

## Module 3: Learning from Mistakes

### Training Neural Networks with Backpropagation (1986-2012)

Neural Networks for Finance

BSc Lecture Series

November 26, 2025

*“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**

## 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.

## 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 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?

## 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

[charts/timeline\\_1986\\_2012/timeline\\_1986\\_2012.pdf](#)

# 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

## 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



## 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)

We need a way to measure how wrong our predictions are

## 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

## 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

## 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:

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

**Goal:**

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

---

The loss function quantifies prediction error

## 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 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
<b>MSE</b>		<b>4.67</b>

Units: (percentage points)<sup>2</sup>

**RMSE:**  $\sqrt{MSE} = 2.16\%$

“On average, we’re off by about 2%”

“How far off were we, on average?”

## 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.

[charts/mse\\_visualization/mse\\_visualization.pdf](#)

## Worked Example

### Predictions for 5 Stocks:

Stock	$\hat{y}$	$y$	Error <sup>2</sup>
AAPL	+5%	+2%	9
MSFT	+3%	+4%	1
GOOG	-1%	+2%	9
AMZN	+4%	+4%	0
META	+2%	-3%	25
<b>MSE</b>			<b>8.8</b>

$$\text{RMSE} = 2.97\%$$

## Interpretation

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

### Is This Good?

Depends on context:

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

### Reality Check:

Even small predictability ( $\text{RMSE slightly} < \text{volatility}$ ) can be valuable in trading.

---

Worked example with stock returns



*“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**

## 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)

`charts/loss_landscape_3d/loss_landscape_3d.pdf`

## 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?"

## 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:

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

---

Move in the direction that goes down

## 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

$\eta$  = learning rate (step size)

# 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  $\approx 0$

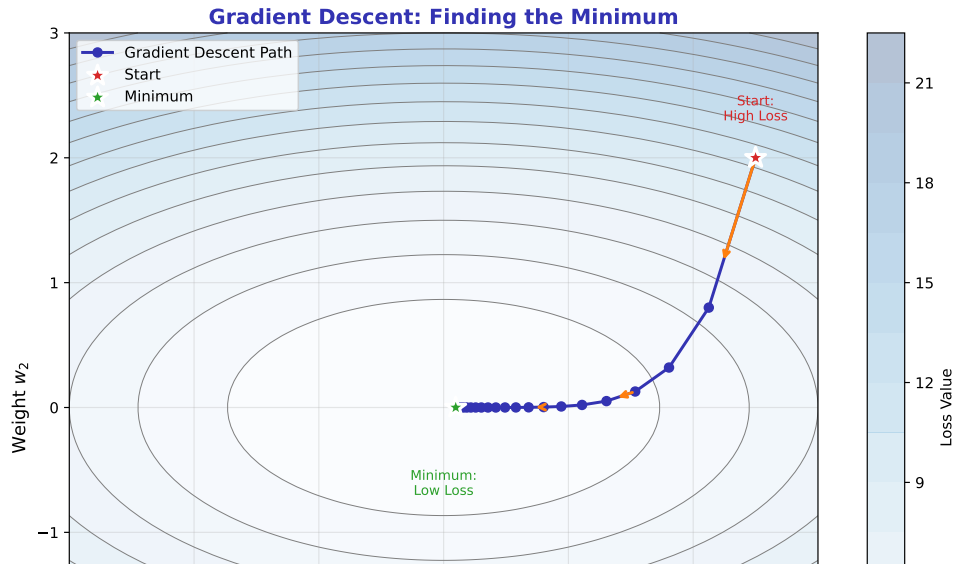
### At a Minimum:

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

No direction goes further down.

---

The gradient tells us which way is “up”



## 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

Similar to iterative portfolio rebalancing

## 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).



# The Learning Rate: Step Size

## The Hyperparameter $\eta$

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

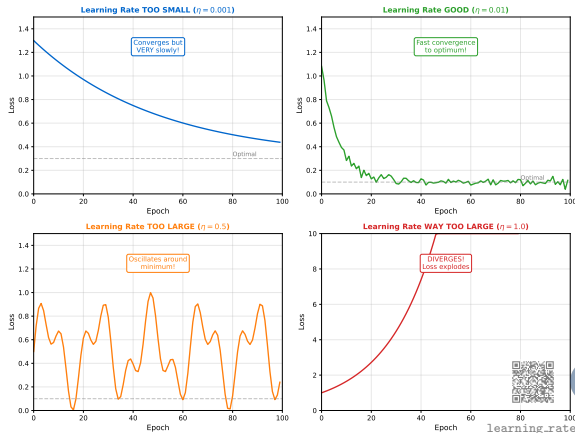
### $\eta$ Controls:

- 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: The Most Important Hyperparameter



Rule of thumb: Start with 0.001-0.01 and adjust based on training dynamics

Learning rate controls how far we move each step

## The Problem

When  $\eta$  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  $\eta$ .

---

Too big = overshoot the minimum

## The Problem

When  $\eta$  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  $\eta$  automatically.

---

Too small = converge too slowly

*“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 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:

$$w_1 \rightarrow h_1 \rightarrow h_2 \rightarrow \cdots \rightarrow \hat{y} \rightarrow \mathcal{L}$$

Each weight affects the loss through many intermediate steps.

---

The output was wrong. Which weights caused it?

## 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.

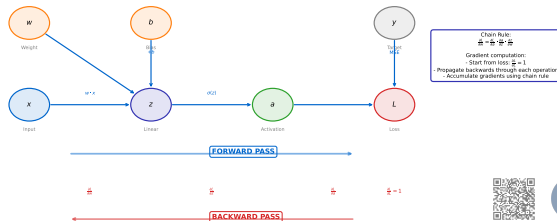
## Two Passes:

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

## Efficiency:

Computes ALL gradients in time proportional to one forward pass.

Computational Graph: Forward and Backward Pass



backprop.computational\_graph

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  $\rightarrow$  Ice cream sales  $\rightarrow$  Profit

How does temperature affect profit?

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

## Chain Rule: Foundation of Backpropagation

### Chain Rule: The Key to Backprop



Chain Rule:

$$\frac{d}{dx} f(g(x)) = \frac{df}{dg} \cdot \frac{dg}{dx}$$

Intuition: Rate of change multiplies through the chain

Example:

$$f(x) = x^2, \quad g(x) = 3x + 1$$
$$\frac{d}{dx} f(g(x)) = 2(3x + 1) \cdot 3 = 6(3x + 1)$$

### Multi-Variable Chain Rule



Total Derivative (sum over all paths):

$$\frac{dy}{dx} = \frac{\partial y}{\partial z_1} \frac{\partial z_1}{\partial x} + \frac{\partial y}{\partial z_2} \frac{\partial z_2}{\partial x}$$

In neural networks: gradients flow back through ALL paths



chain\_rule\_visualization

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



## 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

## 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:

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

At the output, error attribution is straightforward

## 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!

## 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  $\rightarrow$  Last hidden  $\rightarrow \dots \rightarrow$  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

## One Training Step

### 1. Forward Pass

- Compute all activations
- Get prediction  $\hat{y}$

### 2. Compute Loss

- $\mathcal{L} = \ell(\hat{y}, y)$

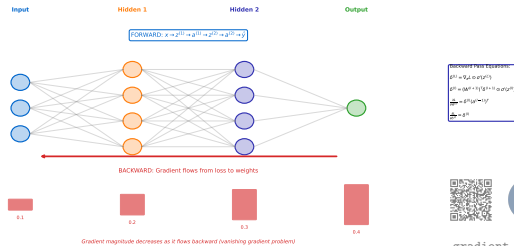
### 3. Backward Pass

- 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 Through a 3-Layer MLP



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

*“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**

## 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:

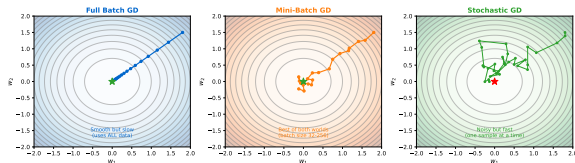
- + Stable gradient estimate
- + Deterministic updates
- + Guaranteed descent direction

## Disadvantages:

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

Compute gradient using the entire dataset

Gradient Descent Variants: Trading Off Speed vs Stability



Full Batch: Stable gradients, slow updates | Mini-Batch: Good balance, standard choice | SGD: Fast updates, noisy gradients



batch\_vs\_stochasti

## 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



## 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

Balance between efficiency and noise

## 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

# 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

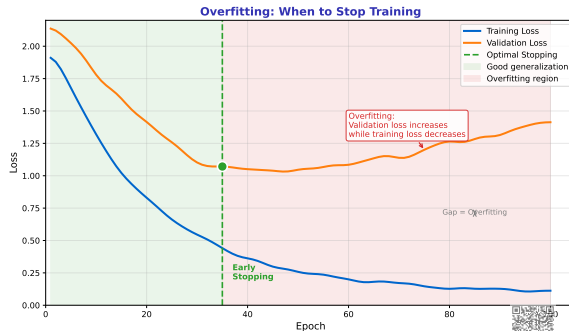
- Training loss vs. epoch
- Validation loss vs. epoch
- Learning rate schedule
- Gradient norms (debugging)

## Healthy Training:

- Both losses decrease
- Validation tracks training
- Smooth convergence

## Warning Signs:

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



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



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:

$$\begin{aligned}\delta^{(2)} &= (0.62 - 1) \cdot 0.62(1 - 0.62) \\ &= -0.089\end{aligned}$$

#### 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:  $\sigma'(z) \leq 0.25$

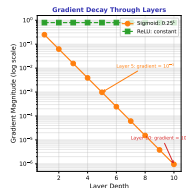
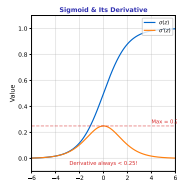
Through 10 layers: gradient  $\times 10^{-6}$

## Symptoms:

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

Deep networks: gradients can become vanishingly small

Vanishing Gradients: Why Deep Networks Were Hard to Train



## The Vanishing Gradient Problem

**Problem:** Gradients shrink exponentially with depth

**Cause:** Sigmoid/tanh derivatives are  $< 1$

**Effect:** Early layers don't learn (gradients  $\sim 0$ )

1. Use ReLU activation (gradient = 1 for  $z > 0$ )

**Solutions:**

2. Careful weight initialization (He, Xavier)

3. Batch normalization

4. Residual connections (skip connections)

5. LSTM/GRU for recurrent networks

Key insight: ReLU solved this for feedforward networks!



vanishing\_gradient\_demo

## 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:

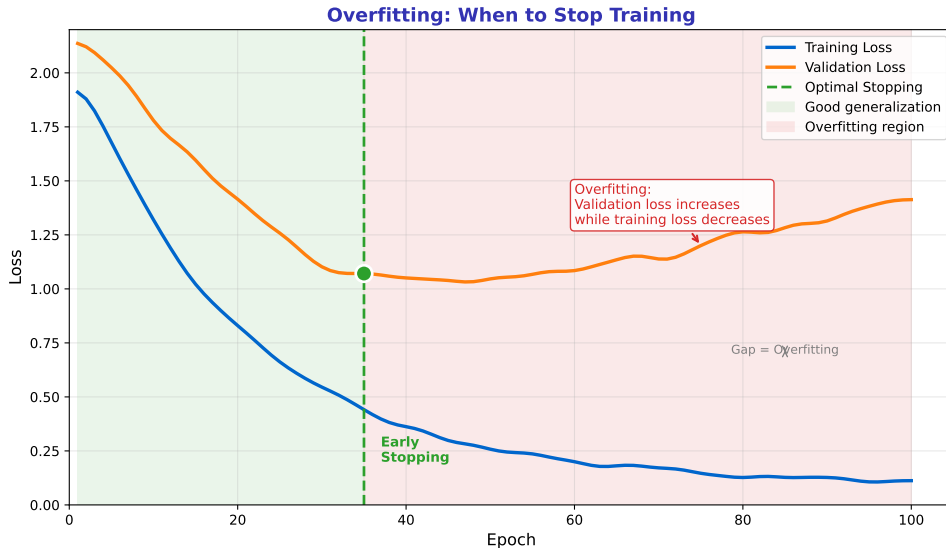
- Not enough examples
- Training set not representative
- Noise gets learned as signal

### Training Too Long:

- Model eventually memorizes
- Needs early stopping

---

When your model memorizes instead of learns



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



## 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”

## Data Limitations

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

## 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.

---

Always monitor out-of-sample performance

## 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

*“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**

## 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

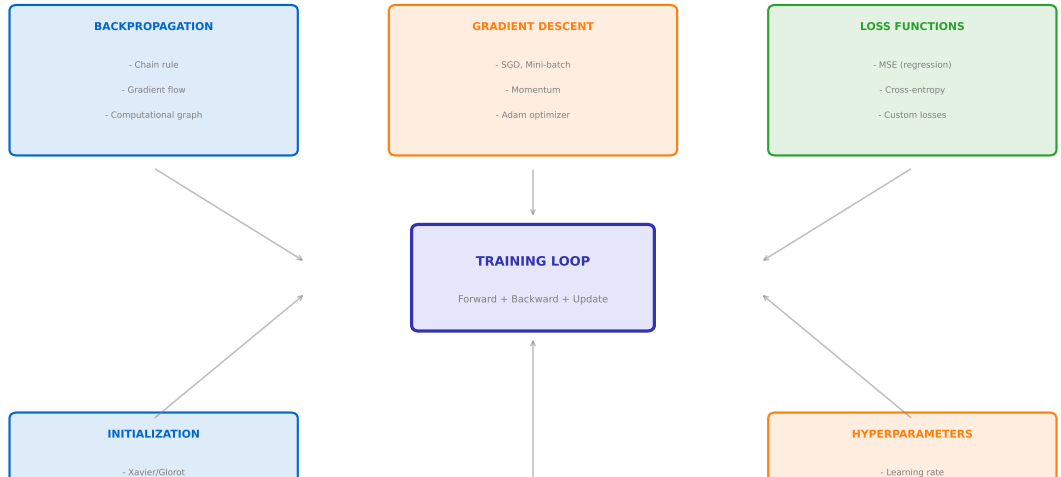
1. **Walk-Forward Validation**
  - Respect time ordering
  - Rolling windows
2. **Cross-Validation Variants**
  - Purged CV
  - Combinatorial CV
3. **Ensemble Methods**
  - Average multiple models
  - Reduce variance

*Module 4 will cover these in detail.*

---

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

## Module 3 Summary: Training Neural Networks



## 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

## 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



**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

*“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**