Lesson 35: Backpropagation

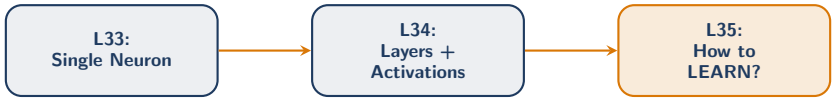
Data Science with Python – BSc Course

Data Science Program

BSc Course

45 Minutes

Previously on L34...



We can **build** networks. But weights are random.

How does the network figure out the **RIGHT** weights?

We can BUILD networks. Now we teach them to LEARN.

The Challenge

The Problem: Your network has 10,000 weights, all initialized randomly. After one prediction, it's wrong. Which weights caused the error? How much should each change?

This is the problem backpropagation solves.

After this lesson, you will be able to:

- Understand gradient descent as optimization
- Interpret loss curves and diagnose training
- Configure learning rate and its effects
- Monitor training with validation metrics

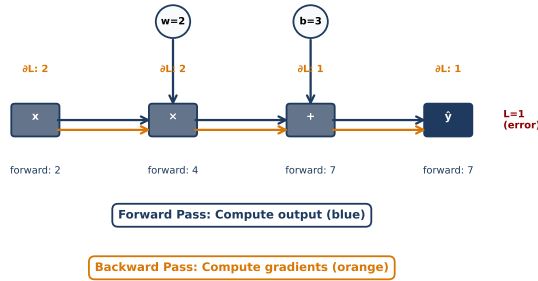
Finance Application: Training predictive models on financial data

The Big Picture: Forward and Backward

Two directions:

- **Forward:** data flows left-to-right, produces a prediction
- **Backward:** error flows right-to-left, updates weights

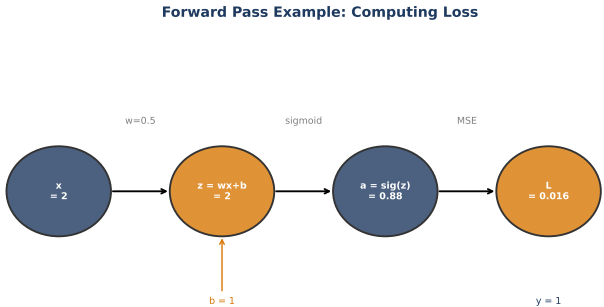
Each node asks: "How much did I contribute to the error?"



Each node asks: how much did I contribute to the error?

Step 1: The Forward Pass

Just multiply-and-add, then apply activation. Nothing more.



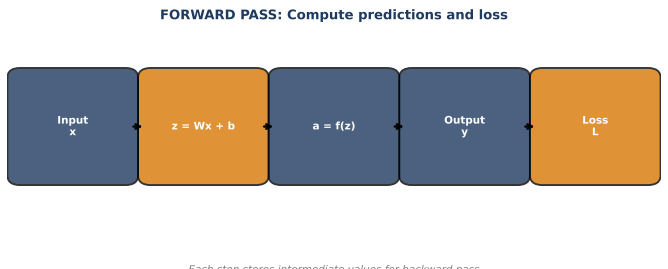
Trace the numbers: input $\times$ weight $+$ bias $\longrightarrow$ activation $\longrightarrow$ next layer

Forward Pass Concept (Detail)

Layer-by-layer computation:

- Input flows through network layer by layer
- Each layer: $z = Wx + b$, then $a = \sigma(z)$

Forward Pass: Input to Loss



Forward pass: input $\rightarrow$ hidden(s) $\rightarrow$ output = prediction

Forward Pass Implementation

forward_pass(x, weights, biases):

1. **Initialize:** `activations = [x]; a = x`
2. **For each layer:** `z = W @ a + b`
3. **Activation:** ReLU for hidden (`np.maximum(0, z)`), sigmoid for output
4. **Store:** `activations.append(a)`

Key Functions:

- `relu(z) = np.maximum(0, z)`
- `sigmoid(z) = 1 / (1 + np.exp(-z))`

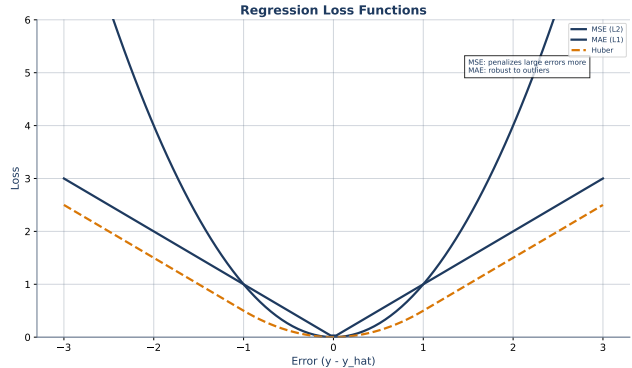
Matrix multiply + activation, layer by layer

Step 2: Measuring Error – The Loss

After the forward pass, we have a prediction. **How wrong is it?**

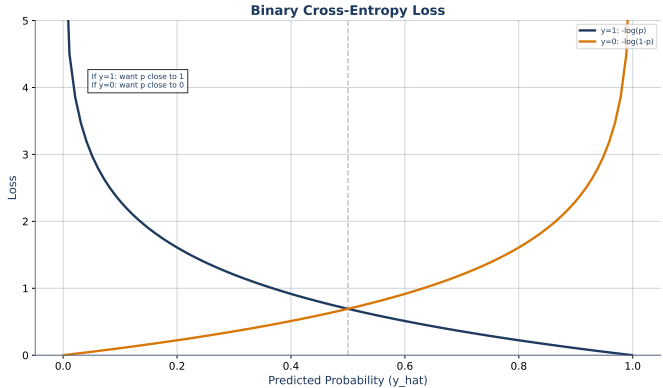
- **Regression:** MSE
- **Classification:** Cross-entropy

The loss is a **single number** that captures total wrongness.



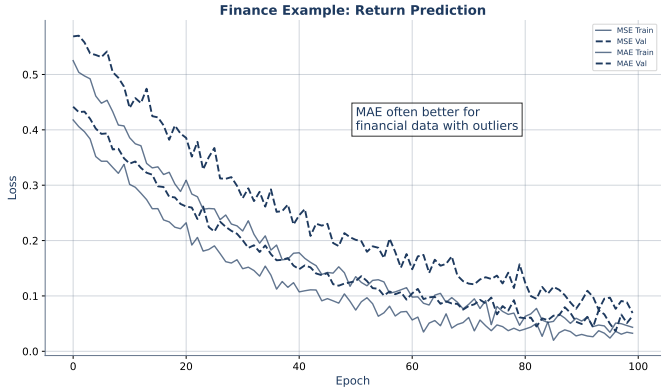
Loss = single number measuring prediction quality

Binary Cross-Entropy Loss



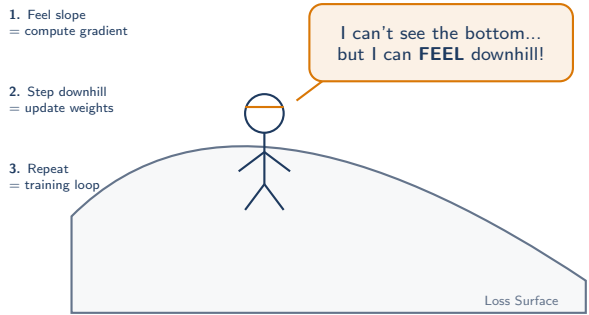
BCE: penalizes confident wrong predictions heavily

Finance: Return Prediction Loss



Predicting returns: MSE or custom loss based on business needs

The Blind Hill-Walker



Gradient descent: feel the slope, step downhill, repeat until you reach the valley

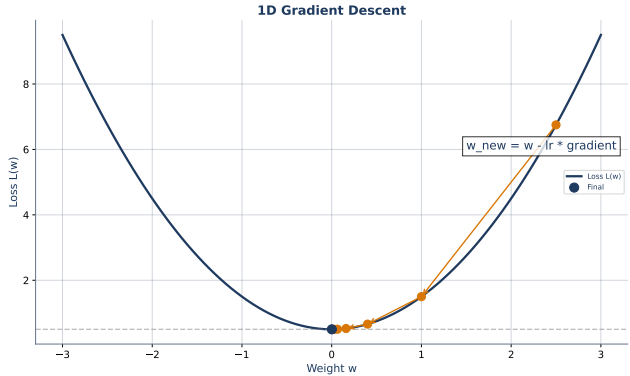
Gradient Descent: Walking Downhill

The gradient tells you which direction is **UP**. We go the **OPPOSITE** direction.

Update rule:

$$w_{\text{new}} = w_{\text{old}} - \text{step\_size} \times \text{slope}$$

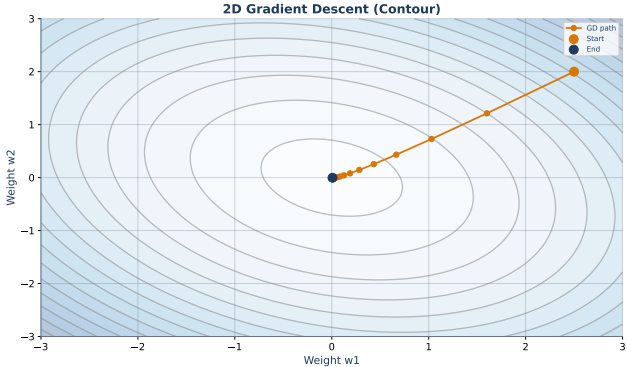
The slope tells direction; the step size tells how far.



The slope tells direction; the step size tells how far

The 2D Loss Landscape

In real networks: thousands of dimensions, but the same idea.



Always walk downhill.

Same principle in any dimension: follow the negative gradient

Discovery: Which Way Is Downhill?

The hill-walker has ONE weight.

Real networks have **thousands**. How do we find the slope for EACH weight simultaneously?

Think for 10 seconds before scrolling...

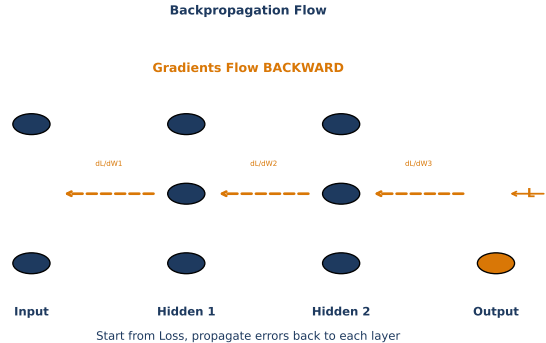
Answer: Each layer passes its share of blame backward. This is **backpropagation**.

Backpropagation = tracing blame backward through every layer

Blame-Tracing: The Backward Pass

Intuitive explanation:

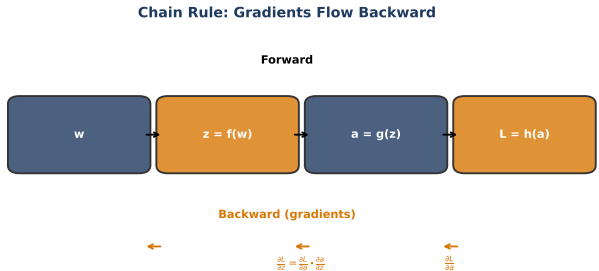
- Output layer: "My prediction was wrong by X."
- Hidden layer: "Which of my neurons contributed most?"
- Each layer passes blame backward, proportional to its contribution



Error flows backward: each weight learns its share of the blame

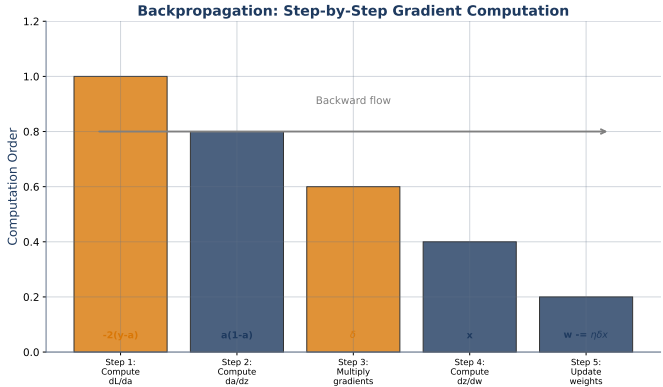
Efficiently Computing Gradients

- Chain rule: propagate error backward through layers
- Each layer's gradient depends on layers after it



Backprop = efficient gradient computation via chain rule

Backprop Step by Step



Compute output error, propagate back, accumulate weight gradients

Key Derivatives

Activation Derivatives:

- **Sigmoid:** $\sigma'(z) = \sigma(z)(1 - \sigma(z))$
- **ReLU:** $\text{relu}'(z) = 1$ if $z > 0$, else 0
- **Softmax + CE:** $\frac{\partial L}{\partial z} = \hat{y} - y$ (elegantly simple!)

Weight Gradient: For $z = Wa + b$:

- $\frac{\partial L}{\partial W} = \frac{\partial L}{\partial z} \cdot a^T$
- $\frac{\partial L}{\partial b} = \frac{\partial L}{\partial z}$
- $\frac{\partial L}{\partial a} = W^T \cdot \frac{\partial L}{\partial z}$ (passed to previous layer)

Good news: Keras/TensorFlow computes all this automatically!

The Complete Training Loop

Putting it all together:

1. **Forward pass** $\rightarrow$ prediction
2. **Loss** $\rightarrow$ how wrong
3. **Backward pass** $\rightarrow$ blame each weight
4. **Update** $\rightarrow$ adjust weights

Repeat for hundreds of **epochs**.

In Keras – four lines:

- `model.compile(optimizer='adam', loss='mse')`
- `model.fit(X, y, epochs=100, validation_split=0.2)`

You define architecture. Keras computes all the gradients automatically.

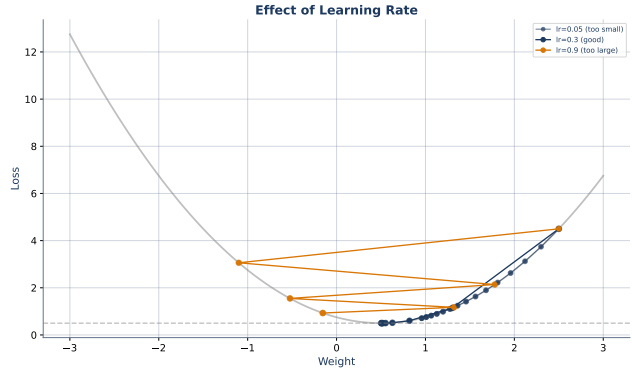
You define architecture; Keras computes all gradients automatically

The Learning Rate: Step Size Matters

How big a step do we take downhill?

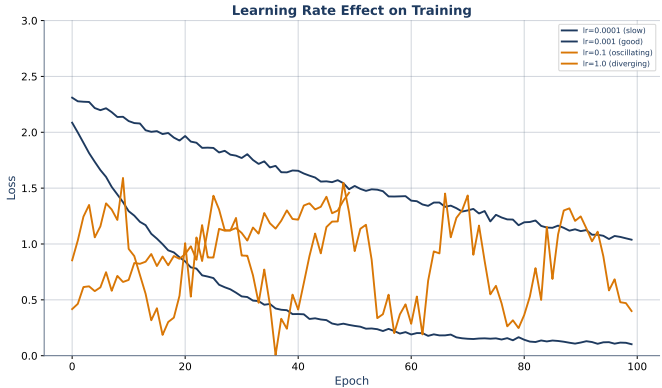
- **Too big:** overshoot the minimum
- **Too small:** takes forever
- **Just right:** steady descent

Default: `0.001` (Adam).



Start with 0.001 (Adam default). Adjust if training is unstable or too slow

Learning Rate: Visual Comparison

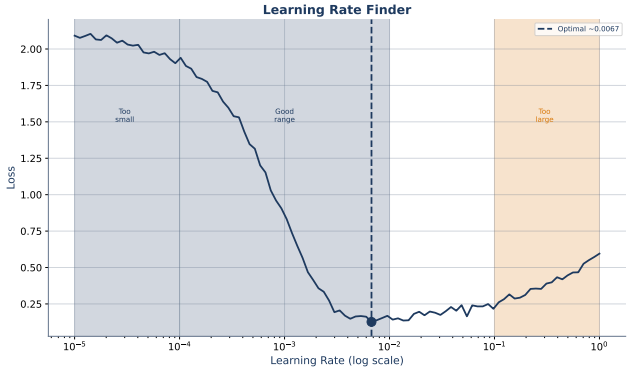


Which curve shows too-high, too-low, and just-right learning rate?

Learning Rate Finder

Systematic approach:

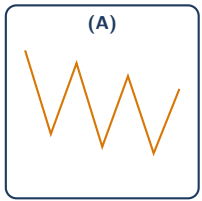
- Sweep learning rates from very small to very large
- Plot loss vs learning rate – pick steepest descent



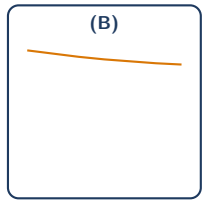
Sweep learning rates, pick value where loss decreases fastest

Checkpoint: Diagnose the Training!

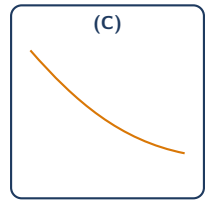
Match each curve to: too-high LR, too-low LR, just right.



Too high? Too low? Just right?



Match each to a



learning rate setting

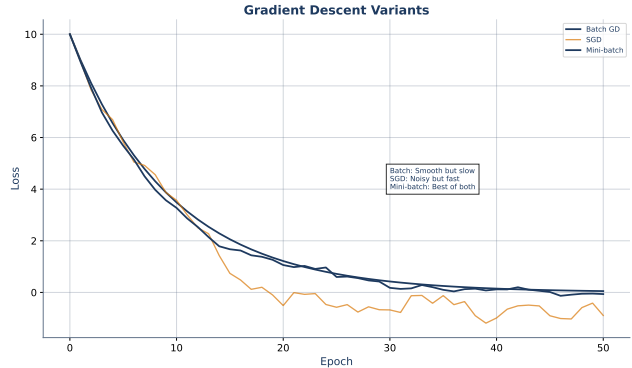
Think for 10 seconds... A = too high, B = too low, C = just right.

Diagnose before tuning: the loss curve tells you what's wrong

Optimizers: From Basic to Smart

SGD is the basic hill-walker.
Adam is a hill-walker with **MEMORY** (momentum) and **ADAPTIVE** step size.

- Adam = default in 2025
- Handles most problems well
- Rarely needs manual tuning



Adam = Adaptive Moment estimation. The modern default optimizer

Diagnosing Training: Loss Curves

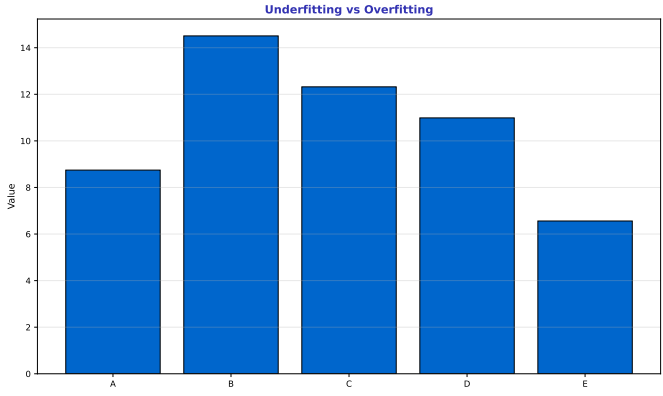
ALWAYS plot train loss AND val loss.

- **Good:** both decrease, small gap
- **Overfitting:** train decreases, val increases

This is the **MOST IMPORTANT** diagnostic tool.

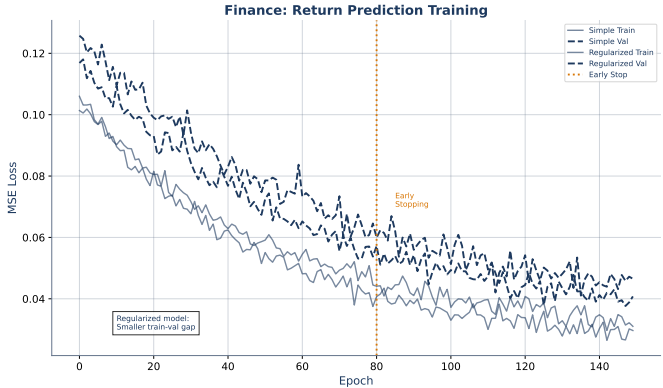
If you learn one thing: always plot train AND validation loss

Underfitting vs Overfitting



Train ↓, val ↑: overfitting. Both high: underfitting.

Finance: Return Prediction Curves



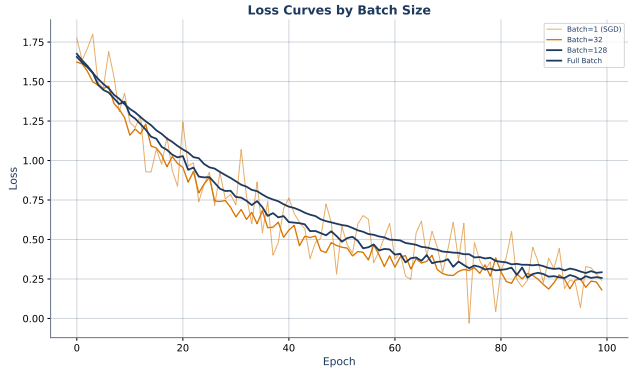
Financial data: expect noisier curves, earlier overfitting

Batch Size: How Much Data Per Step

All data per update (slow, stable) or small batches (fast, noisy)?

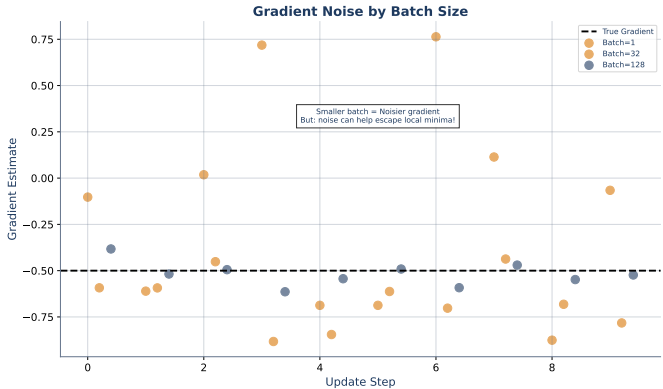
- **Batch:** all data, stable
- **Mini-batch:** 32–256, fast
- **SGD:** 1 sample, noisy

Default: mini-batch of **32**.



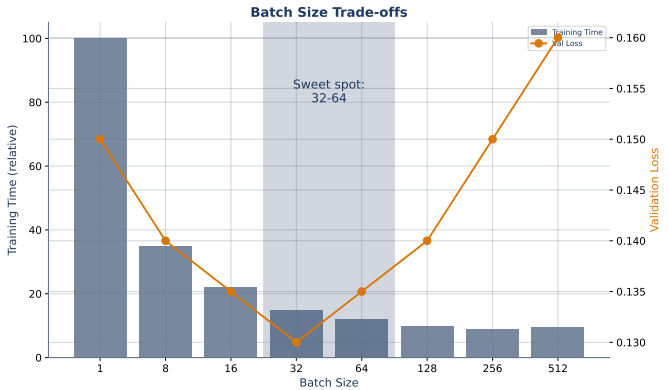
Default: mini-batch of 32. Larger batches = smoother but less regularization

Gradient Noise by Batch Size



Smaller batches = noisier gradients but can escape local minima

Batch Size Trade-offs



Memory vs noise vs convergence speed

Loss Function Cheat Sheet

Match loss to output activation (from L34):

Task	Loss Function	Output Activation
Regression	MSE / MAE	Linear (none)
Binary classification	Binary cross-entropy	Sigmoid
Multiclass	Categorical cross-entropy	Softmax

In Keras:

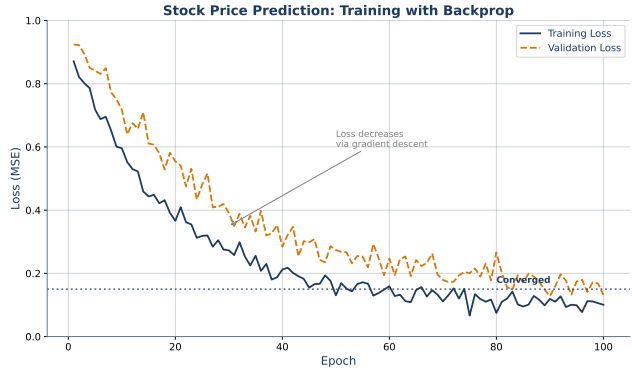
- `loss='mse'` with linear output
- `loss='binary_crossentropy'` with `sigmoid`
- `loss='sparse_categorical_crossentropy'` with `softmax`

Always match loss function to your output activation

Finance: Volatility Prediction

Regression problem:
Predict next-day volatility.
Inputs: lagged returns, volume, past volatility.

Financial data is noisy – watch for early overfitting.



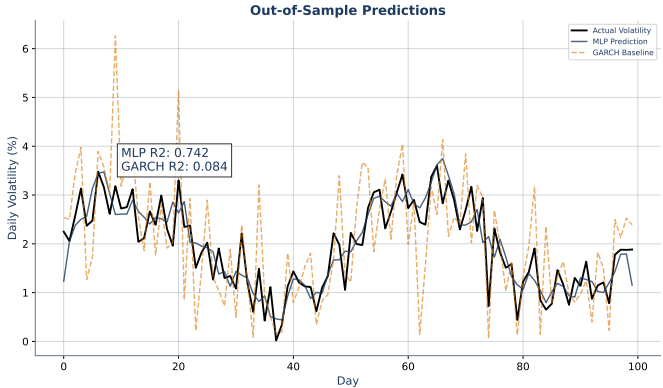
Financial ML: always expect noisier training than textbook examples

Training Curves on Financial Data

Noisy curves. Val loss diverges early. This motivates our next lesson: how to **PREVENT** the network from learning too well.

When val loss diverges early, you need regularization. That's L36.

Out-of-Sample Predictions



Predicted vs actual volatility on test data

Hands-On Exercise (25 min)

Tasks:

1. Visualize gradient descent on a parabola for 3 learning rates
2. Train Keras MLP on `make_classification`, plot loss curves
3. Try batch sizes 16, 64, 256 – compare convergence

Extension:

- Add momentum visualization and compare to vanilla gradient descent

---

Extension: Implement momentum and compare to vanilla gradient descent

Key Takeaways

1. **Forward pass:** compute prediction
2. **Loss:** measure error (one number)
3. **Backward pass:** trace blame to each weight
4. **Gradient descent:** walk downhill on the loss surface
5. **Keras does the calculus** – focus on architecture and hyperparameters

Practical defaults:

- Optimizer: Adam, learning rate: 0.001
- Batch size: 32, always plot train + val loss

Remember: forward computes, backward blames, gradient descent corrects

Preview: Too Good to Be True

Our network learns... maybe **TOO well**.

When train accuracy is 99% but test accuracy is 60%, something is very wrong.

Next lesson: three weapons against overfitting.

- Dropout
- Early stopping
- Regularization

Remember those financial data curves from slide 21? The val loss diverged early. L36 fixes that.

Next: L36 Overfitting Prevention – three weapons against memorization