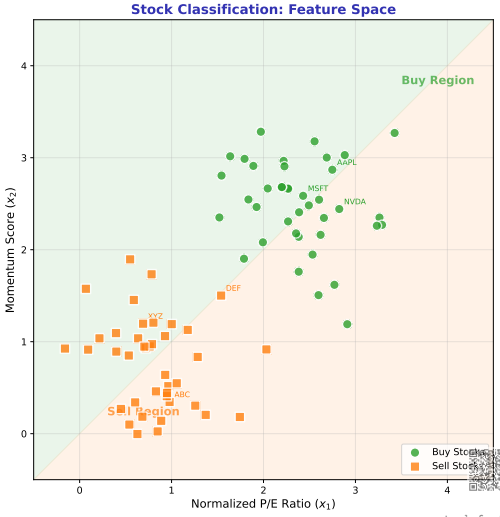
Classes:

- Green: Outperformed (Buy)
- Red: Underperformed (Sell)

Goal:
Find w_1, w_2, b such that:

$$w_1 \cdot \text{P/E} + w_2 \cdot \text{Momentum} + b = 0$$

separates the classes.



stock_features_scatte

Challenge: Can we draw a single line to separate Buy from Sell?

The Decision Boundary Formula

In 2D: The Line Equation

From $w_1 x_1 + w_2 x_2 + b = 0$:

$$x_2 = -\frac{w_1}{w_2} x_1 - \frac{b}{w_2}$$

This is a line with:

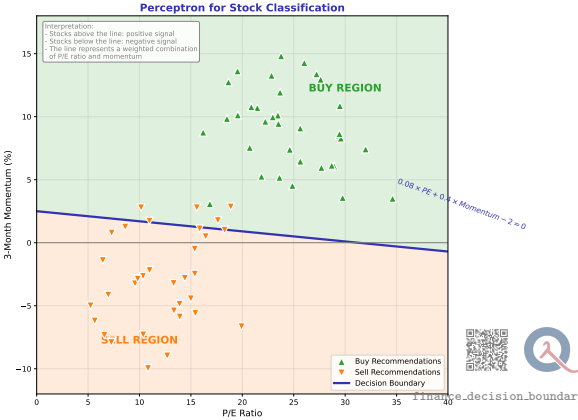
- Slope: $-\frac{w_1}{w_2}$
- Intercept: $-\frac{b}{w_2}$

Example:

If $w_1 = 2, w_2 = 1, b = -3$:

$$x_2 = -2x_1 + 3$$

Stocks above this line: Buy
Stocks below this line: Sell



The line that separates buy from sell

How Does the Perceptron Learn?

The Learning Problem
Given:

- Training data: $\{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^{m}$
- Each $\mathbf{x}^{(i)}$: feature vector
- Each $y^{(i)} \in \{0, 1\}$: true label

Find:

- Weights $\mathbf{w}$
- Bias b
- Such that predictions match labels

The Approach:
Start with random weights, then iteratively adjust based on mistakes.

The Core Idea

If prediction is correct:
Do nothing. Weights are fine.

If prediction is wrong:
Adjust weights to make this example more likely to be correct next time.

Repeat:
Keep cycling through training data until no mistakes (or convergence).

Key Insight: Learning = adjusting weights based on errors.

Learning = adjusting weights based on mistakes

Learning from Mistakes

Two Types of Errors

False Negative ($\hat{y} = 0$, $y = 1$):

- Predicted Sell, should be Buy
- The score z was too low
- Need to *increase* score for this $\mathbf{x}$
- Solution: Add $\mathbf{x}$ to $\mathbf{w}$

False Positive ($\hat{y} = 1$, $y = 0$):

- Predicted Buy, should be Sell
- The score z was too high
- Need to *decrease* score for this $\mathbf{x}$
- Solution: Subtract $\mathbf{x}$ from $\mathbf{w}$

Visual Intuition

Before update:
Point is on wrong side of boundary.

After update:
Boundary moves to include the point on the correct side.

The Update Rule:

$$\mathbf{w}_{\text{new}} = \mathbf{w}_{\text{old}} + (y - \hat{y}) \cdot \mathbf{x}$$

Check:

- If $y = 1, \hat{y} = 0$: add $\mathbf{x}$
- If $y = 0, \hat{y} = 1$: subtract $\mathbf{x}$
- If $y = \hat{y}$: no change

Each mistake is a learning opportunity

The Learning Rule: Intuition

Why Adding x Works

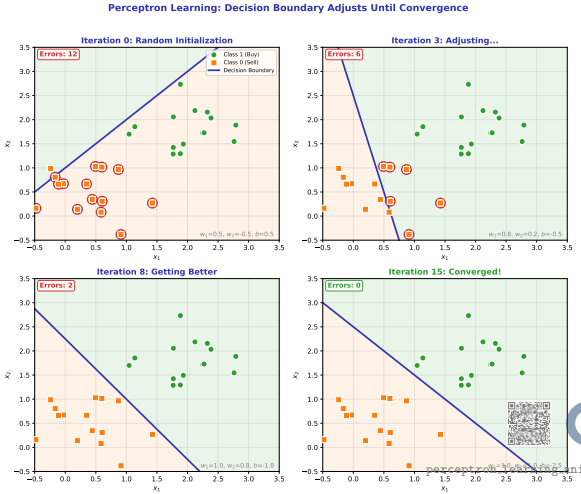
For a false negative (missed Buy):

- Current: $\mathbf{w}^T\mathbf{x} + b < 0$
- After adding $\mathbf{x}$ to $\mathbf{w}$:
- New score: $(\mathbf{w} + \mathbf{x})^T\mathbf{x} + b$
- $= \mathbf{w}^T\mathbf{x} + \mathbf{x}^T\mathbf{x} + b$
- $= \mathbf{w}^T\mathbf{x} + \|\mathbf{x}\|^2 + b$

Since $\|\mathbf{x}\|^2 > 0$, the new score is higher!

Geometrically:

Adding $\mathbf{x}$ rotates the decision boundary toward classifying $\mathbf{x}$ correctly.



If wrong, move the boundary

The Perceptron Learning Rule

The Update Equations
For each training example $(\mathbf{x}, y)$:

Weight update:

$$\mathbf{w} \leftarrow \mathbf{w} + \eta(y - \hat{y})\mathbf{x}$$

Bias update:

$$b \leftarrow b + \eta(y - \hat{y})$$

where:

- $\eta > 0$ is the learning rate
- $\hat{y} = f(\mathbf{w}^T\mathbf{x} + b)$ is prediction
- y is true label

The Complete Algorithm

1. Initialize $\mathbf{w} = \mathbf{0}$, $b = 0$
2. **repeat**:
   a. For each $(\mathbf{x}^{(i)}, y^{(i)})$ in training set:
   b. Compute $\hat{y}^{(i)} = f(\mathbf{w}^T\mathbf{x}^{(i)} + b)$
   c. If $\hat{y}^{(i)} \neq y^{(i)}$:
   $\mathbf{w} \leftarrow \mathbf{w} + \eta(y^{(i)} - \hat{y}^{(i)})\mathbf{x}^{(i)}$
   $b \leftarrow b + \eta(y^{(i)} - \hat{y}^{(i)})$
3. **until** no errors (or max iterations)

The mathematical update rule

The Learning Rate

What is η?
The learning rate controls step size:

- How much weights change per update
- Typical values: 0.01 to 1.0
- For perceptron: often $\eta = 1$

Effects:
η too small:

- Very slow learning
- Many iterations needed
- But stable

η too large:

- May overshoot
- Oscillate around solution
- But faster initially

For the Perceptron

Good news:
For linearly separable data, the perceptron converges regardless of $\eta > 0$.

Why?
The convergence theorem (next slides) guarantees finding a solution if one exists.

In Practice:
$\eta = 1$ is common for perceptron. Learning rate matters more for:

- Gradient descent (Module 3)
- Non-separable data
- Multi-layer networks

Step size matters: too big or too small both cause problems

Worked Example: Stock Classification

Setup

Two stocks, two features:

- $\mathbf{x}^{(1)} = (0.5, 0.8)$, $y^{(1)} = 1$ (Buy)
- $\mathbf{x}^{(2)} = (0.2, 0.3)$, $y^{(2)} = 0$ (Sell)

Initialize: $\mathbf{w} = (0, 0)$, $b = 0$, $\eta = 1$

Iteration 1: Example 1

- $z = 0 \cdot 0.5 + 0 \cdot 0.8 + 0 = 0$
- $\hat{y} = f(0) = 1$ (threshold at 0)
- $y = 1$, correct! No update.

Iteration 1: Example 2

- $z = 0$, $\hat{y} = 1$
- $y = 0$, wrong!
- $\mathbf{w} \leftarrow (0, 0) + 1(0 - 1)(0.2, 0.3) = (-0.2, -0.3)$
- $b \leftarrow 0 + 1(0 - 1) = -1$

Iteration 2: Example 1

- $z = -0.2(0.5) - 0.3(0.8) - 1 = -1.34$
- $\hat{y} = 0$
- $y = 1$, wrong!
- $\mathbf{w} \leftarrow (-0.2, -0.3) + (0.5, 0.8) = (0.3, 0.5)$
- $b \leftarrow -1 + 1 = 0$

Iteration 2: Example 2

- $z = 0.3(0.2) + 0.5(0.3) + 0 = 0.21$
- $\hat{y} = 1$, $y = 0$, wrong!
- $\mathbf{w} \leftarrow (0.3, 0.5) - (0.2, 0.3) = (0.1, 0.2)$
- $b \leftarrow 0 - 1 = -1$

Continue until convergence...

Following the math with real numbers

Convergence: Does It Always Work?

The Perceptron Convergence Theorem
Theorem (Rosenblatt, 1962):
If the training data is **linearly separable**, the perceptron learning algorithm will find a separating hyperplane in a **finite** number of updates.

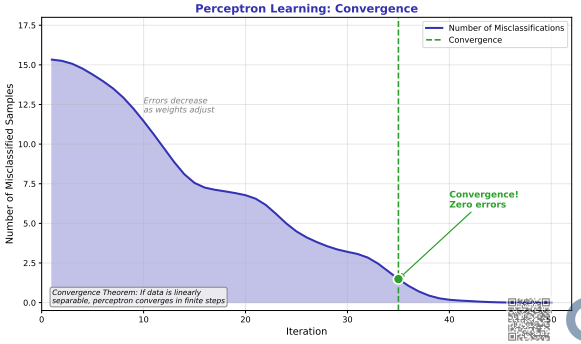
Key Conditions:

- Data must be linearly separable
- Learning rate $\eta > 0$
- Cycling through all examples

Bound on Updates:

$$\text{mistakes} \leq \frac{R^2}{\gamma^2}$$

where R = max norm, γ = margin



convergence_plo

The perceptron convergence theorem guarantees finding a solution IF one exists

Discussion Question

"What happens when data isn't linearly separable in financial markets? Can you think of examples?"

Consider:

Examples of Non-Separable Data:

- High P/E growth stocks AND low P/E value stocks both outperform
- Medium-risk investments underperform both conservative and aggressive
- "Buy the rumor, sell the news" patterns

What Happens to the Perceptron?

- Never converges
- Oscillates forever
- Best we can do: minimize errors
- Need something more powerful...

Foreshadowing: This is exactly why we need **multi-layer** networks (Module 2).

Think-Pair-Share: 3 minutes

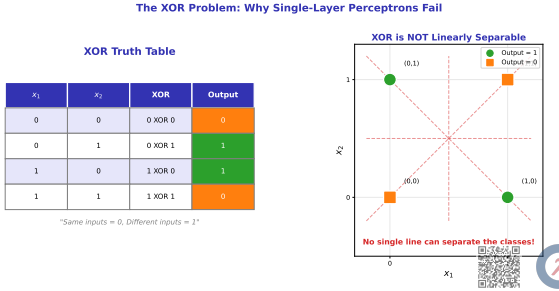
The XOR Problem

The Exclusive OR Function

x_1	x_2	XOR
0	0	0
0	1	1
1	0	1
1	1	0

In Words:
Output is 1 if inputs are *different*, 0 if inputs are *same*.

The Challenge:
Try to draw a single line that separates the 1s from the 0s...



Some patterns cannot be separated by a single line

Why XOR Cannot Be Solved

Geometric Impossibility

Perceptron decision boundary:
$w_1 x_1 + w_2 x_2 + b = 0$
This is always a **straight line**.

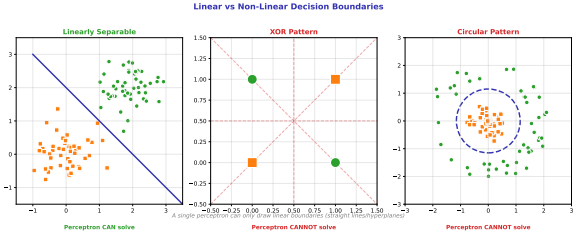
XOR requires:
A boundary that curves or has multiple segments.

Linear vs Non-Linear
Linearly Separable:

- AND, OR, NAND, NOR
- One line can separate

Not Linearly Separable:

- XOR, XNOR
- No single line works



linear_vs_nonlinear_pattern

No single hyperplane can separate XOR

1969: The Critique That Changed Everything

Minsky and Papert's Book
"Perceptrons: An Introduction to Computational Geometry" (1969)

Key Arguments:

1. Single-layer perceptrons cannot compute XOR
2. Many important functions are non-linear
3. No known training algorithm for multi-layer networks
4. Scaling limitations

The Impact:
The book was rigorous and influential. It convinced funding agencies that neural networks were a dead end.

The Controversy

Valid Points:

- Single layers *are* limited
- XOR problem is real
- No training algorithm existed (then)

Overstated Points:

- "Neural networks can't work"
- Implied multi-layer networks wouldn't help
- Discouraged research for 15+ years

Lesson: Valid criticism of current methods shouldn't stop research into future improvements.

Marvin Minsky and Seymour Papert: "Perceptrons" book

The First AI Winter Begins

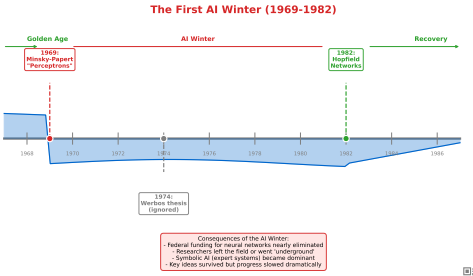
The Collapse
After 1969:

- Funding dried up
- Researchers left the field
- "Neural networks don't work"
- Symbolic AI took over

Duration: 1969 to ∼1982

What Survived:

- A few dedicated researchers
- Theoretical work continued quietly
- Hopfield networks (1982)
- Backpropagation (1986)



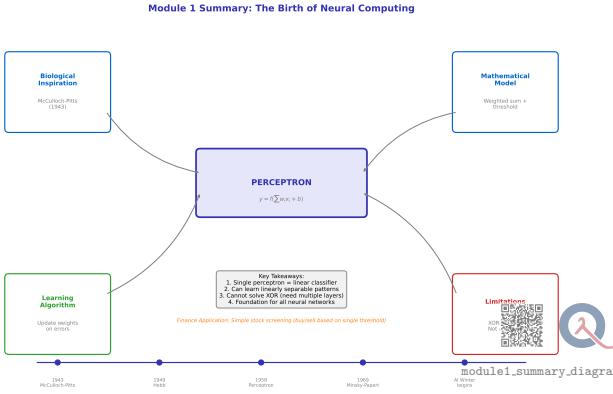
ai_winter_timelin

1969-1982: The dark ages of neural network research

Module 1: Key Takeaways

What We Learned

1. **Historical Foundation**
   - McCulloch-Pitts (1943): neurons compute
   - Hebb (1949): learning strengthens connections
   - Rosenblatt (1958): perceptron learns
2. **The Perceptron Model**
   - Weighted sum + threshold
   - Linear decision boundary
   - Learns from mistakes
3. **Limitations**
   - Only linearly separable problems
   - XOR is impossible
   - Led to AI Winter



From biological inspiration to mathematical limitation

Preview: Module 2

"What if we stack multiple perceptrons?"

The Problem We Face
Single perceptrons can only solve linearly separable problems. Real financial data is rarely that simple.

The Solution Preview:

- Add "hidden" layers
- Non-linear activation functions
- Multi-Layer Perceptrons (MLPs)

Coming in Module 2:

- How XOR gets solved
- MLP architecture
- Activation functions (sigmoid, ReLU)
- Universal Approximation Theorem
- Loss functions

Spoiler: Adding just one hidden layer changes everything.

Mathematical details for this module: See Appendix A (Perceptron Convergence Proof)

Next: Solving XOR with Multi-Layer Perceptrons

Module 1: The Birth of Neural Computing

From Biological Inspiration to the Perceptron (1943-1969)

Neural Networks for Finance

BSc Lecture Series

November 28, 2025

Where We Left Off

Module 1 Summary
We learned that a single perceptron:

- Takes weighted inputs
- Applies a threshold
- Outputs a binary decision
- Can only draw **linear** boundaries

The Perceptron Equation:

$$y = f\left(\sum_{i=1}^{n} w_i x_i + b\right)$$

The Problem
The perceptron cannot solve XOR or any non-linearly separable problem.

The AI Winter:

- Minsky-Papert (1969) critique
- Funding dried up
- "Neural networks don't work"

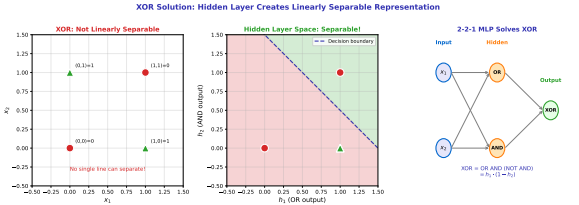
Today's Question:
What if we stack multiple perceptrons together?

The perceptron: powerful but limited

The XOR Problem Revisited

Why One Line Isn't Enough

x_1	x_2	XOR
0	0	0
0	1	1
1	0	1
1	1	0



The Geometry:

- Opposite corners have same label
- No single line can separate them
- We need *multiple* boundaries

xor_solution_ml

Some patterns require more than a single line

The Finance Parallel

Single Analyst (Perceptron)
One junior analyst screening stocks:

- Looks at a few metrics
- Applies simple rules
- Makes direct decisions
- Limited perspective

Limitation:
"Buy if P/E < 15 AND momentum > 0"
This is a single linear rule.

Investment Team (MLP)
A hierarchical team:

- Junior analysts find patterns
- Senior analysts synthesize
- CIO makes final call
- Complex reasoning emerges

Capability:
"Consider value metrics, momentum signals, AND market regime together"
Multiple non-linear patterns.

Key Insight: Hierarchical processing enables complex pattern recognition.

A single analyst sees simple patterns. A team sees complex ones.

Module 2 Roadmap

What We'll Cover

1. **Historical Context**
   - AI Winter survival
   - Backprop rediscovery (1986)
2. **MLP Architecture**
   - Intuition: The firm analogy
   - Math: Matrix notation
3. **Activation Functions**
   - Why non-linearity matters
   - Sigmoid, Tanh, ReLU
4. **Universal Approximation**
   - The fundamental theorem
   - Implications and limits
5. **Loss Functions**
   - MSE for regression
   - Cross-entropy for classification

Learning Objectives:

- Understand MLP architecture
- Master matrix notation
- Know when to use which activation
- Appreciate universal approximation

From single perceptron to universal function approximation

The AI Winter (1969-1982)

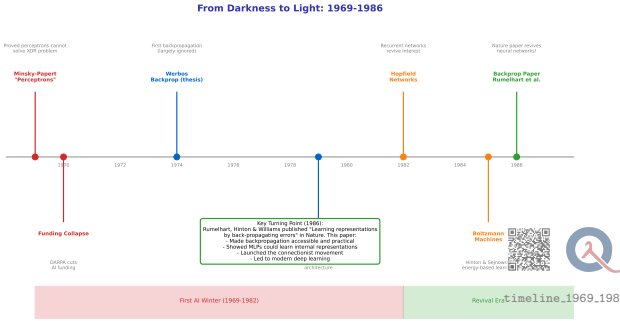
After Minsky-Papert

The neural network winter:

- Government funding cut
- Researchers moved to other fields
- “Connectionism is dead”
- Symbolic AI dominated

The Mood:

- Perceptrons can’t solve XOR
- Multi-layer networks exist but...
- No efficient training algorithm
- Why bother?



After Minsky-Papert, neural network research nearly died

Underground Progress

Paul Werbos (1974)
PhD thesis at Harvard:

- Derived backpropagation
- For general non-linear systems
- Applied to neural networks
- Largely ignored

Why Ignored?

- Published in economics, not CS
- AI winter was at its coldest
- No computational power to test
- No community to spread ideas

Parallel Discoveries

1970s:

- Linnainmaa: automatic differentiation
- Control theory: similar ideas

1980s:

- Parker (1982): rediscovery
- LeCun (1985): independent work
- Rumelhart/Hinton/Williams (1986): fame

Lesson: Good ideas can be discovered multiple times before they "take off."

The key ideas existed but were ignored

1982: Hopfield Networks

John Hopfield
A physicist (not AI researcher) revived interest:

- Connected neural networks to physics
- Energy-based formulation
- Published in PNAS (prestigious)
- Showed neural nets could store memories

The Impact:

- Legitimized neural network research
- Attracted physicists to the field
- New mathematical tools
- Funding started returning

Why Physics Helped

Physics Connection:

- Neurons $\leftrightarrow$ spins in magnets
- Learning $\leftrightarrow$ energy minimization
- Networks $\leftrightarrow$ statistical mechanics

Finance Parallel:
Physicists would later apply similar ideas to:

- Option pricing
- Market dynamics
- Risk modeling
- Quantitative finance

John Hopfield: Physicist rediscovers neural networks

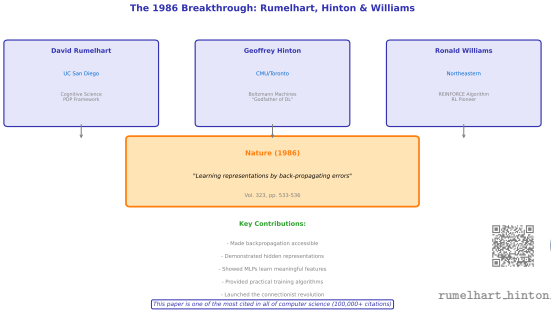
1986: The Backpropagation Paper

The Paper That Changed Everything
Rumelhart, Hinton, Williams in Nature (1986):
"Learning representations by back-propagating errors"

Key Contributions:

- Clear algorithm presentation
- Demonstrated on real problems
- Published in high-impact journal
- Well-communicated to broad audience

The Result:
Neural network renaissance begins.



Nature paper: "Learning representations by back-propagating errors"

What Made 1986 Different?

Werbos (1974)

+ Correct algorithm
+ General framework
- Wrong field (economics)
- No demonstrations
- No community
- No computers

Rumelhart et al. (1986)

+ Correct algorithm
+ Clear presentation
+ Compelling demos
+ High-profile venue (Nature)
+ Growing community
+ Computers available

Lesson for Researchers:
Being right isn't enough. You need:

- The right timing
- The right communication
- The right audience
- The right technology

The right idea at the right time with the right people

Discussion Question

"Backpropagation was discovered multiple times (1974, 1982, 1986). Why do some discoveries get ignored while others take off? What role did timing play?"

Consider:

- Publication venue matters
- Community readiness
- Computational infrastructure
- Demonstration quality
- Today: transformers (2017) exploded
- LSTMs existed since 1997
- What changed?

Think-Pair-Share: 3 minutes

The Neural Network Renaissance

After 1986
Neural networks were back:

- Funding returned
- New conferences (NIPS, now NeurIPS)
- "Connectionism" movement
- Real applications emerged

Key Milestones:

- 1989: LeNet for digit recognition
- 1990s: Speech recognition
- 1990s: Financial applications begin

But Challenges Remained

Not everything worked:

- Deep networks hard to train
- Vanishing gradients
- Limited compute power
- Another "winter" in 2000s

True Revolution: 2012
AlexNet on ImageNet marked the deep learning era. (Module 4)

But first, we need to understand the architecture...

Neural networks are back - and this time they can learn

The Investment Firm Analogy

Hierarchical Decision Making

Level 1: Junior Analysts (Hidden Layer 1)

- Look at raw data
- Find basic patterns
- "This looks like a value stock"
- "This has momentum"

Level 2: Senior Analysts (Hidden Layer 2)

- Combine junior reports
- Higher-level synthesis
- "Value + momentum = quality"

Level 3: CIO (Output Layer)

- Final buy/sell decision
- Combines all analyses
- Single decision point

Key Properties:

1. Information flows upward
2. Each level adds abstraction
3. Later layers see patterns in patterns
4. Final layer integrates everything

This is an MLP!

Hierarchical decision making

Input Layer: The Data Gatherers

The Input Layer
What it does:

- Receives raw data
- One neuron per feature
- No computation
- Just passes data forward

In Finance:

- P/E ratio
- Momentum (returns)
- Volume
- Volatility
- Sector indicators
- Market cap

Notation

$$\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}$$

where:

- n = number of features
- x_i = value of feature i

Example (n=4):

$$\mathbf{x} = \begin{pmatrix} 15 \\ 0.08 \\ 1.2M \\ 0.25 \end{pmatrix} = \begin{pmatrix} \text{P/E} \\ \text{Return} \\ \text{Volume} \\ \text{Vol} \end{pmatrix}$$

The input layer receives raw information

Hidden Layers: The Pattern Finders

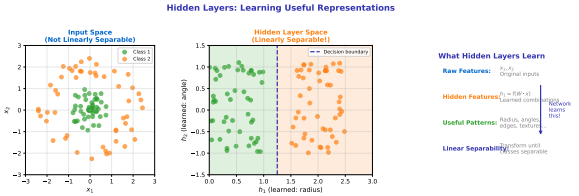
What Hidden Layers Do

They discover intermediate patterns:

- Not explicitly programmed
- Emerge from training
- Often uninterpretable
- But highly useful

Each Hidden Neuron:

- Receives weighted inputs
- Applies activation function
- Outputs a single number
- "Detects" a specific pattern



hidden_layer_representation

"They see things in the data you didn't explicitly ask for"

Finance Example: What Hidden Layers Find

Hypothetical Hidden Neurons

Hidden Neuron 1: "Value Detector"

- Positive weight on low P/E
- Positive weight on high book value
- Activates for value stocks

Hidden Neuron 2: "Momentum Detector"

- Positive weight on recent returns
- Positive weight on volume
- Activates for trending stocks

Hidden Neuron 3: "Risk Detector"

- Positive weight on volatility
- Positive weight on debt
- Activates for risky stocks

The Output Layer

Combines hidden neuron outputs:

$$\text{Buy} = f(w_1 \cdot \text{Value} + w_2 \cdot \text{Momentum} - w_3 \cdot \text{Risk})$$

Key Insight:

We never told the network what "value" or "momentum" means. It *discovered* these concepts from data.

Caveat:

Real hidden neurons may not be this interpretable. They might detect patterns we can't name.

Hidden neurons learn abstract concepts

Output Layer: The Final Decision

The Output Layer
Takes hidden representations and produces:

- Classification: probability of class
- Regression: continuous prediction
- Multiple outputs possible

For Binary Classification:
Single output neuron with sigmoid:

$$\hat{y} = \sigma(w^T h + b)$$

Output $\in (0, 1)$ interpreted as probability.

For Regression:
Single output neuron with no activation (or linear):

$$\hat{y} = w^T h + b$$

Output is predicted value.

Finance Examples

Buy/Sell Classification:

- Output: $P(\text{Buy})$
- If > 0.5: recommend Buy
- If < 0.5: recommend Sell

Return Prediction:

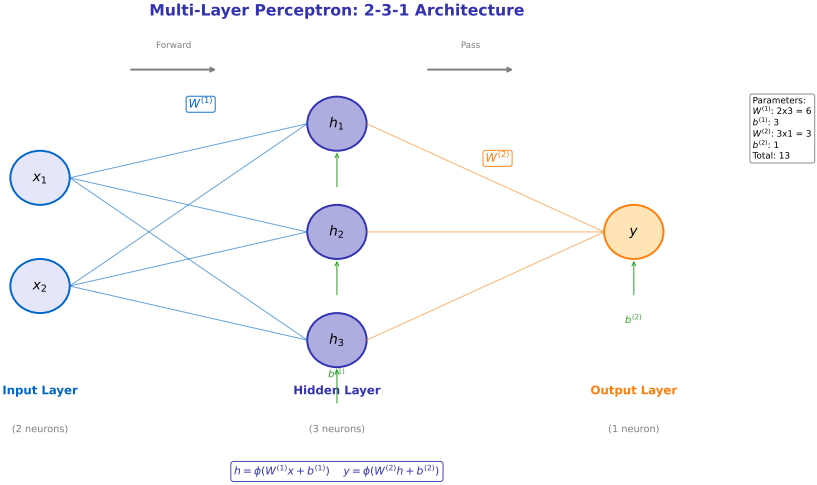
- Output: predicted return
- Could be next-day, next-month
- Continuous value

Multi-Class (Sector):

- Multiple output neurons
- Softmax activation
- Each output = probability of sector

The output layer synthesizes everything into a decision

The Full MLP Architecture



Why Are They Called "Hidden"?

We Don't Observe Them Directly
Observable:

- Input layer: the features we provide
- Output layer: the prediction we get

Hidden:

- Internal representations
- Not directly specified
- Learned automatically
- "Hidden" from us

We Don't Tell Them What to Learn

Traditional ML:
"Here are features: P/E, momentum, volume"
We engineer the features.

Deep Learning Philosophy:
"Here is raw data. Find useful patterns."
Network discovers features.

Trade-off:
More automatic, but less interpretable.

Hidden layers discover features automatically

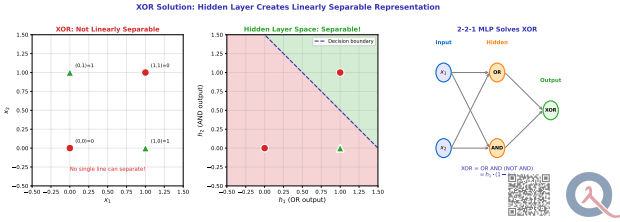
How MLPs Solve XOR

The Two-Hidden-Neuron Solution

Hidden Neuron 1:
Learns: "Is it in the upper-right region?"
$h_1 = \sigma(w_{11}x_1 + w_{12}x_2 + b_1)$

Hidden Neuron 2:
Learns: "Is it in the lower-left region?"
$h_2 = \sigma(w_{21}x_1 + w_{22}x_2 + b_2)$

Output Neuron:
Combines: "If h_1 XOR h_2, output 1"
Each hidden neuron draws *one* line. Together, they create a non-linear boundary.



xor_solution_ml

Multiple decision boundaries working together

Discussion Question

"If hidden layers find features automatically, why do we still need feature engineering in finance?"

Consider:

Arguments for Feature Engineering:

- Domain knowledge helps
- Less data needed
- More interpretable
- Faster training

Reality: In finance, hybrid approaches often work best.

Arguments Against:

- Human biases
- Miss non-obvious patterns
- Deep learning works on raw data
- ImageNet revolution

Think-Pair-Share: 3 minutes

Universal Approximation: The Big Promise

A Remarkable Theorem

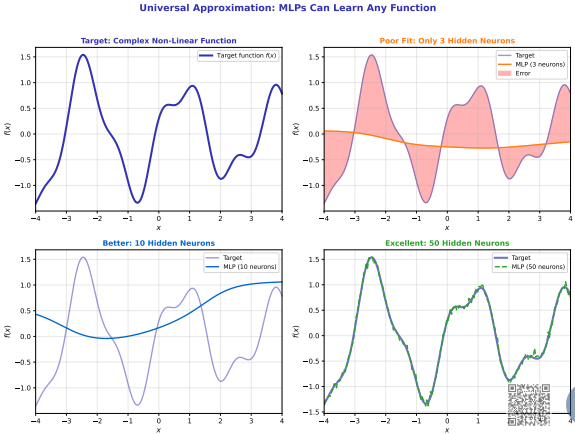
With just *one* hidden layer and enough neurons, an MLP can approximate **any** continuous function to arbitrary accuracy.

Implications:

- MLPs are universal function approximators
- No pattern is too complex (in theory)
- The architecture is not the bottleneck

Caveats:

- "Enough neurons" may be exponential
- Finding the right weights is hard
- Theory vs practice gap



universal_approximation_dem

Universal Approximation Theorem (Cybenko, 1989): A feedforward network with a single hidden layer containing a finite number of neurons can approximate any continuous function on compact subsets of $\mathbb{R}^n$

MLPs can learn ANY pattern (in theory)

Now Let's Formalize

What You Already Know
From the intuition section:

- Layers process sequentially
- Each layer transforms its input
- Hidden layers find patterns
- Output layer makes predictions

What's Next

- Matrix notation for efficiency
- Precise forward pass equations
- Parameter counting
- Worked numerical examples

Why Matrix Notation?

Without Matrices:
Write $n \times m$ separate equations for each weight.

With Matrices:

$$\mathbf{h} = f(\mathbf{W}\mathbf{x} + \mathbf{b})$$

One equation captures everything.

Benefits:

- Compact notation
- Efficient computation (GPUs)
- Easier to implement
- Clearer understanding

You understand the intuition. Let's write it precisely.

Matrix Notation: Why Matrices?

Single Neuron (Scalar)

$$h = f(w_1x_1 + w_2x_2 + w_3x_3 + b)$$

As Dot Product:

$$h = f(\mathbf{w}^T\mathbf{x} + b)$$

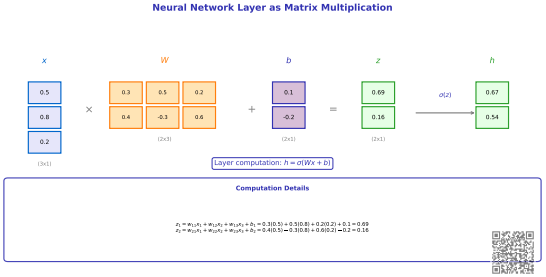
where $\mathbf{w}, \mathbf{x} \in \mathbb{R}^3$

Multiple Neurons (Matrix):

$$\mathbf{h} = f(\mathbf{W}\mathbf{x} + \mathbf{b})$$

where $\mathbf{W} \in \mathbb{R}^{m \times n}$

Each *row* of $\mathbf{W}$ is the weights for one hidden neuron.



matrix_multiplication_visua

Matrices make neural network math elegant

The Weight Matrix

Weight Matrix $\mathbf{W}^{(l)}$
For layer l:

$$\mathbf{W}^{(l)} \in \mathbb{R}^{n_l \times n_{l-1}}$$

where:

- n_l = neurons in layer l
- n_{l-1} = neurons in layer $l-1$

Entry $W_{ij}^{(l)}$:
Weight from neuron j in layer $l-1$ to neuron i in layer l.

Bias Vector $\mathbf{b}^{(l)}$

$$\mathbf{b}^{(l)} \in \mathbb{R}^{n_l}$$

One bias per neuron in layer l.

Example: 4-3 Layer
Input: 4 neurons, Hidden: 3 neurons

$$\mathbf{W}^{(1)} = \begin{pmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \\ w_{31} & w_{32} & w_{33} & w_{34} \end{pmatrix}$$

Size: 3×4 (12 weights)
$\mathbf{b}^{(1)} \in \mathbb{R}^3$ (3 biases)

Each layer has its own weight matrix

Forward Pass: Layer by Layer

One Layer Computation

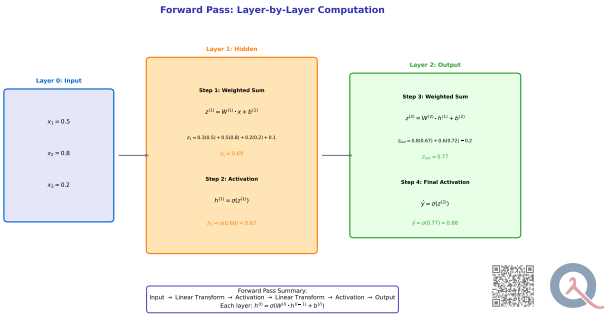
$$\mathbf{z}^{(l)} = \mathbf{W}^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}$$
$$\mathbf{a}^{(l)} = f(\mathbf{z}^{(l)})$$

where:

- $\mathbf{z}^{(l)}$: pre-activation (weighted sum)
- $\mathbf{a}^{(l)}$: activation (after f)
- $\mathbf{a}^{(0)} = \mathbf{x}$: input

The Steps:

1. Matrix multiply: $\mathbf{W}^{(l)}\mathbf{a}^{(l-1)}$
2. Add bias: $+\mathbf{b}^{(l)}$
3. Apply activation: $f(\cdot)$



layer_by_layer_computatio

Computing outputs one layer at a time

The Complete Forward Pass

For an L-Layer Network

Input:

$$\mathbf{a}^{(0)} = \mathbf{x}$$

Hidden Layers ($l = 1, \ldots, L-1$):

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

$$\mathbf{a}^{(l)} = f(\mathbf{z}^{(l)})$$

Output Layer:

$$\mathbf{z}^{(L)} = \mathbf{W}^{(L)}\mathbf{a}^{(L-1)} + \mathbf{b}^{(L)}$$

$$\hat{\mathbf{y}} = g(\mathbf{z}^{(L)})$$

where g may differ from f.

Example: 2-Layer Network

Layer 1 (hidden):

$$\mathbf{z}^{(1)} = \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}$$

$$\mathbf{a}^{(1)} = \text{ReLU}(\mathbf{z}^{(1)})$$

Layer 2 (output):

$$\mathbf{z}^{(2)} = \mathbf{W}^{(2)}\mathbf{a}^{(1)} + \mathbf{b}^{(2)}$$

$$\hat{y} = \sigma(\mathbf{z}^{(2)})$$

Compact Form:

$$\hat{y} = \sigma(\mathbf{W}^{(2)}\text{ReLU}(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}) + \mathbf{b}^{(2)})$$

Chaining layer computations together

Dimensions Matter

Dimension Checking
For $\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b}$:

$\mathbf{W}$:	$(n_{\text{out}} \times n_{\text{in}})$
$\mathbf{x}$:	$(n_{\text{in}} \times 1)$
$\mathbf{W}\mathbf{x}$:	$(n_{\text{out}} \times 1)$
$\mathbf{b}$:	$(n_{\text{out}} \times 1)$
$\mathbf{z}$:	$(n_{\text{out}} \times 1)$

Rule:
Inner dimensions must match.
$(m \times \mathbf{n}) \times (\mathbf{n} \times p) = (m \times p)$

Example: 4-3-1 Network

Layer 1:

- $\mathbf{W}^{(1)}$: 3×4
- $\mathbf{x}$: 4×1
- $\mathbf{z}^{(1)}$: 3×1

Layer 2:

- $\mathbf{W}^{(2)}$: 1×3
- $\mathbf{a}^{(1)}$: 3×1
- $\mathbf{z}^{(2)}$: 1×1 (scalar)

Common Error: Transposed matrices. Always check dimensions!

Matrix dimensions must be compatible

Worked Example: 2-3-1 Network

Network Setup

Input: $\mathbf{x} = \begin{pmatrix} 0.5 \\ 0.8 \end{pmatrix}$

Layer 1 weights:

$$\mathbf{W}^{(1)} = \begin{pmatrix} 0.2 & 0.4 \\ 0.3 & 0.1 \\ 0.5 & 0.2 \end{pmatrix}$$

$$\mathbf{b}^{(1)} = \begin{pmatrix} 0.1 \\ -0.1 \\ 0.0 \end{pmatrix}$$

Layer 2 weights:

$$\mathbf{W}^{(2)} = \begin{pmatrix} 0.6 & 0.3 & 0.4 \end{pmatrix}$$

$$b^{(2)} = -0.2$$

Forward Pass

Layer 1:

$$\mathbf{z}^{(1)} = \begin{pmatrix} 0.2(0.5) + 0.4(0.8) + 0.1 \\ 0.3(0.5) + 0.1(0.8) - 0.1 \\ 0.5(0.5) + 0.2(0.8) + 0.0 \end{pmatrix} = \begin{pmatrix} 0.52 \\ 0.13 \\ 0.41 \end{pmatrix}$$

$$\mathbf{a}^{(1)} = \text{ReLU}(\mathbf{z}^{(1)}) = \begin{pmatrix} 0.52 \\ 0.13 \\ 0.41 \end{pmatrix}$$

Layer 2:

$$z^{(2)} = 0.6(0.52) + 0.3(0.13) + 0.4(0.41) - 0.2 = 0.315$$

$$\hat{y} = \sigma(0.315) = 0.578$$

Output: 57.8% probability of class 1

Following the numbers through the network

Counting Parameters

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

Weights: $n_l \times n_{l-1}$
Biases: n_l
Total: $n_l \times n_{l-1} + n_l = n_l(n_{l-1} + 1)$

Network Total:

$$\text{Params} = \sum_{l=1}^{L} n_l(n_{l-1} + 1)$$

Example: 4-10-5-1 Network

Layer 1 ($4 \to 10$):

$$10 \times 4 + 10 = 50$$

Layer 2 ($10 \to 5$):

$$5 \times 10 + 5 = 55$$

Layer 3 ($5 \to 1$):

$$1 \times 5 + 1 = 6$$

Total: 111 parameters

For 100 training samples: < 2 samples per parameter.
Risk of overfitting!

How many weights does your network have?

Discussion Question

"A 4-10-5-1 network has how many parameters? Calculate and discuss: is this a lot or a little for stock prediction?"

Answer: 111 parameters

Consider:

Stock Data Context:

- Daily data: ∼252 days/year
- 10 years = 2,520 samples
- 111 params: 23 samples/param
- Seems okay...

But Also Consider:

- Financial regimes change
- Not all data equally relevant
- Need train/val/test split
- Model complexity vs data size

Exercise: 3 minutes

Finance Example: Multi-Factor Stock Prediction

A Realistic Setup

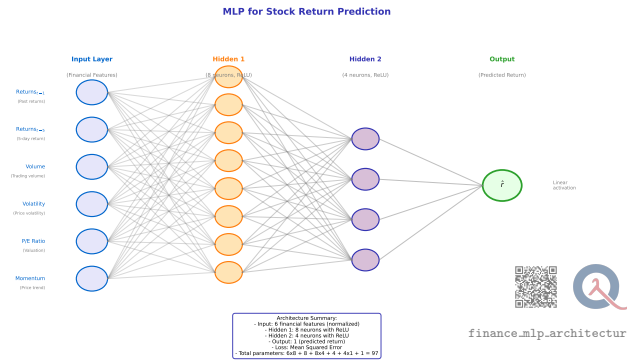
Input Features (10):

- P/E, P/B, EV/EBITDA (value)
- 1m, 3m, 6m returns (momentum)
- 20d volatility (risk)
- Volume ratio (liquidity)
- Sector one-hot (2 features)

Architecture:

- Hidden 1: 20 neurons (ReLU)
- Hidden 2: 10 neurons (ReLU)
- Output: 1 neuron (sigmoid)

Total: 441 parameters



Multiple factors combined through hidden layers

Why Non-Linearity?

The Core Problem
Without activation functions:

$$\mathbf{a}^{(1)} = \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}$$
$$\hat{\mathbf{y}} = \mathbf{W}^{(2)}\mathbf{a}^{(1)} + \mathbf{b}^{(2)}$$

Substituting:

$$\hat{\mathbf{y}} = \mathbf{W}^{(2)}(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}) + \mathbf{b}^{(2)}$$
$$= (\mathbf{W}^{(2)}\mathbf{W}^{(1)})\mathbf{x} + (\mathbf{W}^{(2)}\mathbf{b}^{(1)} + \mathbf{b}^{(2)})$$
$$= \mathbf{W}'\mathbf{x} + \mathbf{b}'$$

Result: A single linear transformation!

The Solution
Non-linear activation functions:

$$\mathbf{a}^{(l)} = f(\mathbf{z}^{(l)})$$

where f is non-linear.

Why This Works:

- Non-linearity breaks the collapse
- Composition of non-linear functions
- Can approximate any function

Key Insight:
Non-linearity is what makes deep networks "deep" in a meaningful sense.

Non-linearity is essential for learning complex patterns

Linear Networks Collapse

Mathematical Proof
For any number of linear layers:

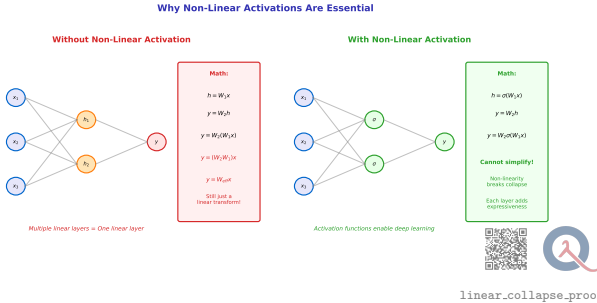
$$\mathbf{y} = \mathbf{W}^{(L)}\mathbf{W}^{(L-1)} \dots \mathbf{W}^{(1)}\mathbf{x}$$

Since matrix multiplication is associative:

$$= (\mathbf{W}^{(L)}\mathbf{W}^{(L-1)} \dots \mathbf{W}^{(1)})\mathbf{x}$$

$$= \mathbf{W}^{\text{eff}}\mathbf{x}$$

Conclusion:
100 linear layers = 1 linear layer.
No benefit from depth without non-linearity.



Stacked linear layers = single linear layer

The Sigmoid Function

Definition

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

Properties:

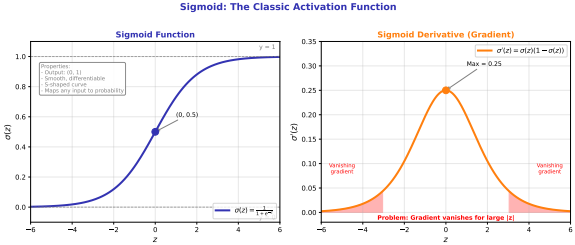
- Range: $(0, 1)$
- Smooth and differentiable
- $\sigma(0) = 0.5$
- Symmetric: $\sigma(-z) = 1 - \sigma(z)$

Derivative:

$$\sigma'(z) = \sigma(z)(1 - \sigma(z))$$

Use Cases:

- Binary classification (output)
- Probability interpretation
- Historical (hidden layers)



sigmoid_functio

The classic activation: squashes to probability

Sigmoid: Properties and Problems

Advantages

+ Bounded output $(0, 1)$
+ Smooth gradient
+ Probability interpretation
+ Historically important

Disadvantages

- **Vanishing gradients**
  For $|z| > 4$: $\sigma'(z) \approx 0$
  Gradients become tiny
  Deep networks can't learn
- Not zero-centered
  All positive outputs
  Zig-zag weight updates
- Computationally expensive
  Requires exp function

The Vanishing Gradient Problem

When z is very positive or negative:

z	$\sigma'(z)$
0	0.25
2	0.10
4	0.018
6	0.0025

Gradients shrink exponentially through layers!

Result: Early layers learn very slowly in deep networks. This limited deep learning until ReLU.

Smooth and bounded, but gradients can vanish

The Tanh Function

Definition

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} = 2\sigma(2z) - 1$$

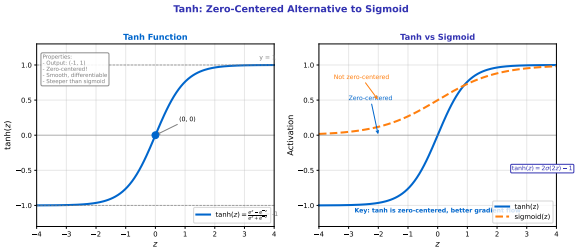
Properties:

- Range: $(-1, 1)$
- Zero-centered
- $\tanh(0) = 0$
- Odd function: $\tanh(-z) = -\tanh(z)$

Derivative:

$$\tanh'(z) = 1 - \tanh^2(z)$$

Advantage over Sigmoid:
Zero-centered outputs lead to more stable gradient updates.



tanh_functio

Zero-centered: range (-1, 1)

ReLU: Rectified Linear Unit

Definition

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

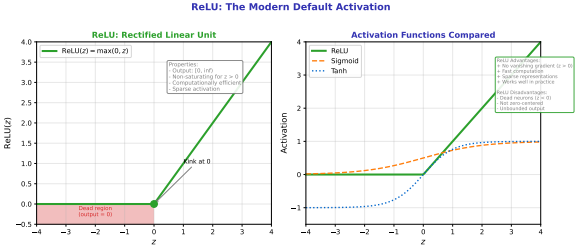
Properties:

- Range: $[0, \infty)$
- Not bounded above
- Not differentiable at $z = 0$
- Piecewise linear

Derivative:

$$\text{ReLU}'(z) = \begin{cases} 1 & z > 0 \\ 0 & z \leq 0 \end{cases}$$

The Modern Default for hidden layers.



relu_functio

Simple but powerful: the modern default

Why ReLU Works So Well

Advantages

+ **No vanishing gradient**
  Gradient is 1 for $z > 0$
  Signal propagates through layers
+ **Computationally cheap**
  Just comparison and assignment
  No exponentials
  6x faster than sigmoid
+ **Sparse activation**
  Many neurons output 0
  Efficient representation
+ **Biological plausibility**
  Neurons can be "off"

Disadvantages

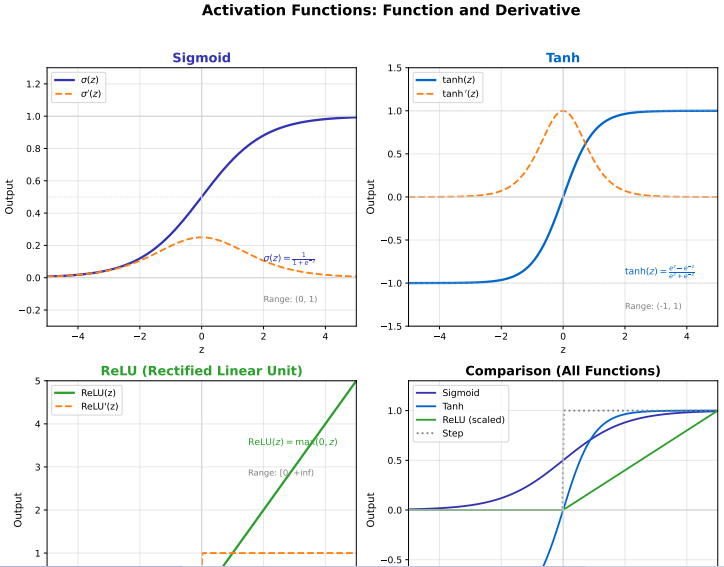
- **"Dying ReLU" problem**
  If $z < 0$ always: gradient $= 0$
  Neuron never updates
  Can "die" permanently
- Not zero-centered
- Unbounded (can explode)

Variants:

- Leaky ReLU: $\max(0.01z, z)$
- ELU: z if $z > 0$, $\alpha(e^z - 1)$ otherwise
- GELU: used in transformers

Cheap to compute, gradients don't vanish (for positive inputs)

Activation Functions: Comparison



Discussion Question

"Which activation function would you use for: (a) predicting stock returns, (b) buy/sell classification? Why?"

Consider:

(a) Stock Returns (Regression)

- Output: continuous value
- Can be positive or negative
- Hidden: ReLU or tanh
- Output: **Linear (none)**
- Returns are unbounded

(b) Buy/Sell (Classification)

- Output: probability $\in (0, 1)$
- Two mutually exclusive classes
- Hidden: ReLU
- Output: **Sigmoid**
- Or softmax for multi-class

Think-Pair-Share: 3 minutes

Choosing the Right Activation

Hidden Layer Guidelines

Default: ReLU

- Works well in most cases
- Fast and stable

If dying ReLU: Leaky ReLU

- Small negative slope
- Prevents dead neurons

For RNNs: Tanh

- Bounded outputs help stability
- Zero-centered

Output Layer Guidelines

Task	Activation
Binary class	Sigmoid
Multi-class	Softmax
Regression	Linear
Bounded regression	Sigmoid/tanh
Positive only	ReLU

Finance Examples:

- Return prediction: Linear
- Direction prediction: Sigmoid
- Sector classification: Softmax
- Volatility: ReLU or Softplus

Output layer choice depends on your problem type

The Fundamental Question

How Powerful Are Neural Networks?
We've seen that MLPs can:

- Solve XOR (non-linear patterns)
- Combine features hierarchically
- Learn from data

But a Deeper Question:
Are there functions that MLPs fundamentally *cannot* represent?

Or can they approximate *anything*?

Why This Matters

If MLPs are limited:

- Need to check if problem is solvable
- Architecture constraints matter
- Some patterns impossible

If MLPs are universal:

- Architecture is not the bottleneck
- Challenges are elsewhere (data, training)
- Theoretical guarantee of capability

Spoiler: MLPs are universal approximators!

Just how powerful are neural networks?

Universal Approximation Theorem

The Theorem (Informal)
A feedforward network with:

- One hidden layer
- Sufficient hidden neurons
- Non-linear activation (e.g., sigmoid)

can approximate any continuous function on a compact domain to arbitrary accuracy.

Key Contributors:

- Cybenko (1989): sigmoid
- Hornik (1991): general activations
- Further extensions since

Formal Statement
Let $f : [0,1]^n \to \mathbb{R}$ be continuous.
For any $\epsilon > 0$, there exists an MLP $\hat{f}$ with:

$$|\hat{f}(\mathbf{x}) - f(\mathbf{x})| < \epsilon$$

for all $\mathbf{x} \in [0,1]^n$.

In Plain English:
No matter how complex the pattern, an MLP with enough hidden neurons can match it as closely as you want.

With enough hidden neurons, you can approximate any continuous function

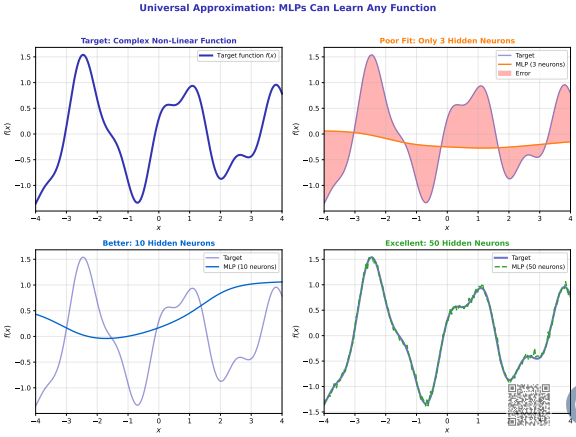
What Universal Approximation Means

The Good News

- No function is “too complex”
- MLPs are theoretically complete
- Architecture is not the limit
- One hidden layer is enough (in theory)

Visual Intuition:
Each hidden neuron contributes a “bump” or “step.”
With enough bumps, you can approximate any shape.

Think of it like approximating a curve with many small line segments.



universal_approximation_dem

Universal Approximation Theorem (Cybenko, 1989): A feedforward network with a single hidden layer containing a finite number of neurons can approximate any continuous function on compact subsets of $\mathbb{R}^n$

More neurons = better approximation

What It Doesn't Mean

Common Misconceptions

"Any network can learn anything"

No. Need enough neurons

- May need exponentially many

"Training will find the solution"

No. Theorem is about existence

- Says nothing about finding weights
- Optimization may fail

"One layer is always enough"

cally no.

- Deep networks often more efficient
- Fewer parameters for same accuracy

The Gap: Existence vs Construction

The theorem says:
"A good approximation exists."
It does NOT say:

- How many neurons you need
- How to find the right weights
- How much data is required
- How long training takes
- Whether it will generalize

Analogy:
"There exists a needle in this haystack" doesn't help you find it.

Existence of a solution does not mean we can find it

Theory vs Practice

Theoretical Guarantees
Universal approximation says:

- Given infinite neurons: perfect fit
- Given infinite data: find the function
- Given infinite compute: optimize

Practical Reality
We have:

- Finite neurons: limited capacity
- Finite data: must generalize
- Finite compute: approximate solutions

What Matters More in Practice

1. **Data quality and quantity**
   - More important than architecture
2. **Regularization**
   - Prevent overfitting
3. **Optimization**
   - Finding good weights
4. **Generalization**
   - Performance on new data

Module 3 will address these practical challenges.

Universal approximation is necessary but not sufficient

Implications for Finance

The Optimistic View
If markets have patterns, MLPs can learn them:

- Non-linear relationships? Possible.
- Complex interactions? Possible.
- Hidden factors? Possible.

Theoretical Capability:
"An MLP could, in principle, capture any market pattern."

The Realistic View

Challenges Remain:

- Signal-to-noise ratio is low
- Markets are non-stationary
- Past patterns may not repeat
- Data is limited (especially for crashes)
- Overfitting is easy

The EMH Counterargument:
If markets are efficient, there's nothing systematic to learn.

Module 4 will explore this tension.

In theory, yes. In practice, many challenges remain.

Why Loss Functions?

Learning Requires an Objective
To train a neural network, we need:

1. A way to measure errors
2. A number that decreases as we improve
3. A signal for weight updates

The Loss Function:
$\mathcal{L}(\hat{\mathbf{y}}, \mathbf{y})$ measures how wrong our predictions are.

Goal of Training:
Find weights that minimize $\mathcal{L}$.

Finance Analogy

Profit & Loss (P&L):

- Measures trading performance
- Negative P&L = bad trades
- Optimize to maximize P&L

Loss Function:

- Measures prediction errors
- High loss = bad predictions
- Optimize to minimize loss

Note: "Loss" is the opposite of "profit" – we minimize loss!

To learn, we must measure mistakes

Mean Squared Error (MSE)

Definition

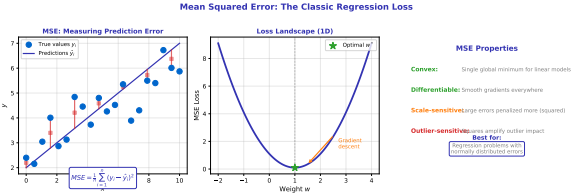
$$\mathcal{L}_{\text{MSE}} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Properties:

- Always non-negative
- Zero only if perfect predictions
- Penalizes large errors heavily
- Differentiable everywhere

Use Case:

- Regression problems
- Predicting continuous values
- Stock returns, prices, etc.



mse_visualizatio

The standard loss for predicting continuous values

Cross-Entropy Loss

Binary Cross-Entropy

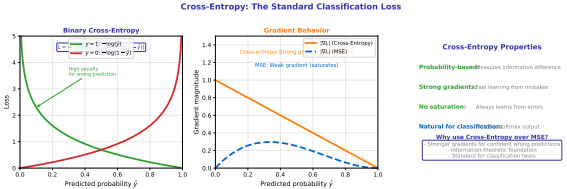
$$\mathcal{L}_{\text{BCE}} = -\frac{1}{n}\sum_{i=1}^{n}[y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)]$$

Properties:

- For probability outputs
- Heavily penalizes confident wrong answers
- Connected to information theory

Use Case:

- Classification problems
- Buy/sell decisions
- Any yes/no prediction



cross_entropy_visualizatio

The standard loss for classification

The Loss Landscape

Loss as a Function of Weights

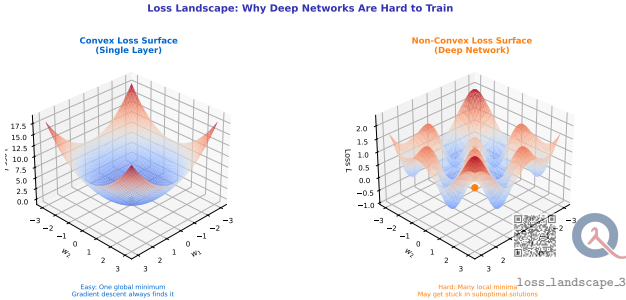
$$\mathcal{L}(\mathbf{W}, \mathbf{b})$$

For every choice of weights, there's a loss value.

The Landscape:

- High regions: bad weights
- Low regions: good weights
- Global minimum: best weights
- Local minima: traps

Training =
Finding the lowest point in this landscape.



Training = finding the lowest point in this landscape

Finance: Choosing Your Loss

Task-Specific Loss Functions

Task	Loss
Return prediction	MSE
Direction prediction	Cross-entropy
Volatility forecast	MSE
Multi-class sector	Categorical CE

Beyond Standard Losses:

- Sharpe ratio optimization
- Asymmetric losses (penalize losses more than gains)
- Custom finance metrics

Important Consideration

MSE vs Business Metric:
A model with low MSE may still lose money!

Example:

- Predict returns with 5% MSE
- But wrong on big moves
- Transaction costs eat profits
- Risk-adjusted return is poor

Lesson:
Statistical accuracy $\neq$ Trading profitability
Module 4 explores this gap.

Different problems, different loss functions

Module 2: Key Takeaways

What We Learned

1. **Historical Context**
   - AI Winter (1969-1982)
   - Backprop renaissance (1986)
   - Right idea + right time
2. **MLP Architecture**
   - Hidden layers find patterns
   - Matrix notation for computation
   - Parameter counting
3. **Activation Functions**
   - Non-linearity is essential
   - ReLU for hidden, task-specific for output
4. **Universal Approximation**
   - MLPs can learn any function
   - But existence $\neq$ construction
5. **Loss Functions**
   - MSE for regression
   - Cross-entropy for classification
   - Loss landscape visualization

The Big Picture:
We now have powerful architectures. But how do they *learn*?

From single perceptron to universal function approximator

Preview: Module 3

"We have the architecture. But how does it LEARN?"

The Missing Piece

We know:

- How to compute forward pass
- What loss functions measure
- That good weights exist

We don't know:

- How to find good weights
- How errors update weights
- How to avoid overfitting

Coming in Module 3:

- Gradient descent (intuition)
- Backpropagation (the magic)
- Training dynamics
- Overfitting and regularization
- Practical training tips

The Key: Backpropagation – the algorithm that made deep learning possible.

Mathematical details: See Appendix B (Backpropagation Derivation)

Next: The magic of backpropagation

Module 1: The Birth of Neural Computing

From Biological Inspiration to the Perceptron (1943-1969)

Neural Networks for Finance

BSc Lecture Series

November 28, 2025

The Central Question

"We have the architecture. How does it LEARN?"

What We Know:

- MLP architecture (Module 2)
- Forward pass computation
- Loss functions measure error
- Good weights exist (universal approximation)

What We Don't Know:

- How to find good weights
- How errors guide updates
- Why training sometimes fails
- How to avoid overfitting

This module bridges the gap from architecture to learning.

The fundamental challenge of neural network training

Finance Parallel: The Trading Desk

How Traders Improve

A trader's learning process:

1. Make a trade (forward pass)
2. Wait for P&L (loss function)
3. Analyze what went wrong (gradient)
4. Adjust strategy (weight update)
5. Repeat thousands of times (epochs)

Key Insight:

Mistakes are information. Each error tells you how to adjust.

Neural Network Training

Trading	Neural Net
Trade execution	Forward pass
P&L calculation	Loss function
Post-trade analysis	Backpropagation
Strategy adjustment	Weight update
Experience	Training epochs

Both learn by **iteratively correcting mistakes**.

How does a trader improve? By analyzing what went wrong.

Module 3 Roadmap

Today's Journey

1. **Loss Functions (Review)**
   - Measuring prediction error
   - MSE intuition
2. **Gradient Descent**
   - Finding the minimum
   - Learning rate tuning
3. **Backpropagation**
   - Credit assignment
   - Chain rule in action
4. **Training Dynamics**
   - Batch vs. stochastic
   - Epochs and convergence
5. **Overfitting**
   - The enemy of generalization
   - The backtest trap

Learning Objectives:

- Understand gradient descent intuitively
- Grasp backpropagation as "blame assignment"
- Recognize and prevent overfitting

From measuring error to updating weights

The Learning Problem

The Challenge
Given:

- Training data: $\{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^{m}$
- Network architecture
- Loss function $\mathcal{L}$

Find:

- Weights $\mathbf{W}$ and biases $\mathbf{b}$
- That minimize $\mathcal{L}$
- And generalize to new data

Scale of the Problem:
A 4-10-5-1 network: 111 parameters
A ResNet-50: 25 million parameters

Why Is This Hard?

Dimensionality:

- Millions of weights to tune
- Exponentially many combinations
- Can't try them all

Non-Convexity:

- Many local minima
- Saddle points
- Flat regions

The Solution:
Gradient-based optimization
"Move downhill in weight space"

Thousands of weights to tune - how do we find the right values?

1989: LeNet and Practical Success

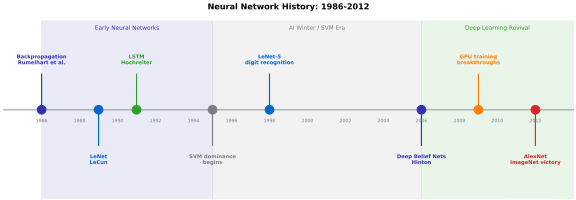
Yann LeCun at Bell Labs

First commercially deployed neural network:

- Handwritten digit recognition
- Used by US Postal Service
- Read millions of checks
- Proved neural nets could work

Key Innovations:

- Convolutional architecture
- Shared weights
- Backprop through convolutions



timeline_1986_201

Yann LeCun: First commercially deployed neural network

1991: The Vanishing Gradient Problem

The Discovery
Sepp Hochreiter (1991) identified why deep networks fail:

The Problem:

- Gradients multiply through layers
- Sigmoid derivative: max 0.25
- Through 10 layers: $0.25^{10} \approx 10^{-6}$
- Early layers learn nothing

Symptoms:

- Later layers learn quickly
- Early layers stuck at random
- Network never converges

Why Sigmoid Causes Problems

For sigmoid: $\sigma'(z) = \sigma(z)(1 - \sigma(z))$
Maximum value: $\sigma'(0) = 0.25$

Layers	Max Gradient
1	0.25
5	10^{-3}
10	10^{-6}
20	10^{-12}

Implication: Deep networks seemed impossible until ReLU (2010).

Deep networks couldn't learn - gradients disappeared

1997: LSTM Networks

Long Short-Term Memory
Hochreiter & Schmidhuber solution:

- Designed for sequences
- Explicit "memory" cells
- Gating mechanisms
- Gradients can flow unchanged

Key Innovation:
The "constant error carousel" – a path where gradients don't decay.

Applications:

- Speech recognition
- Machine translation
- Time series prediction

Finance Relevance

LSTMs became popular for:

- Stock price prediction
- Volatility forecasting
- Sentiment analysis
- Algorithmic trading

Why LSTM for Finance?

- Financial data is sequential
- Long-term dependencies matter
- Regime changes persist

Note: Now largely replaced by Transformers (2017).

Hochreiter and Schmidhuber: Long Short-Term Memory

2012: The ImageNet Moment

AlexNet Wins ImageNet

Alex Krizhevsky, Ilya Sutskever, Geoffrey Hinton:

- 15.3% error rate
- Second place: 26.2%
- **40% relative improvement**
- Used GPUs for training

What Made It Work:

1. ReLU activation (not sigmoid)
2. Dropout regularization
3. GPU training (60x faster)
4. Large dataset (1.2M images)
5. Data augmentation

Why This Was Different

Previous Attempts:

- Shallow networks
- Hand-crafted features
- Small datasets
- CPU training

AlexNet:

- 8 layers deep
- Learned features
- Massive data
- GPU parallelism

The Result: Deep learning became the dominant paradigm. Every major AI company pivoted.

AlexNet: Deep learning proves its superiority

What Changed Between 1990 and 2012?

The Ingredients for Success

1. **Big Data**
   - ImageNet: 1.2M labeled images
   - Internet made data collection possible
   - 1990: thousands of samples
2. **Compute Power**
   - GPUs: 100x speedup
   - Moore's law compounding
   - Training in days, not years
3. **Algorithmic Improvements**
   - ReLU: no vanishing gradients
   - Dropout: better generalization
   - Batch normalization (2015)
4. **Open Research Culture**
   - arXiv preprints
   - Open-source frameworks
   - Reproducibility

Key Insight: The core ideas from 1986 worked – they just needed scale and engineering.

Big data + GPUs + ReLU + dropout = breakthrough

What Does "Wrong" Mean?

Quantifying Prediction Error
We need a function that:

- Takes predictions and labels
- Returns a single number
- Higher = worse predictions
- Differentiable (for gradients)

The Loss Function:

$$\mathcal{L}(\hat{y}, y)$$

Properties We Want:

- $\mathcal{L} \geq 0$ (non-negative)
- $\mathcal{L} = 0$ iff perfect prediction
- Smooth (for optimization)

Different Tasks, Different Losses

Task	Loss
Regression	MSE
Binary classification	Cross-entropy
Multi-class	Categorical CE
Ranking	Hinge loss

Finance Examples:

- Return prediction: MSE
- Buy/sell: Binary CE
- Sector classification: Categorical CE

We need a way to measure how wrong our predictions are

Finance Analogy: Profit and Loss

P&L as a Loss Function
For traders:

- P&L = realized gain/loss
- Negative P&L = bad trades
- Goal: maximize P&L

Connection to ML Loss:

- ML loss = prediction error
- Higher loss = worse model
- Goal: minimize loss

Key Difference:
P&L is a *performance* metric.
ML loss is an *optimization* target.
They may not align perfectly!

When P&L $\neq$ Loss

A model might have:

- Low MSE (accurate predictions)
- But low P&L (wrong on big moves)

Or:

- High MSE (noisy predictions)
- But high P&L (right when it matters)

Implication:
Consider using custom loss functions that better align with trading goals.

Module 4 explores this tension.

P&L is the loss function of trading

Loss Function: Error Measurement

Total Loss Over Dataset
For m training examples:

$$\mathcal{L}(\mathbf{W}) = \frac{1}{m} \sum_{i=1}^{m} \ell(\hat{y}^{(i)}, y^{(i)})$$

where:

- ℓ: loss per example
- $\hat{y}^{(i)} = f(\mathbf{x}^{(i)}; \mathbf{W})$: prediction
- $y^{(i)}$: true label
- $\mathbf{W}$: all network weights

Goal:

$$\mathbf{W}^* = \arg\min_{\mathbf{W}} \mathcal{L}(\mathbf{W})$$

Why Average?

Sum vs Average:

- Sum: scales with dataset size
- Average: comparable across datasets
- Gradient magnitude consistent

The Optimization Landscape:
$\mathcal{L}(\mathbf{W})$ defines a surface over weight space.

- High regions: bad weights
- Low regions: good weights
- We seek the lowest point

The loss function quantifies prediction error

Mean Squared Error: Intuition

The Formula

$$\mathcal{L}_{MSE} = \frac{1}{m}\sum_{i=1}^{m}(y^{(i)} - \hat{y}^{(i)})^2$$

In Words:

1. Compute error: $y - \hat{y}$
2. Square it: $(y - \hat{y})^2$
3. Average over all samples

Why Squaring?

- Makes all errors positive
- Penalizes large errors heavily
- Mathematically convenient

Example

y	$\hat{y}$	$(y - \hat{y})^2$
5%	3%	4
-2%	1%	9
8%	7%	1
MSE		**4.67**

Units: (percentage points)^2

RMSE: $\sqrt{MSE} = 2.16\%$
"On average, we're off by about 2%"

"How far off were we, on average?"

MSE: Visual Interpretation

Squared Errors as Areas
Each error $(y - \hat{y})^2$ is the area of a square with side length $|y - \hat{y}|$.

MSE = Average Square Area

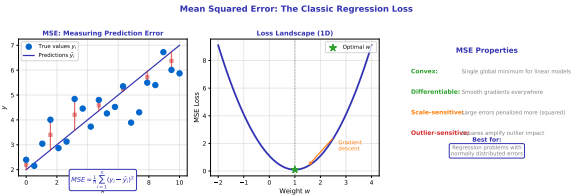
Why This Matters:

- Error of 4 is 16x worse than error of 1
- Large errors dominate
- Outliers have huge impact

Alternative: MAE
Mean Absolute Error:

$$\mathcal{L}_{MAE} = \frac{1}{m} \sum |y - \hat{y}|$$

More robust to outliers.



Squaring emphasizes large errors

Finance Application: Return Prediction

Worked Example
Predictions for 5 Stocks:

Stock	$\hat{y}$	y	Error^2
AAPL	+5%	+2%	9
MSFT	+3%	+4%	1
GOOG	-1%	+2%	9
AMZN	+4%	+4%	0
META	+2%	-3%	25
MSE			**8.8**

RMSE = 2.97%

Interpretation

"On average, our return predictions are off by about 3 percentage points."

Is This Good?
Depends on context:

- Market daily vol: $\sim$1%
- 3% RMSE = 3 std devs
- Not very predictive

Reality Check:
Even small predictability (RMSE slightly $<$ volatility) can be valuable in trading.

Worked example with stock returns

Discussion Question

"Why might we want to penalize large errors more than small ones in stock prediction?"

Consider:

Arguments For (Use MSE):

- Big errors are costlier
- Crashes matter more than small moves
- Position sizing affected
- Risk management

Arguments Against (Use MAE):

- Markets have fat tails
- Outliers can dominate MSE
- May optimize for rare events
- Robustness to noise

Reality: Many practitioners use MAE or Huber loss (combines both) for financial applications.

Think-Pair-Share: 3 minutes

The Loss Landscape

Loss as a Function of Weights

$$\mathcal{L}(\mathbf{W})$$

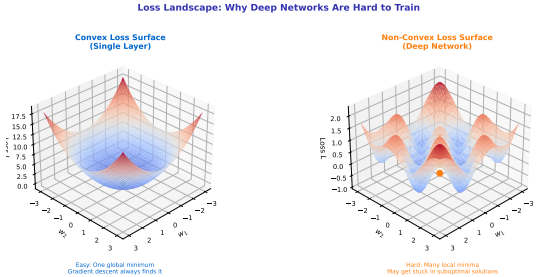
For every choice of weights, there's a loss value.

In 2D (two weights):
A surface we can visualize.

In High Dimensions:
A hypersurface we navigate blindly.

Features:

- Global minimum (best)
- Local minima (traps)
- Saddle points
- Flat regions (plateaus)



Finding the minimum of a high-dimensional function

The Optimization Problem

The Challenge
Find:

$$\mathbf{W}^* = \arg\min_{\mathbf{W}} \mathcal{L}(\mathbf{W})$$

Difficulties:

- Millions of dimensions
- Non-convex landscape
- No closed-form solution
- Can't try all possibilities

We Need:
An *iterative* algorithm that gradually improves weights.

Possible Approaches

Random Search:

- Try random weights
- Keep best so far
- Hopelessly slow

Grid Search:

- Try all combinations
- 10^{100} possibilities
- Impossible

Gradient-Based:

- Use local slope information
- Move toward improvement
- Tractable!

How do we find the weights that minimize loss?

The Blind Hiker Analogy

The Scenario
Imagine you're:

- Blindfolded
- On a mountainside
- Trying to reach the valley
- Can only feel the local slope

What Would You Do?

1. Feel the ground around you
2. Determine which way is downhill
3. Take a step in that direction
4. Repeat until you reach a valley

Neural Network Translation

Hiker	Network
Position	Weights $\mathbf{W}$
Altitude	Loss $\mathcal{L}$
Slope	Gradient $\nabla\mathcal{L}$
Step	Weight update
Valley	Minimum loss

Key Insight:
We don't need to see the whole landscape. Local slope is enough!

"You're blindfolded on a mountain. How do you find the valley?"

Answer: Feel the Slope

The Strategy

1. Compute the slope (gradient)
2. Move opposite to the slope
3. Repeat until convergence

Why Opposite?

- Gradient points uphill
- We want to go downhill
- Move in negative gradient direction

The Update Rule:

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \nabla_{\mathbf{W}} \mathcal{L}$$

Gradient Descent Algorithm

1. Initialize $\mathbf{W}$ randomly
2. **repeat**:
   a. Compute loss $\mathcal{L}(\mathbf{W})$
   b. Compute gradient $\nabla \mathcal{L}$
   c. Update: $\mathbf{W} \leftarrow \mathbf{W} - \eta \nabla \mathcal{L}$
3. **until** convergence

η = learning rate (step size)

Move in the direction that goes down

The Gradient: Direction of Steepest Ascent

What Is the Gradient?
The gradient $\nabla \mathcal{L}$ is a vector of partial derivatives:

$$\nabla_{\mathbf{w}} \mathcal{L} = \begin{pmatrix} \frac{\partial \mathcal{L}}{\partial w_1} \\ \frac{\partial \mathcal{L}}{\partial w_2} \\ \vdots \\ \frac{\partial \mathcal{L}}{\partial w_n} \end{pmatrix}$$

Each Component:
$\frac{\partial \mathcal{L}}{\partial w_i}$ = How much does loss change if we change w_i slightly?

Properties

Direction:

- Points toward steepest increase
- $-\nabla \mathcal{L}$ points toward steepest decrease

Magnitude:

- $\|\nabla \mathcal{L}\|$ = slope steepness
- Near minimum: gradient ≈ 0

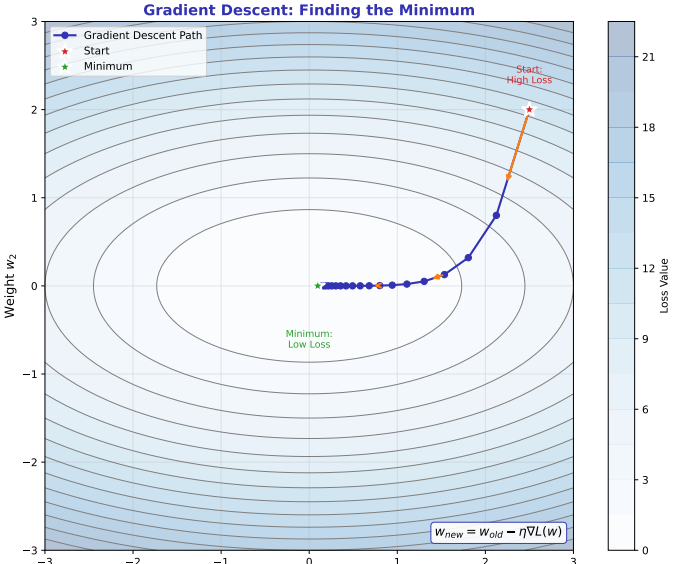
At a Minimum:

$$\nabla \mathcal{L} = \mathbf{0}$$

No direction goes further down.

The gradient tells us which way is "up"

Gradient Descent: Move Downhill



Finance Parallel: Portfolio Optimization

Portfolio Adjustment
Similar iterative process:

1. Evaluate current portfolio
2. Estimate sensitivities ("greeks")
3. Adjust positions to reduce risk
4. Repeat periodically

Delta Hedging:

- Measure option delta
- Adjust stock position
- Move toward neutral

Comparison

GD	Portfolio
Loss	Risk/Variance
Weights	Positions
Gradient	Sensitivities
Learning rate	Trading aggressiveness
Convergence	Optimal allocation

Key Difference:
Markets change continuously. Portfolios must adapt. Neural networks train once (mostly).

Similar to iterative portfolio rebalancing

The Learning Rate: Step Size

The Hyperparameter η

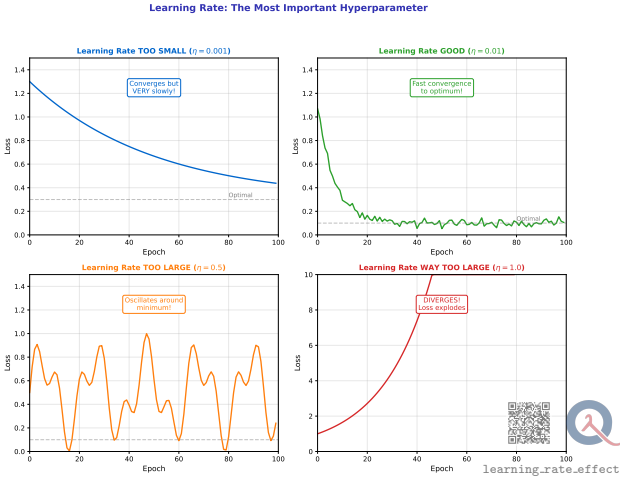
$$\mathbf{W} \leftarrow \mathbf{W} - \eta \nabla \mathcal{L}$$

η Controls:

- Size of each weight update
- Speed of convergence
- Stability of training

Typical Values:

- 10^{-4} to 10^{-1}
- Often starts at 0.01 or 0.001
- May decrease during training



Learning rate controls how far we move each step

Learning Rate Too High

The Problem
When η is too large:

- Steps overshoot the minimum
- May jump to worse regions
- Loss oscillates or explodes
- Training diverges

Symptoms:

- Loss goes up, not down
- Loss becomes NaN
- Weights grow very large
- Erratic training curves

Finance Analogy

Overtrading:

- Adjusting positions too aggressively
- Chasing every signal
- Transaction costs accumulate
- Portfolio becomes unstable

Solution:
Reduce learning rate until stable.

Rule of Thumb: If loss explodes, halve η.

Too big = overshoot the minimum

Learning Rate Too Low

The Problem
When η is too small:

- Steps are tiny
- Progress is slow
- May get stuck in flat regions
- Training takes forever

Symptoms:

- Loss decreases very slowly
- Many epochs with little improvement
- May stop before reaching minimum
- Wasted computation

Finance Analogy

Underreacting:

- Ignoring market signals
- Missing opportunities
- Portfolio drifts from target
- Slow adaptation to regime changes

Solution:
Increase learning rate or use adaptive methods.

Modern Practice: Adaptive optimizers (Adam, RMSprop) adjust η automatically.

Too small = converge too slowly

Discussion Question

"In trading, what's analogous to learning rate? What happens if you adjust positions too aggressively or too conservatively?"

Consider:

Position Sizing:

- How much to trade per signal
- Kelly criterion vs. fractional Kelly
- Risk management constraints

Rebalancing Frequency:

- How often to adjust
- Transaction cost vs. tracking error
- Market impact considerations

Key Insight: Both trading and ML require balancing responsiveness against stability.

Think-Pair-Share: 3 minutes

The Attribution Problem

The Challenge
We know:

- The output was wrong
- We need to update weights
- There are thousands of weights

The Question:

Which weights caused the error?

Credit Assignment:
Attributing output error to individual weights deep in the network.

Why Is This Hard?

Direct Attribution:

- Output layer weights: clear influence
- Hidden layer weights: indirect
- Early layers: very indirect

The Chain of Influence:
$w_1 \to h_1 \to h_2 \to \cdots \to \hat{y} \to \mathcal{L}$
Each weight affects the loss through many intermediate steps.

The output was wrong. Which weights caused it?

Finance Analogy: Post-Trade Analysis

Attribution in Trading

A portfolio lost money. Why?

1. Macro call wrong?
2. Sector allocation off?
3. Stock selection bad?
4. Timing poor?
5. Execution costly?

Performance Attribution:

- Decompose returns by factor
- Trace P&L to decisions
- Learn which calls were wrong

Neural Network Attribution

Trading	Neural Net
Macro view	Early layers
Sector allocation	Hidden layers
Stock picks	Later layers
Final trades	Output
P&L	Loss

Backpropagation is the neural network's performance attribution algorithm.

"Which decisions led to this P&L?"

Backpropagation: Blame Assignment

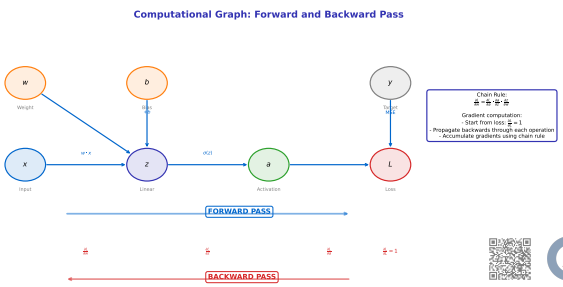
The Algorithm
Backpropagation computes $\frac{\partial \mathcal{L}}{\partial w}$ for every weight w in the network.

Key Idea:
Work backward from output to input, propagating error attribution.

Two Passes:

1. **Forward Pass:** Compute outputs
2. **Backward Pass:** Compute gradients

Efficiency:
Computes ALL gradients in time proportional to one forward pass.



backprop_computational_grap

Propagating error backward through the network

The Chain Rule: Intuition

The Core Mathematical Tool

If A affects B and B affects C:

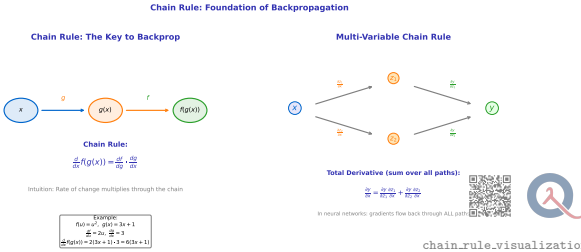
$$\frac{\partial C}{\partial A} = \frac{\partial C}{\partial B} \cdot \frac{\partial B}{\partial A}$$

Example:

Temperature → Ice cream sales → Profit

How does temperature affect profit?

$$\frac{\partial \text{Profit}}{\partial \text{Temp}} = \frac{\partial \text{Profit}}{\partial \text{Sales}} \cdot \frac{\partial \text{Sales}}{\partial \text{Temp}}$$



"If A affects B and B affects C, how does A affect C?"

Finance Chain Example

Chain of Effects

Fed Rate → Mortgages → Housing → Banks → Portfolio

How does Fed rate affect your portfolio?

$$\frac{\partial \text{Portfolio}}{\partial \text{Fed}} = \frac{\partial P}{\partial B} \cdot \frac{\partial B}{\partial H} \cdot \frac{\partial H}{\partial M} \cdot \frac{\partial M}{\partial F}$$

Each Link:

- Fed → Mortgages: rate sensitivity
- Mortgages → Housing: demand elasticity
- Housing → Banks: credit exposure
- Banks → Portfolio: position size

Neural Network Parallel

Finance	Neural Net
Fed rate	Input x
Mortgages	Hidden layer 1
Housing	Hidden layer 2
Banks	Hidden layer 3
Portfolio	Output

Backprop does this automatically:
Chains together all the local sensitivities to get the total effect of each input/weight on the loss.

Effects propagate through chains of influence

Backprop: Output Layer

At the Output
For output weight $w^{(L)}$:

$$\frac{\partial \mathcal{L}}{\partial w^{(L)}} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z^{(L)}} \cdot \frac{\partial z^{(L)}}{\partial w^{(L)}}$$

Each Term:

- $\frac{\partial \mathcal{L}}{\partial \hat{y}}$: How loss changes with output
- $\frac{\partial \hat{y}}{\partial z^{(L)}}$: Activation derivative
- $\frac{\partial z^{(L)}}{\partial w^{(L)}}$: Input from previous layer

For MSE + Sigmoid:

$$\frac{\partial \mathcal{L}}{\partial w^{(L)}} = (\hat{y} - y) \cdot \hat{y}(1 - \hat{y}) \cdot a^{(L-1)}$$

Output Error ($\delta^{(L)}$)

Define the "error signal":

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

For MSE loss + sigmoid:

$$\delta^{(L)} = (\hat{y} - y) \cdot \sigma'(z^{(L)})$$

Then:

$$\frac{\partial \mathcal{L}}{\partial w^{(L)}} = \delta^{(L)} \cdot a^{(L-1)}$$

This is just error $\times$ input!

At the output, error attribution is straightforward

Backprop: Hidden Layers

The Key Insight
Hidden layer error comes from downstream:

$$\delta^{(l)} = ((W^{(l+1)})^T \delta^{(l+1)}) \odot \sigma'(z^{(l)})$$

In Words:

1. Take error from next layer ($\delta^{(l+1)}$)
2. Multiply by weights connecting to next layer
3. Scale by local activation derivative

Error Flows Backward:
Output → Last hidden → ... → First hidden

Why This Works

Chain rule connects layers:

$$\frac{\partial \mathcal{L}}{\partial z^{(l)}} = \sum_j \frac{\partial \mathcal{L}}{\partial z_j^{(l+1)}} \cdot \frac{\partial z_j^{(l+1)}}{\partial z^{(l)}}$$

Gradient for Hidden Weight:

$$\frac{\partial \mathcal{L}}{\partial w^{(l)}} = \delta^{(l)} \cdot a^{(l-1)}$$

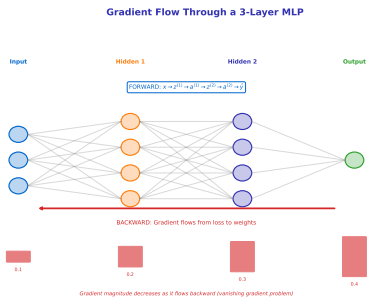
Same formula as output layer!

Hidden layer gradients require the chain rule

The Complete Training Loop

One Training Step

1. **Forward Pass**
   - Compute all activations
   - Get prediction $\hat{y}$
2. **Compute Loss**
   - $\mathcal{L} = \ell(\hat{y}, y)$
3. **Backward Pass**
   - Compute $\delta^{(L)}$ at output
   - Propagate backward to get all $\delta^{(l)}$
   - Compute all weight gradients
4. **Update Weights**
   - $\mathbf{W} \leftarrow \mathbf{W} - \eta \nabla \mathcal{L}$



gradient_flow_ml

Forward pass, compute loss, backward pass, update weights

Why "Backpropagation"?

The Name
"Back-propagation of errors"

Information Flow:
Forward:

- Data flows input $\rightarrow$ output
- Activations computed layer by layer

Backward:

- Errors flow output $\rightarrow$ input
- Gradients computed layer by layer

Symmetry:
Each layer: one forward operation, one backward operation.

Historical Note

The Algorithm:

- Werbos (1974): first derivation
- Rumelhart et al. (1986): popularized
- Now standard in all deep learning

Modern Perspective:
Backprop is just automatic differentiation applied to neural networks.

Frameworks (PyTorch, TensorFlow):
Compute gradients automatically – you just specify the forward pass!

Error information flows from output to input

Discussion Question

"Why do deeper networks make training harder? What happens to gradients as they flow backward through many layers?"

Consider:

Vanishing Gradients:

- Sigmoid: max derivative 0.25
- Through 10 layers: 0.25^{10}
- Early layers get tiny gradients
- Learn extremely slowly

Exploding Gradients:

- If derivatives > 1
- Gradients grow exponentially
- Weights become huge
- Training diverges

Solutions: ReLU, batch normalization, residual connections, careful initialization.

Think-Pair-Share: 3 minutes

Batch Gradient Descent

Definition
Use **all** training data to compute gradient:

$$\nabla \mathcal{L} = \frac{1}{m} \sum_{i=1}^{m} \nabla \ell(\hat{y}^{(i)}, y^{(i)})$$

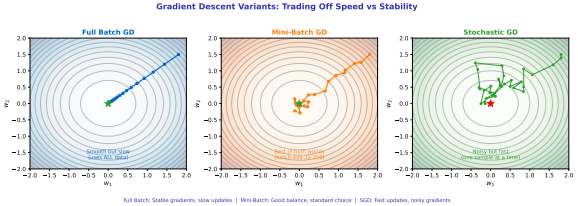
Then update weights once.

Advantages:

+ Stable gradient estimate
+ Deterministic updates
+ Guaranteed descent direction

Disadvantages:

- Slow for large datasets
- Must load all data in memory
- One update per full pass



batch_vs_stochasti

Compute gradient using the entire dataset

Stochastic Gradient Descent (SGD)

Definition
Update after **each** single example:

$$\nabla \mathcal{L} \approx \nabla \ell(\hat{y}^{(i)}, y^{(i)})$$

One sample = one update.

Advantages:

+ Very fast updates
+ Can handle huge datasets
+ Noise helps escape local minima
+ Online learning possible

Disadvantages:

- Noisy gradient estimate
- Erratic convergence
- May not settle at minimum

Why "Stochastic"?

Random sampling of training examples introduces randomness into gradient.

Expected Value:

$$\mathbb{E}[\nabla \ell^{(i)}] = \nabla \mathcal{L}$$

On average, SGD points in the right direction.

Variance:
Individual updates are noisy, but noise can help exploration.

Update after each single example

Mini-Batch: The Sweet Spot

Definition
Use small batches of B examples:

$$\nabla \mathcal{L} \approx \frac{1}{B} \sum_{i=1}^{B} \nabla \ell(\hat{y}^{(i)}, y^{(i)})$$

Typical B: 32, 64, 128, 256

Advantages:

+ Reduced variance vs SGD
+ GPU parallelization
+ Reasonable memory usage
+ Frequent updates

The Modern Default

Batch Size Trade-offs

Size	Noise	Speed
1 (SGD)	High	Fast updates
32-256	Medium	Best practice
Full batch	Low	Slow updates

Large Batch Issues:

- May converge to sharp minima
- Worse generalization
- Need learning rate scaling

Balance between efficiency and noise

Epochs: Full Passes Through Data

Definition

Epoch = one complete pass through all training data.

With Mini-Batches:

- 10,000 samples
- Batch size 100
- 100 updates per epoch

Typical Training:

- 10-1000 epochs
- Monitor loss curve
- Stop when converged

Training Timeline

Stage	Behavior
Early epochs	Loss drops quickly
Middle epochs	Progress slows
Late epochs	Diminishing returns

When to Stop?

- Loss stops improving
- Validation loss increases (overfitting!)
- Resource constraints

Training typically requires multiple epochs

Training Curves

What to Plot

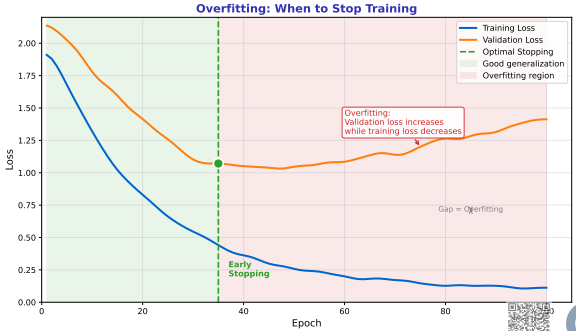
- Training loss vs. epoch
- Validation loss vs. epoch
- Learning rate schedule
- Gradient norms (debugging)

Healthy Training:

- Both losses decrease
- Validation tracks training
- Smooth convergence

Warning Signs:

- Training drops, validation rises
- Loss oscillates wildly
- Loss becomes NaN



overfitting_curve

Monitoring progress during training

Worked Example: One Training Step

Simple 2-2-1 Network
Given:

- Input: $\mathbf{x} = (0.5, 0.8)^T$
- Target: $y = 1$
- Current weights (simplified)

Forward Pass:

$$z^{(1)} = W^{(1)}\mathbf{x} + b^{(1)}$$
$$a^{(1)} = \sigma(z^{(1)})$$
$$z^{(2)} = W^{(2)}a^{(1)} + b^{(2)}$$
$$\hat{y} = \sigma(z^{(2)}) = 0.62$$

Loss and Backward
Loss:

$$\mathcal{L} = \frac{1}{2}(y - \hat{y})^2 = \frac{1}{2}(1 - 0.62)^2 = 0.072$$

Backward Pass:

$$\delta^{(2)} = (0.62 - 1) \cdot 0.62(1 - 0.62)$$
$$= -0.089$$

Weight Gradient:

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

Update:

$$W^{(2)} \leftarrow W^{(2)} - 0.1 \cdot \nabla W^{(2)}$$

Following the numbers through one training step

The Vanishing Gradient Problem

The Problem

Gradients shrink as they flow backward:

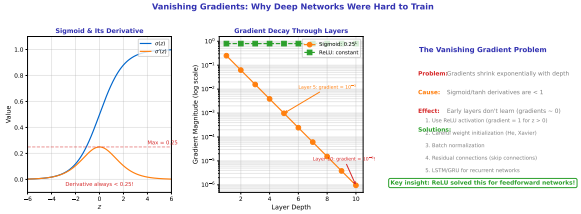
$$\delta^{(l)} \propto \prod_{k=l}^{L-1} \sigma'(z^{(k)})$$

For sigmoid: $\sigma'(z) \leq 0.25$

Through 10 layers: gradient $\times 10^{-6}$

Symptoms:

- Early layers don't learn
- Deep networks fail to train
- Loss plateaus quickly



vanishing_gradient_dem

Deep networks: gradients can become vanishingly small

Full Mathematical Derivation

This Module: Intuition
We covered:

- Why backprop works (chain rule)
- How errors flow backward
- Update rule intuition
- Training dynamics

What We Skipped:

- Full mathematical derivation
- Matrix calculus details
- Vectorized implementations
- Automatic differentiation theory

Appendix B Contains:

1. Chain rule setup
2. Output layer error derivation
3. Hidden layer recursion formula
4. Complete gradient equations
5. Weight and bias gradients
6. Algorithm pseudocode

For the mathematically curious:
The appendix provides the rigorous derivation with all matrix calculus steps.

See Appendix B for complete backpropagation derivation

What Is Overfitting?

Definition

Overfitting: When a model learns the training data too well, including its noise, and fails to generalize.

Analogy:

A student who memorizes exam answers but doesn't understand the material.

Symptoms:

- Training loss: very low
- Test loss: high
- Model is "too confident"

Why It Happens

Model Complexity:

- Too many parameters
- Can fit any training data perfectly
- Including noise

Limited Data:

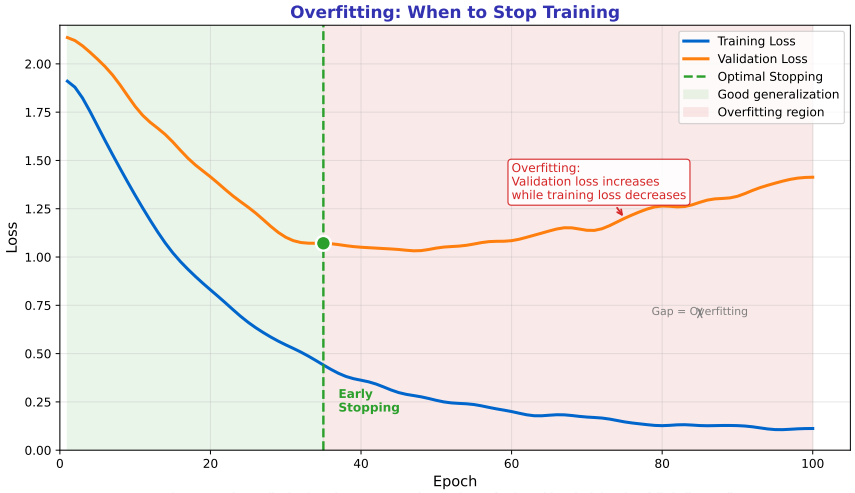
- Not enough examples
- Training set not representative
- Noise gets learned as signal

Training Too Long:

- Model eventually memorizes
- Needs early stopping

When your model memorizes instead of learns

Training vs Validation Loss



Finance Analogy: Like backtesting a strategy that works perfectly on historical data but fails in live trading

overfitting_curve

Training loss decreases but validation increases

Finance: The Backtest Trap

The Trap
Every trading strategy looks good on historical data – that's how you found it!

The Process:

1. Try many strategies
2. Keep the one that worked best
3. By construction, it fits the past
4. Future performance? Unknown.

Multiple Testing:

- Try 1000 random strategies
- Best one has Sharpe 2.0
- Is it skill or luck?

Why Finance Overfits Easily

1. **Limited Data**
   - 20 years = 5000 trading days
   - Few independent observations
2. **Low Signal-to-Noise**
   - Markets are noisy
   - Easy to fit noise
3. **Non-Stationarity**
   - Regimes change
   - Past may not predict future
4. **Look-Ahead Bias**
   - Using future information
   - Subtle but deadly

"Every strategy looks good on historical data"

Why Finance Overfits So Easily

Data Limitations

Domain	Samples
ImageNet	1,200,000
MNIST	60,000
Stock returns (daily, 10y)	2,520
Stock returns (monthly, 50y)	600
Market crashes	$\sim$10

The Problem:
Neural networks have thousands of parameters but only thousands of data points.

Signal vs Noise

Image Classification:

- A cat is always a cat
- Signal is strong and consistent
- R^2 can reach 99%+

Stock Prediction:

- Returns are mostly random
- Signal is weak and changing
- R^2 of 1% is excellent!

Implication:
Standard ML practices don't directly transfer to finance.

Limited data, high noise, non-stationary markets

Detecting Overfitting

Train/Validation/Test Split

1. **Training Set** (60-80%)
   - Used to fit weights
2. **Validation Set** (10-20%)
   - Used to tune hyperparameters
   - Monitor for overfitting
3. **Test Set** (10-20%)
   - Final evaluation only
   - Touch only once!

Key Rule:
Never use test data for decisions.

Warning Signs

Overfitting Indicators:

- Training loss $\ll$ validation loss
- Validation loss starts increasing
- Model predictions are "too confident"
- Performance degrades out-of-sample

For Finance:

- Backtest Sharpe $\gg$ live Sharpe
- Strategy "stops working"
- Drawdowns worse than expected

Always monitor out-of-sample performance

Discussion Question

"How would you know if your stock prediction model is overfitting? What specific symptoms would you look for?"

Consider:

In Training:

- Training/validation gap
- Validation loss trend
- Prediction confidence

In Production:

- Live vs. backtest performance
- Regime sensitivity
- Transaction cost impact

Best Practice: Always maintain a truly out-of-sample test set that you evaluate only once.

Think-Pair-Share: 3 minutes

Preview: Fighting Overfitting

Solutions (Module 4)

1. **L1/L2 Regularization**
   - Penalize large weights
   - Simpler models
2. **Dropout**
   - Randomly disable neurons
   - Ensemble effect
3. **Early Stopping**
   - Stop before overfitting
   - Use validation loss
4. **Data Augmentation**
   - Create more training data
   - Finance: bootstrap?

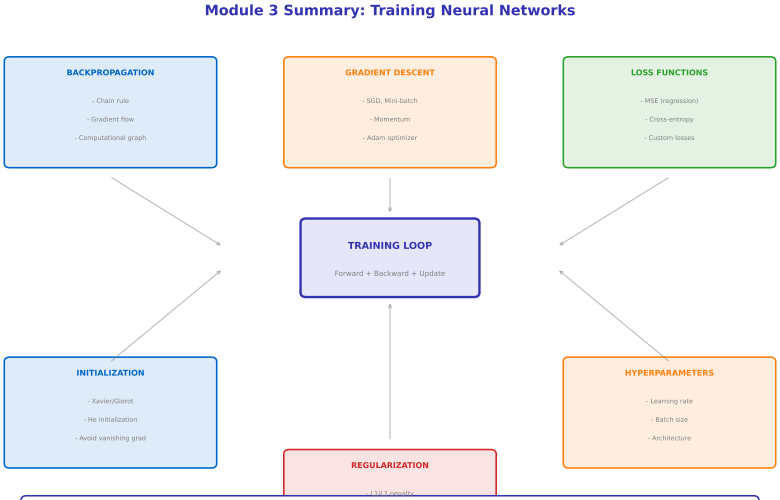
Finance-Specific

1. **Walk-Forward Validation**
   - Respect time ordering
   - Rolling windows
2. **Cross-Validation Variants**
   - Purged CV
   - Combinatorial CV
3. **Ensemble Methods**
   - Average multiple models
   - Reduce variance

Module 4 will cover these in detail.

Module 4 will cover solutions: regularization, dropout, early stopping

Training Pipeline Overview



module3_summary_diagram

Module 3: Key Takeaways

What We Learned

1. **Loss Functions**
   - Measure prediction error
   - MSE, cross-entropy
   - Define what "good" means
2. **Gradient Descent**
   - Follow the slope downhill
   - Learning rate matters
   - Batch vs stochastic
3. **Backpropagation**
   - Chain rule for credit assignment
   - Error flows backward
   - Enables efficient gradient computation
4. **Training Dynamics**
   - Epochs and batches
   - Monitoring with curves
   - Vanishing gradients
5. **Overfitting**
   - Memorizing vs learning
   - Train/val/test split
   - Finance-specific challenges

The Big Picture:

We can now train neural networks. But making them work well requires more...

From measuring error to updating weights

What We've Built So Far

Modules 1-3 Foundation

1. **Module 1: Architecture**
   - Perceptron basics
   - Linear decision boundaries
   - Limitations (XOR)
2. **Module 2: MLPs**
   - Hidden layers
   - Non-linear activation
   - Universal approximation
3. **Module 3: Training**
   - Gradient descent
   - Backpropagation
   - Overfitting awareness

You Can Now:

- Explain how neural networks compute
- Understand the training process
- Recognize overfitting
- Follow the math (or know where to look)

What's Missing:

- Practical regularization
- Real-world applications
- Finance case studies
- Modern developments

Modules 1-3: The complete neural network foundation

Key Questions for Reflection

Think about these as you move to Module 4:

1. **Loss vs. Profit:**
   Why might minimizing MSE not maximize trading profit? What loss function would better align with trading goals?
2. **Overfitting in Finance:**
   With only 20 years of daily data, how many parameters can we safely learn? What's the ratio of samples to parameters you'd be comfortable with?
3. **Non-Stationarity:**
   If market regimes change, what does that mean for our training strategy? Should we weight recent data more heavily?
4. **The Efficient Market Hypothesis:**
   If markets are efficient, can neural networks find persistent patterns? What would success look like?

Reflect on the learning process

Preview: Module 4

"Theory meets practice. How do we actually use neural networks in finance?"

Coming Up:

- Regularization techniques
  - L1/L2, dropout, early stopping
- Financial data challenges
  - Non-stationarity, noise
- Complete case study
  - Stock prediction end-to-end

Also:

- Modern architectures overview
  - CNN, RNN, Transformers
- Limitations and ethics
  - Black-box decisions
  - Regulatory concerns
- Future directions
  - Where the field is heading

Mathematical details: See Appendix B-D for derivations

Next: Regularization, case studies, and modern developments

Module 1: The Birth of Neural Computing

From Biological Inspiration to the Perceptron (1943-1969)

Neural Networks for Finance

BSc Lecture Series

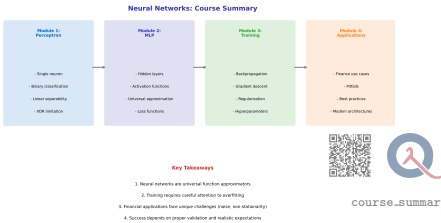
November 28, 2025

The Journey So Far

What We've Covered:

- **Module 1:** The Perceptron
  - Single neuron, decision boundaries
  - XOR limitation → AI Winter
- **Module 2:** Multi-Layer Perceptrons
  - Hidden layers, activation functions
  - Universal Approximation Theorem
- **Module 3:** Training
  - Gradient descent, backpropagation
  - Overfitting warning signs

The Foundation is Complete



Perceptron → MLP → Training: The complete foundation

The Final Question

"How do we actually use this for stock prediction?"

From theory to practice:

- How do we prevent overfitting in finance?
- What makes financial data different?
- Does this actually work?
- What are the ethical considerations?

Theory meets practice

Module 4 Roadmap

1. **Historical Context** (2012-Present)
   - The deep learning revolution
2. **Regularization Techniques**
   - L1/L2, dropout, early stopping
3. **Financial Data Challenges**
   - Non-stationarity, regime changes, biases
4. **Case Study: Stock Prediction**
   - S&P 500 direction prediction (realistic assessment)
5. **Modern Architectures**
   - CNN, RNN, Transformer overview
6. **Limitations and Ethics**
   - What neural networks can and cannot do

From theory to real-world applications

The Reality Check

Theory is Clean:

- Data is stationary
- Training set represents test set
- Patterns persist
- No transaction costs
- Unlimited computing power

Finance is Messy:

- Markets change constantly
- Past may not predict future
- Regime changes happen
- Costs eat into profits
- Latency matters

Warning: Paper profits $\neq$ Real profits

"Theory is clean. Finance is messy."

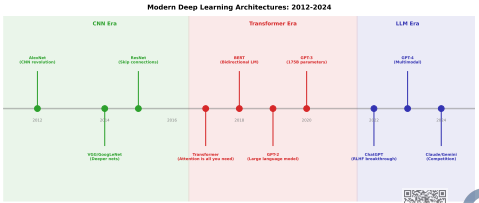
2012: The Deep Learning Revolution

AlexNet (Krizhevsky et al., 2012):

- ImageNet competition: 1.2M images, 1000 classes
- **Error rate: 15.3%** (vs. 26.2% second place)
- Deep convolutional neural network (8 layers)

Why This Mattered:

- 10+ percentage points better than alternatives
- Proved deep learning works at scale
- GPU training (2x NVIDIA GTX 580)
- Started the deep learning “gold rush”



modern_architectures_timelin

AlexNet: When deep learning proved its superiority

What Made Deep Learning Work?

Three Factors Converged in the 2010s:

1. Big Data

- ImageNet: 14M+ images
- Internet scale data
- Labeled datasets
- In finance: tick data, alternative data

2. GPU Computing

- Parallel matrix operations
- 100x speedup vs CPU
- CUDA programming
- Cloud GPU access

3. Better Algorithms

- ReLU activation
- Dropout regularization
- Batch normalization
- Better optimizers (Adam)

All three were necessary; none was sufficient alone

The convergence of data, compute, and algorithms

2017: Attention Is All You Need

The Transformer Architecture (Vaswani et al., 2017):

- Originally for machine translation
- Key innovation: **Self-attention mechanism**
- No recurrence needed $\rightarrow$ parallelizable

Self-Attention Intuition:

- Each position "attends" to all other positions
- Learns which inputs are relevant to each other
- "The cat sat on the mat because *it* was tired"
- Attention reveals that "it" refers to "cat"

Attention Formula:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

- Q: Query (what am I looking for?)
- K: Key (what do I have?)
- V: Value (what do I return?)

Vaswani et al.: The architecture that changed everything

2020+: The GPT Era

Scaling Laws and Foundation Models:

Key Developments:

- GPT-2 (2019): 1.5B parameters
- GPT-3 (2020): 175B parameters
- GPT-4 (2023): rumored 1T+ parameters
- ChatGPT: Conversational interface

Scaling Discovery:

- Performance scales predictably with:
  - Model size
  - Dataset size
  - Compute budget

Impact on Finance:

- Sentiment analysis from news/social media
- Document understanding (10-K filings)
- Natural language queries for data
- Automated research summarization

But: LLMs don't predict stock prices

- Different problem domain
- Time series $\neq$ language patterns

From GPT-2 to GPT-4 and beyond

Discussion Question

"Why did neural networks succeed in 2012 but not in 1990?

What changed?"

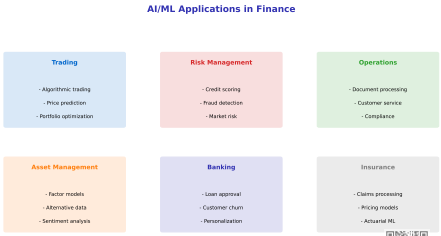
- Was it just computing power?
- What role did data play?
- Were the algorithms fundamentally different?
- Could we have predicted this breakthrough?

Think-Pair-Share: 3 minutes

AI in Finance Today

Major Players:

- **Renaissance Technologies**
  - Medallion Fund: 66% avg. return (1988-2018)
  - Highly secretive, physics/math PhDs
- **Two Sigma**
  - $60B+ AUM
  - Heavy ML/AI focus
- **Citadel**
  - Market making + hedge fund
  - ML for high-frequency trading



Common Applications: Signal generation, portfolio optimization, risk management, alternative data analysis

ai_applications_financ

Renaissance, Two Sigma, Citadel: Industry adoption

The Current Landscape

What's Actually Working:

- Risk management and fraud detection
- High-frequency market making
- Alternative data processing
- Portfolio optimization
- Credit scoring
- Sentiment analysis

What's Mostly Hype:

- "AI that beats the market consistently"
- Perfect stock price prediction
- Fully automated trading for retail
- "Guaranteed returns" from AI

Red Flag: If someone claims their AI consistently beats the market, ask why they're selling it instead of using it.

Separating reality from marketing

The Overfitting Problem Revisited

Recall from Module 3:

- Model learns training data too well
- Memorizes noise instead of patterns
- Fails on new, unseen data

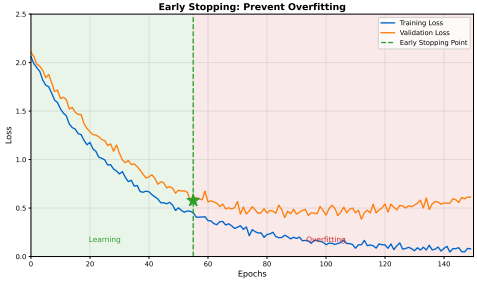
In Finance, This Is Critical:

- Backtest shows 40% annual returns
- Live trading shows -15%
- This happens constantly

Why Module 4 Focuses on This:

- Overfitting is the #1 failure mode
- Financial data is especially prone
- Must master regularization techniques

The Overfitting Gap:



early_stoppin

Overfitting: The greatest challenge in financial ML

Why Finance Overfits So Easily

Limited Data:

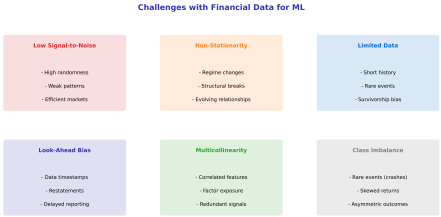
- 20 years of daily data = 5,000 samples
- Compare to ImageNet: 14,000,000 images
- Regime changes reduce effective samples further

High-Dimensional Features:

- 50 technical indicators $\times$ 10 lookbacks = 500 features
- More parameters than data points = guaranteed overfitting

Low Signal-to-Noise:

- Daily stock returns: 95%+ noise
- Real patterns are tiny



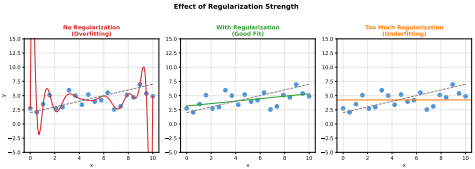
financial_data_challenge

Limited data, high noise, changing regimes

L2 Regularization (Ridge)

The Idea: Add penalty for large weights

$$\mathcal{L}_{reg} = \mathcal{L} + \frac{\lambda}{2}\|\mathbf{W}\|_2^2 = \mathcal{L} + \frac{\lambda}{2}\sum_i w_i^2$$



Effect on Optimization:

- Original gradient: $\nabla_w \mathcal{L}$
- With L2: $\nabla_w \mathcal{L} + \lambda w$
- Weights decay toward zero each update
- Also called "weight decay"

Hyperparameter λ:

- $\lambda = 0$: No regularization
- λ large: All weights $\to 0$
- Typical: 10^{-4} to 10^{-2}

regularization_effec

Push weights to be small

L2 Intuition

Why Does Penalizing Large Weights Help?

Mathematical View:

- Large weights $\rightarrow$ extreme predictions
- Small changes in input $\rightarrow$ big output changes
- High sensitivity = memorization
- L2 forces smoother functions

Bayesian View:

- L2 = Gaussian prior on weights
- Prior belief: weights should be small
- More data $\rightarrow$ prior matters less

Finance Analogy:

- Large weight on one feature = "betting everything on one stock"
- Risky: what if that feature stops working?
- L2 forces diversification across features
- No single feature dominates the prediction

Key Insight:

- L2 doesn't eliminate features
- Just reduces their influence
- All features contribute, but moderately

Don't let any single feature dominate

L1 Regularization (Lasso)

The Idea: Penalty proportional to absolute value

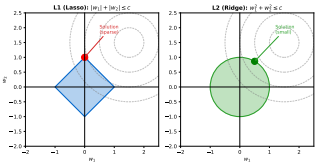
$$\mathcal{L}_{reg} = \mathcal{L} + \lambda \|\mathbf{W}\|_1 = \mathcal{L} + \lambda \sum_i |w_i|$$

Key Difference from L2:

- L1 pushes weights to **exactly zero**
- Creates sparse models (feature selection)
- Automatically identifies irrelevant features

Why Sparsity?

- L1 gradient is $\pm\lambda$ (constant)
- Small weights get pushed to zero
- L2 gradient is λw (proportional)
- Small weights shrink slowly, never reach zero



L1 vs L2 Comparison

Property	L1 (Lasso)	L2 (Ridge)
Constraint	Diamond	Circle
Sparsity	Yes (zeros)	No
Feature Selection	Automatic	No
Correlated Features	Picks one	Shrinks all
Computational	Harder	Easier

l1_vs_l2

Push some weights to exactly zero: feature selection

L1 vs L2: Comparison

Property	L1 (Lasso)	L2 (Ridge)
Penalty term	$\lambda \sum \lvert w_i \rvert$	$\frac{\lambda}{2} \sum w_i^2$
Effect on weights	Some become exactly 0	All shrink toward 0
Feature selection	Yes (automatic)	No
Correlated features	Picks one arbitrarily	Shares weight among them
Sparsity	Sparse solutions	Dense solutions
Computation	Non-differentiable at 0	Smooth, differentiable
Use when	Few features matter	All features may matter

Elastic Net: Combine both: $\lambda_1 \|W\|_1 + \lambda_2 \|W\|_2^2$
Best of both worlds for correlated features

L1 for sparsity, L2 for shrinkage

Dropout: Random Deactivation

The Idea (Hinton et al., 2012):

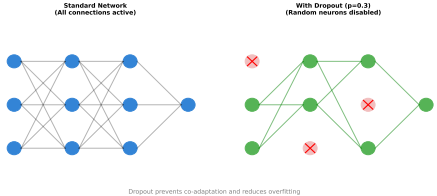
- During training: randomly "drop" neurons
- Each neuron has probability p of being set to 0
- Typically $p = 0.5$ for hidden, $p = 0.2$ for input

Training:

- Each mini-batch sees different network
- Forces redundancy in learned features
- No neuron can become a "crutch"

Inference:

- Use all neurons (no dropout)
- Scale outputs by $(1 - p)$ or use "inverted dropout"



dropout_visualizatio

"No single neuron becomes a crutch"

Discussion Question

"How is dropout like diversifying a portfolio?"

- What happens if you bet everything on one stock?
- What happens if a neural network relies on one neuron?
- How does diversification protect against failure?
- How does dropout force the network to diversify?

Think-Pair-Share: 3 minutes

Dropout Intuition: Ensemble Learning

Ensemble Interpretation:

- Network with n neurons has 2^n possible subnetworks
- Dropout trains all subnetworks simultaneously
- Each mini-batch samples a different subnetwork
- Final prediction: average of all subnetworks

Why Ensembles Work:

- Different models make different errors
- Averaging reduces variance
- More robust to noise

Finance Parallel:

- One analyst: high variance predictions
- Committee of analysts: more stable
- Dropout = "committee of networks"

Practical Notes:

- Dropout slows convergence
- Needs more epochs to train
- Don't use with batch normalization (debate)
- Less common in CNNs today

Dropout approximates training an ensemble of networks

Early Stopping

The Simplest Regularization:

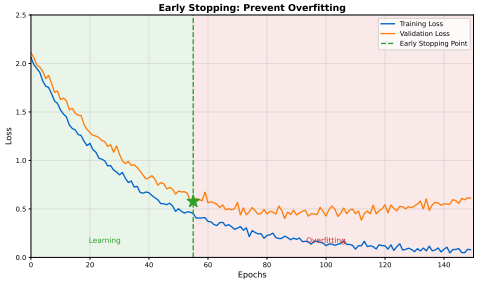
- Monitor validation loss during training
- Stop when validation loss stops improving
- Use the model from the best epoch

Implementation:

- Track best validation loss
- Patience: wait k epochs before stopping
- Save checkpoint at each improvement
- Restore best checkpoint at end

Why It Works:

- Early epochs: learning real patterns
- Later epochs: memorizing training noise
- Sweet spot: generalization peak



Typical patience: 5-20 epochs

early_stoppin

Stop training when validation loss stops improving

Walk-Forward Validation for Time Series

Standard Cross-Validation: WRONG for Time Series

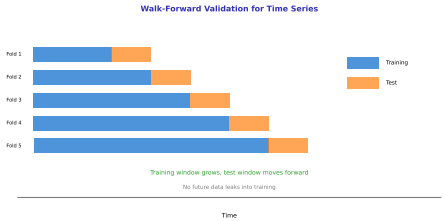
- Random splits leak future information
- Model sees 2024 data, predicts 2023
- Guaranteed overfitting

Walk-Forward Validation:

- Train on [2010-2015], validate on [2016]
- Train on [2010-2016], validate on [2017]
- Train on [2010-2017], validate on [2018]
- Always: train on past, validate on future

Anchored vs Rolling Window:

- Anchored: always start from same date
- Rolling: fixed window slides forward



walk_forward_validatio

Train on past, validate on future (never the reverse)

Fighting Overfitting: Summary

Technique	Mechanism	When to Use
L2 (Ridge)	Penalize large weights	Always (as baseline)
L1 (Lasso)	Push weights to zero	Feature selection needed
Dropout	Random neuron deactivation	Deep networks
Early Stopping	Stop before overfitting	Always (free)
Walk-Forward	Time-respecting validation	Time series only

Practical Recommendation for Finance:

1. Always use walk-forward validation
2. Start with L2 regularization
3. Add early stopping (patience=10)
4. Try dropout (0.2-0.5) for deep networks
5. Use L1 if you need interpretable feature importance

Multiple defenses against overfitting

The Nature of Financial Data

Financial Data is Fundamentally Different:

Images/Text:

- Patterns are stable over time
- Cat in 2020 looks like cat in 2010
- English grammar doesn't change daily
- High signal-to-noise ratio
- Abundant labeled data

Financial Markets:

- Patterns change constantly
- Strategies that work get arbitraged away
- Regime changes (bull/bear/crisis)
- Extremely low signal-to-noise
- Limited history, no "labels" for future

Key Insight: Success in image recognition doesn't translate to finance.
The problems are fundamentally different.

Financial data is fundamentally different from images or text

Non-Stationarity

Definition:

- Statistical properties change over time
- Mean, variance, correlations all shift
- Model trained on past may fail on future

Causes in Finance:

- Central bank policy changes
- Market structure evolution (HFT, ETFs)
- Regulatory changes
- Technology disruption
- Global events (pandemics, wars)

Implication:

- Models have "shelf life"
- Need regular retraining
- "What worked" $\neq$ "what will work"

regime_change

The patterns that worked yesterday may not work tomorrow

Regime Changes

Markets Switch Between Fundamentally Different Behaviors:

Bull Market:

- Upward trend
- Low volatility
- Mean reversion works
- Risk-on behavior
- Correlations low

Bear Market:

- Downward trend
- High volatility
- Momentum works
- Risk-off behavior
- Correlations spike

Crisis:

- Extreme moves
- "All correlations go to 1"
- Historical patterns break
- Liquidity disappears
- Fat tails dominate

Challenge: You don't know which regime you're in until it's over.
Solution: Train separate models or use regime detection.

Markets switch between fundamentally different behaviors

Noise vs Signal

Signal-to-Noise Ratio (SNR):

- Daily stock returns: SNR $\approx$ 0.05
- Speech recognition: SNR $\approx$ 10-20
- 200-400x harder!

What This Means:

- 95%+ of price movement is random
- True patterns are tiny
- Easy to find spurious patterns
- Need massive data to detect signal

Example:

- Average daily return: 0.04%
- Daily standard deviation: 1%
- Signal = return / std = 0.04

Implications for ML:

- Models will find patterns in noise
- Backtests look amazing
- Live performance disappoints
- Need extreme skepticism

Reality Check:

- If returns were 50% predictable, you'd be a billionaire in months
- Markets are efficient enough that small edges are huge
- 55% accuracy is actually impressive

Most price movement is noise, not signal

Look-Ahead Bias

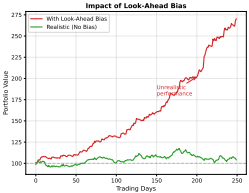
Definition: Using information that wasn't available at decision time.

Common Mistakes:

- Using today's adjusted close to trade at today's open
- Normalizing with full dataset statistics
- Including stocks that didn't exist yet
- Using restated (revised) financial data
- Feature engineering with future data

Example:

- Train on 2020-2023
- Normalize: subtract mean, divide by std
- Problem: Mean includes 2023!
- In 2020, you didn't know 2023 stats



Prevention:

- Point-in-time data
- Rolling normalization
- Careful feature engineering

look_ahead_bia

The silent killer of backtests

Survivorship Bias

Definition: Only successful companies remain in the dataset.

The Problem:

- S&P 500 today has survivors
- Enron, Lehman, Bear Stearns are gone
- Your model never sees failures
- Learns patterns of survivors only

Impact:

- Overstates historical returns
- "Average stock returned 10%/year"
- Actually includes only winners

Real Example:

- Study: "Value stocks beat growth"
- Used current value stocks list
- Many value stocks went bankrupt
- True effect was much smaller

Solution:

- Point-in-time constituent lists
- Include delisted companies
- Use survivorship-bias-free databases
- Account for delisting returns

Your dataset doesn't include the failures

Discussion Question

"What makes financial prediction harder than image recognition?"

- A cat is always a cat. Is a bull market always a bull market?
- ImageNet has 14 million labeled images. How many "market crashes" exist?
- If everyone uses the same model, what happens?
- Does finding patterns in finance make them disappear?

Think-Pair-Share: 3 minutes

Data Preprocessing for Finance

Essential Preprocessing Steps:

Normalization:

- Z-score: $\frac{x-\mu}{\sigma}$
- Use rolling window (e.g., 252 days)
- Never use future data!

Missing Data:

- Forward fill (most common)
- Linear interpolation
- Drop if too many missing
- Never: backward fill

Outlier Handling:

- Winsorize at 1%/99% percentile
- Or use robust statistics (median)
- Don't remove outliers blindly!
- Crashes are real data

Feature Engineering:

- Returns not prices (stationarity)
- Log returns for mathematical convenience
- Technical indicators as features
- Lag features appropriately

Proper preprocessing is essential

Case Study: S&P 500 Direction Prediction

A Realistic Example from Start to Finish

Goal:

- Predict S&P 500 next-day direction
- Binary: Up or Down?
- Use only information available at market close

Why This Problem:

- Simple, well-defined target
- Abundant data
- Common industry problem
- Illustrates key challenges

Our Approach:

1. Define features and target
2. Choose architecture
3. Set up walk-forward validation
4. Train and evaluate
5. Reality check the results

Spoiler: Results will be modest.
That's the honest truth about financial ML.

A realistic example from start to finish

Problem Definition

Target Variable:

$$y_t = \begin{cases} 1 & \text{if } R_{t+1} > 0 \\ 0 & \text{if } R_{t+1} \leq 0 \end{cases}$$

where $R_{t+1} = \frac{P_{t+1} - P_t}{P_t}$ is next-day return.

Baseline:

- Random guess: 50% accuracy
- Actual: S&P 500 up 53% of days (long-term)
- "Always predict up": 53% accuracy

Goal:

- Beat 53% consistently
- Out-of-sample (not just backtest)
- After transaction costs

Data:

- Period: 2000-2023 (24 years)
- Frequency: Daily
- Samples: ∼6,000 trading days

Important Notes:

- This is harder than it sounds
- Small edge = big money
- Markets are highly efficient
- Most published research overfits

Binary classification: Up or Down?

Input Features

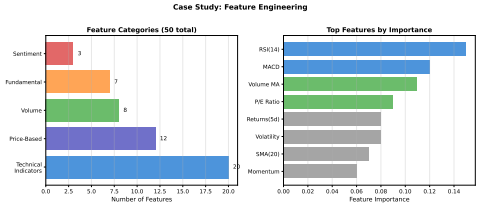
Technical Indicators (15 features):

- Returns: 1-day, 5-day, 20-day
- Moving averages: 10/50/200-day ratios
- Volatility: 20-day rolling std
- RSI (14-day), MACD
- Bollinger Band position
- Volume ratio (vs 20-day avg)

Market Factors (5 features):

- VIX level and change
- Treasury yield (10Y)
- Credit spread
- Put/Call ratio

Preprocessing: Rolling z-score (252-day window), clip at ± 3



Total: 20 input features

case_study_feature

15 technical indicators + 5 market factors

Architecture Decision

Network: 20-16-8-1

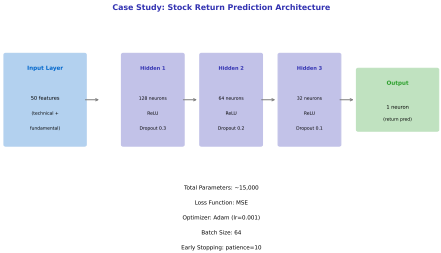
- Input: 20 features
- Hidden 1: 16 neurons (ReLU)
- Hidden 2: 8 neurons (ReLU)
- Output: 1 neuron (Sigmoid)

Why This Architecture?

- Relatively shallow (avoid overfitting)
- Decreasing width (funnel shape)
- Total parameters: $\sim$500
- Parameters $<<$ samples (6,000)

Regularization:

- L2: $\lambda = 0.001$
- Dropout: 0.2 (after each hidden layer)
- Early stopping: patience=10



case_study_architectur

Balancing model capacity with overfitting risk

Training Setup

Walk-Forward Validation:

Data Split:

- Training: 10 years (2,500 days)
- Validation: 2 years (500 days)
- Test: 2 years (500 days)
- Roll forward by 1 year, retrain

Training Details:

- Optimizer: Adam (lr=0.001)
- Loss: Binary cross-entropy
- Batch size: 64
- Max epochs: 200
- Early stopping: patience=10

Walk-Forward Windows:

Train	Valid	Test
2000-09	2010-11	2012-13
2001-10	2011-12	2013-14
2002-11	2012-13	2014-15
...	...	...
2010-19	2020-21	2022-23

Total: 10 test windows

10 years training, 2 years validation, 2 years test

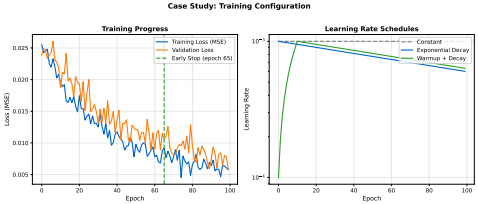
Results: Training Progress

Typical Training Run (2010-2019 → 2022-23):

- Training loss decreases smoothly
- Validation loss: decreases, then flat
- Early stopping at epoch 45-80
- Gap between train/val loss: moderate

Observations:

- Good: Not severe overfitting
- Good: Validation loss improves
- Moderate: Some train-val gap
- Training accuracy: 58-62%
- Validation accuracy: 54-56%



case_study_trainin

Monitoring the training process

Results: Accuracy

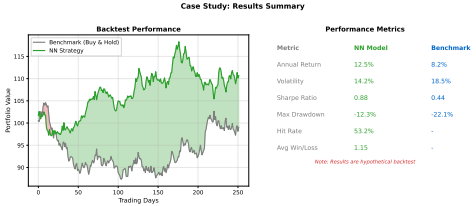
Out-of-Sample Results (2012-2023):

- Average test accuracy: **54.2%**
- Range across windows: 51.8% - 56.7%
- Baseline (always up): 53.1%
- Edge over baseline: +1.1%

By Year:

- Best: 2017 (56.7%) - low volatility
- Worst: 2020 (51.8%) - COVID crash
- Average bull market: 55.1%
- Average bear market: 52.4%

Is 54.2% good? It depends on costs and execution...



case_study_result

54.2% accuracy - is this good?

Discussion Question

"If a model is 54% accurate at predicting direction,

is it profitable?"

- What if each trade costs 0.1% in fees and slippage?
- What if you trade once per day vs once per month?
- Does accuracy equal profitability?
- What other metrics matter?

Think-Pair-Share: 3 minutes

Beyond Accuracy: Risk-Adjusted Returns

Accuracy $\neq$ Profitability

What Accuracy Misses:

- Size of wins vs losses
- 54% accuracy with small wins, large losses = loss
- Timing of predictions
- Risk taken to achieve returns

Better Metrics:

- **Sharpe Ratio**: $\frac{\text{Return} - R_f}{\text{Volatility}}$
- **Max Drawdown**: Largest peak-to-trough loss
- **Win/Loss Ratio**: Avg win / Avg loss

Our Case Study:

- Annual return: 8.2% (vs 9.5% buy-hold)
- Volatility: 12.1% (vs 18.2% buy-hold)
- Sharpe: 0.68 (vs 0.52 buy-hold)
- Max drawdown: 18% (vs 34% buy-hold)

Interpretation:

- Lower return than buy-hold
- But much lower risk
- Better risk-adjusted performance
- Before costs!

Accuracy is not the same as profitability

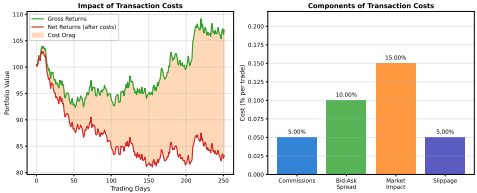
Reality Check: Transaction Costs

Types of Costs:

- Commission: \$0-10 per trade (retail)
- Bid-ask spread: 0.01%-0.1%
- Market impact: depends on size
- Slippage: execution vs expected price

Our Strategy:

- Trades: 252 days/year (daily)
- Round-trip cost: 0.1% (conservative)
- Annual cost: $252 \times 0.1\% = 25.2\%$
- Gross return: 8.2%
- **Net return: -17%**



Lesson: Daily trading requires extremely high accuracy to be profitable.

transaction_cost

Costs can eliminate paper profits entirely

The Efficient Market Hypothesis

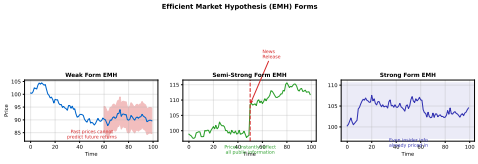
EMH (Fama, 1970):
"Prices fully reflect all available information"

Three Forms:

- **Weak:** Can't profit from past prices
- **Semi-strong:** Can't profit from public info
- **Strong:** Can't profit from any info

Implications for ML:

- If EMH true: all patterns are noise
- If EMH false: patterns exist but are small
- Reality: markets are "mostly efficient"
- Small, temporary inefficiencies exist



Grossman-Stiglitz Paradox:
If markets were perfectly efficient, no one would do research, so they couldn't be efficient.

emh_visualizatio

Markets are (mostly) efficient

What Works and What Doesn't

What Works (Maybe):

- Risk management and hedging
- Alternative data processing
- High-frequency market making
- Factor model enhancement
- Portfolio optimization
- Regime detection

Where NNs Add Value:

- Complex non-linear relationships
- High-dimensional feature spaces
- Alternative data (satellite, NLP)
- Execution optimization

What Doesn't Work:

- "Predicting stock prices" (directly)
- Black-box trading systems
- Complex models on small data
- Ignoring transaction costs
- Overfitting to backtests

Honest Expectations:

- Small edges are valuable
- 55% accuracy is impressive
- Risk management $>$ alpha generation
- Domain knowledge essential

Setting appropriate expectations for neural networks in finance

Beyond MLPs: Modern Architectures

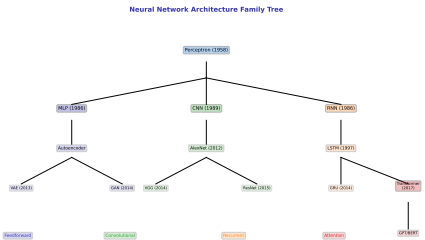
The MLP Foundation:

- Everything we learned applies to modern architectures
- Backpropagation: same algorithm
- Activation functions: same choices
- Regularization: same techniques

Key Modern Architectures:

1. **CNN**: Convolutional Neural Networks
2. **RNN/LSTM**: Recurrent Networks
3. **Transformer**: Attention-based

Common Thread:
All contain feedforward (MLP) components!



architecture_family_tre

MLPs are the foundation for everything that followed

Convolutional Neural Networks

Key Idea: Learnable pattern detectors

- Convolutional filters slide over input
- Detect local patterns (edges, shapes)
- Weight sharing reduces parameters
- Hierarchical feature learning

For Time Series:

- 1D convolutions over time
- Detect patterns in price/volume sequences
- Filter learns what to look for
- E.g., "head and shoulders" pattern

Architecture:

$$\text{Conv1D} \rightarrow \text{ReLU} \rightarrow \text{Pool}$$

$$\rightarrow \text{Conv1D} \rightarrow \text{ReLU} \rightarrow \text{Pool}$$

$$\rightarrow \text{Flatten} \rightarrow \text{MLP} \rightarrow \text{Output}$$

Finance Use Cases:

- Technical pattern recognition
- Order book analysis
- Multi-asset correlation patterns

CNNs: Finding patterns with learnable filters

Recurrent Neural Networks

Key Idea: Memory for sequences

- Process sequences one step at a time
- Maintain hidden state (memory)
- Output depends on current + past inputs
- Natural for time series

RNN Update:

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t + b)$$

Problem: Vanishing gradients over long sequences

LSTM Solution (1997):

- Forget gate: what to discard
- Input gate: what to add
- Output gate: what to reveal
- Cell state: long-term memory

Finance Use Cases:

- Time series forecasting
- Sequence-to-sequence (prices)
- Combining with attention

RNNs and LSTMs: Designed for sequences

Transformers and Attention

Key Innovation: Self-attention

- Each position attends to all others
- No recurrence needed
- Parallelizable (fast training)
- Captures long-range dependencies

Components:

- Multi-head attention
- Feedforward layers (MLPs!)
- Layer normalization
- Positional encoding

Attention Formula:

$$\text{Attn} = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Finance Use Cases:

- News/sentiment analysis (NLP)
- Document understanding
- Multi-asset attention
- Temporal attention for prices

The architecture behind GPT and modern NLP

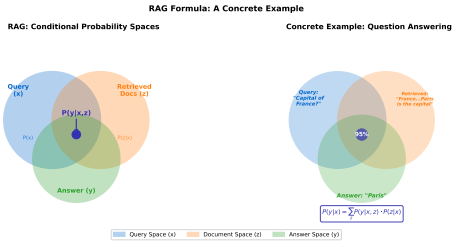
RAG Formula: A Concrete Example

Retrieval-Augmented Generation (RAG):

- Combines retrieval with generation
- Query x retrieves relevant documents z
- Model generates answer y conditioned on both

The RAG Formula:

$$P(y|x) = \sum_{z} P(y|x,z) \cdot P(z|x)$$



- $P(z|x)$: Retriever finds relevant docs
- $P(y|x,z)$: Generator produces answer
- Marginalizes over retrieved documents

Finance Application: Retrieve relevant filings/news, then generate analysis conditioned on context.

The MLP Foundation

Every Modern Architecture Contains MLPs:

CNN:

- Conv layers: shared MLPs
- Final classifier: MLP
- Same activation functions

RNN/LSTM:

- Gate computations: MLPs
- Output layer: MLP
- Same backprop algorithm

Transformer:

- FFN after attention: MLP
- Position-wise: MLP
- 2/3 of parameters in MLPs!

What you learned in this course is the foundation for all of deep learning.

Perceptron $\rightarrow$ MLP $\rightarrow$ CNN/RNN/Transformer

Every modern architecture contains feedforward components

Discussion Question

"If you were building a financial AI startup today,

what architecture and problem would you focus on?"

- Direct price prediction vs risk management?
- Traditional features vs alternative data?
- Simple MLP vs complex Transformer?
- Retail product vs institutional tool?

Think-Pair-Share: 3 minutes

The Black Box Problem

The Interpretability Challenge:

- Neural networks: millions of parameters
- No simple explanation for decisions
- "Why did you sell?" - "Because weight 47,823 was 0.0032"

Why This Matters in Finance:

- Regulatory requirements (explainability)
- Risk management needs understanding
- Client trust requires explanation
- Debugging requires insight

Partial Solutions:

- SHAP values, LIME
- Attention visualization
- Simpler models where possible

Trade-off:

Simple	Complex
Interpretable	Black box
Linear	Non-linear
Stable	May overfit
Lower accuracy	Higher accuracy

Question:
Is 1% more accuracy worth losing all interpretability?

Neural networks are often difficult to interpret

Regulatory Requirements

Key Regulations:

- **MiFID II** (EU): Best execution, transparency
- **GDPR**: Right to explanation for automated decisions
- **SR 11-7** (US): Model risk management
- **Basel III**: Capital requirements, risk models

Explainability Mandates:

- Credit decisions must be explainable
- Trading algorithms need documentation
- Model validation required
- Audit trails essential

Trend: Increasing regulation of algorithmic decision-making

ethical_consideration

Regulations increasingly demand explainable AI

Systemic Risk from Correlated AI

What if everyone uses similar models?

The Problem:

- Similar training data
- Similar architectures
- Similar features
- $\Rightarrow$ Similar predictions
- $\Rightarrow$ Correlated trades
- $\Rightarrow$ Amplified market moves

Historical Example:

- August 2007: Quant meltdown
- Many funds used similar strategies
- All deleveraged simultaneously
- Massive losses in days

Flash Crash Risk:

- May 6, 2010: Dow dropped 1000 points in minutes
- Algorithmic trading implicated
- Feedback loops between systems

Mitigations:

- Circuit breakers
- Position limits
- Diversity requirements
- Human oversight
- Stress testing for crowding

Correlated AI trading could amplify market instability

Model Risk Management

"All models are wrong, some are useful" - George Box

Types of Model Risk:

- **Specification risk**: Wrong model type
- **Implementation risk**: Coding bugs
- **Data risk**: Bad inputs
- **Usage risk**: Misapplication

Famous Failures:

- LTCM (1998): Model assumptions failed
- Knight Capital (2012): $440M in 45 minutes
- London Whale (2012): VAR model issues

Governance Framework:

- Independent model validation
- Documentation requirements
- Regular backtesting
- Stress testing
- Change management
- Clear ownership

Key Principle:
Never deploy a model you don't understand well enough to know when it might fail.

Responsible deployment requires proper oversight

Historical Lesson: Responsible Innovation

AI Has Seen Hype Cycles Before:

The Pattern:

1. Breakthrough discovery
2. Excessive optimism/funding
3. Over-promising
4. Failure to deliver
5. "AI Winter" backlash
6. Quiet progress
7. Next breakthrough...

Examples:

- 1960s: "Machines will think in 20 years"
- 1980s: Expert systems will replace experts
- 2010s: "Deep learning solves everything"
- Today: "AGI is imminent"

Lesson:
Hype damages the field. Responsible claims and honest assessment help it grow sustainably.

Your Responsibility: Be honest about what neural networks can and cannot do.

Hype cycles damage the field; responsible claims help it grow

Where Neural Networks Add Value

Realistic Assessment of Neural Networks in Finance:

High Value Applications:

- **Risk Management**
  - Fraud detection
  - Credit scoring
  - Anomaly detection
- **Alternative Data**
  - Satellite imagery analysis
  - News sentiment
  - Social media signals
- **Execution**
  - Optimal order routing
  - Market making
  - Transaction cost analysis

Lower Value (Often Overhyped):

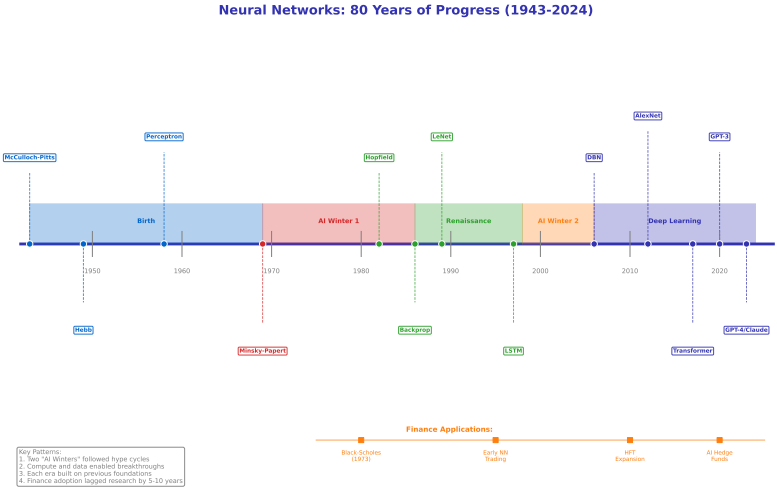
- Direct price prediction
- "AI-powered" retail trading apps
- Fully automated strategies
- Complex models on limited data

Key Insight:
Neural networks work best when:

- Abundant data available
- Clear signal exists
- Domain expertise integrated
- Proper validation done

Risk management, alternative data, market making

Neural Networks: 1943-2024



full_timeline_1943_202

From McCulloch-Pitts to GPT: 80 years of progress

Course Summary: Key Takeaways

Module 1 - Perceptron:

- Neuron as weighted voting
- Linear separability limits
- XOR problem $\rightarrow$ AI Winter

Module 2 - MLPs:

- Hidden layers solve XOR
- Activation functions enable non-linearity
- Universal Approximation Theorem

Module 3 - Training:

- Gradient descent finds minimum
- Backprop: efficient gradient computation
- Overfitting is the main enemy

Module 4 - Practice:

- Regularization fights overfitting
- Financial data is uniquely challenging
- Honest assessment of capabilities

The Foundation: Everything in modern AI builds on these concepts.

The essential concepts from all four modules

Where to Go from Here

Suggested Learning Path:

Theory:

- Deep Learning (Goodfellow et al.)
- Neural Networks and Deep Learning (Nielsen) - free online
- Stanford CS231n (CNNs)
- Stanford CS224n (NLP)

Practice:

- PyTorch or TensorFlow tutorials
- Kaggle competitions
- Personal projects
- Open-source contributions

Finance-Specific:

- Advances in Financial ML (de Prado)
- Machine Learning for Asset Managers
- QuantConnect, Zipline (backtesting)
- Academic papers (SSRN, arXiv q-fin)

Key Advice:

- Build things!
- Start simple, add complexity
- Focus on fundamentals
- Be skeptical of claims
- Domain knowledge matters

Suggested resources for continued learning

Thank You

Neural Networks for Finance
BSc Lecture Series

Questions?

See Mathematical Appendix for full derivations