Day 6: Neural Networks

Perceptron to Deep Learning – Data Science with Python (BSc)

Day 6 – Deep Learning

Today's Arc: One Neuron to Deep Learning

The Problem: Traditional ML uses hand-crafted features. Can a model **learn its own**?

After today, you will be able to:

1. Build and reason about a single **perceptron** (L33)
2. Stack neurons into an **MLP** and pick **activation functions** (L34)
3. Explain **backpropagation** and gradient descent (L35)
4. Prevent **overfitting** with dropout, early stopping, L2 (L36)

The progression:

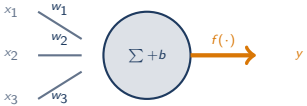
$$\text{Single Neuron (L33)} \rightarrow \text{Stacked Layers (L34)} \rightarrow \text{Learning Algorithm (L35)} \rightarrow \text{Robust Training (L36)}$$

Finance Application: foundation for deep learning models in quantitative finance

Part 1 – From Biology to Math: The Neuron

Biology

Math

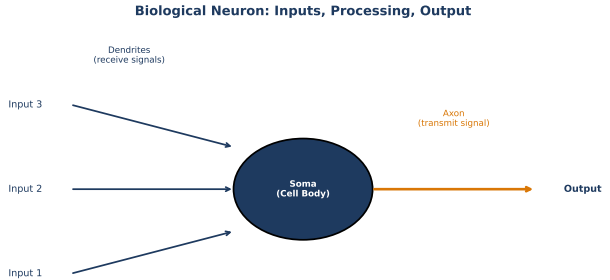


Biology	**Math**
Dendrites receive signals	Inputs $x_1, x_2, \ldots$
Synapse strength	Weights $w_1, w_2, \ldots$
Soma sums and thresholds	$z = \sum w_i x_i + b$
Axon fires or not	$y = f(z)$

We simplify radically: keep only the essential math

The Biological Inspiration

- Receives signals, sums them, fires if threshold exceeded
- We keep this idea but replace biology with linear algebra

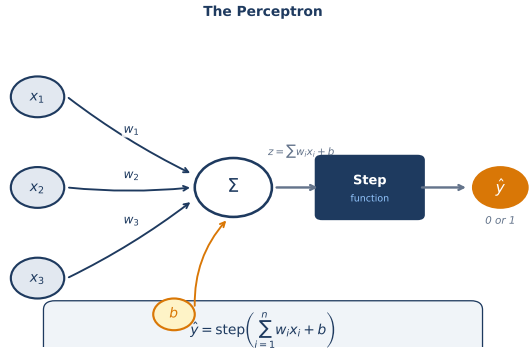


Perceptron: simplified mathematical model of biological neuron

The Perceptron: One Simple Equation

Core: $y = \text{step}\left(\sum_{i=1}^{n} w_i x_i + b\right)$

- Identical to logistic regression with step instead of sigmoid
- Weights w_i control importance; bias b shifts the threshold

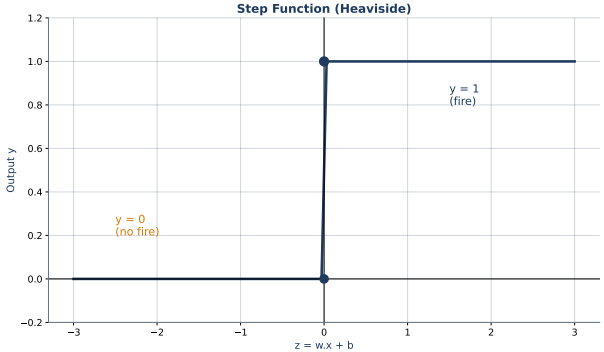


Single perceptron = logistic regression reframed as a "neuron"

Activation Function: The Step Function

The first activation function: the step function.

- An **activation function** $f(\cdot)$ decides if the neuron fires
- Step: output 1 if weighted sum > 0, else 0
- Gradient is **zero everywhere**: cannot learn via gradient descent

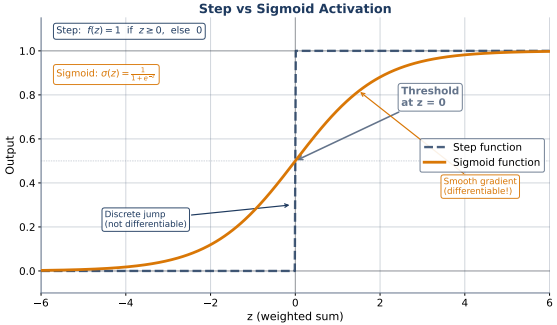


Original perceptron: output 1 if weighted sum > 0, else 0

Activation Function Example: Sigmoid

Problem: the step activation function has zero gradient everywhere.

- **Smooth:** $\sigma'(z) = \sigma(z)(1 - \sigma(z))$
- **Range** $[0, 1]$**:** interpretable as probability
- **Non-zero gradient:** enables backpropagation (Part 3)

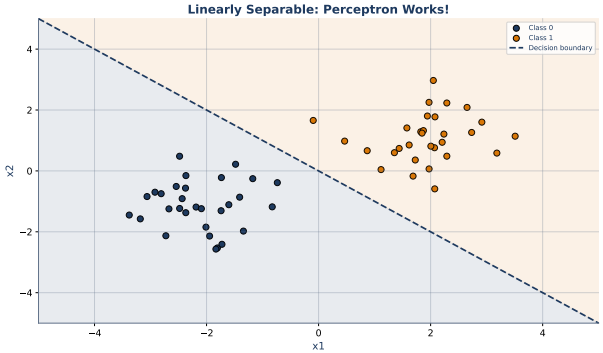


Sigmoid enables gradient-based learning; step does not

What Can a Perceptron Learn?

Linear separability is the hard constraint

- Single perceptron = one straight line (hyperplane in nD)
- Same limitation as logistic regression, because it **is** logistic regression



Perceptron = linear classifier. Can separate AND, OR but not XOR.

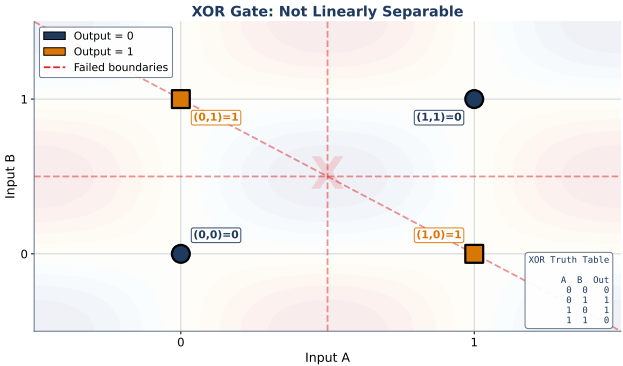
The XOR Problem: The Crisis

1969: Minsky & Papert PROVED a single perceptron CANNOT solve XOR.

XOR Truth Table:

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

No line divides the 1s from 0s.



Consequence: Funding dried up. The **AI Winter** began (1970s to 1980s).

Lesson: never dismiss an idea because of limitations in its simplest form

The Perceptron Learning Rule

Algorithm:

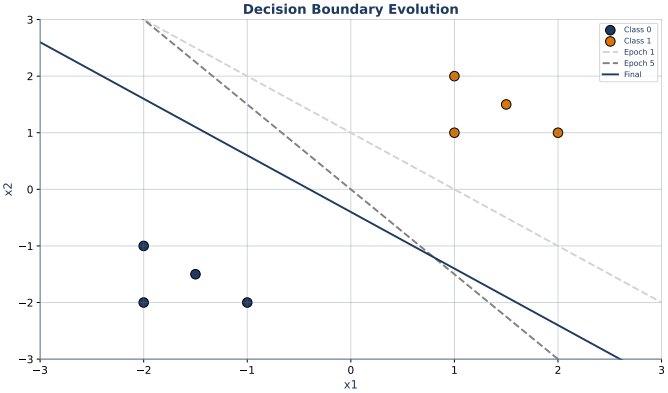
1. **Initialize:** $\mathbf{w} = \mathbf{0}$, $b = 0$
2. **For each sample** (x, y_{true}):
   - Predict: $\hat{y} = \text{step}(\mathbf{w} \cdot \mathbf{x} + b)$
   - If correct ($\hat{y} = y_{\text{true}}$): **do nothing**
   - If wrong: $\mathbf{w} \leftarrow \mathbf{w} + \eta\,(y_{\text{true}} - \hat{y})\,\mathbf{x}$
3. **Repeat** until no errors or max **epochs** (one epoch = one full pass through the training set)

Intuition: Push the decision boundary toward misclassified points.
Learning rate η**:** Typically 1.0 for perceptron (or 0.1 for finer steps).

Simple rule: only update weights when the prediction is WRONG

Watching the Boundary Learn

Decision boundary evolves over training epochs:

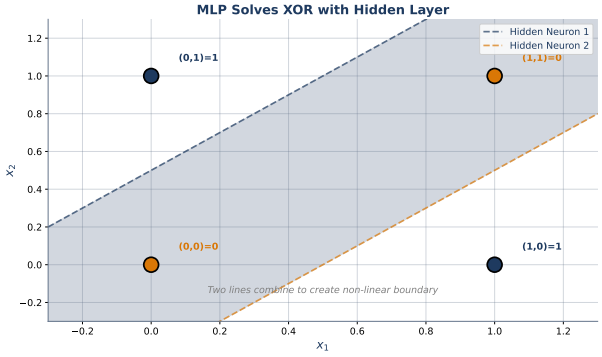


Watch the boundary adjust with each misclassified point

The Solution: Add Hidden Layers

What Minsky and Papert missed:

- Multi-layer networks **can** solve XOR
- Each hidden neuron learns a different boundary; combined: non-linear regions

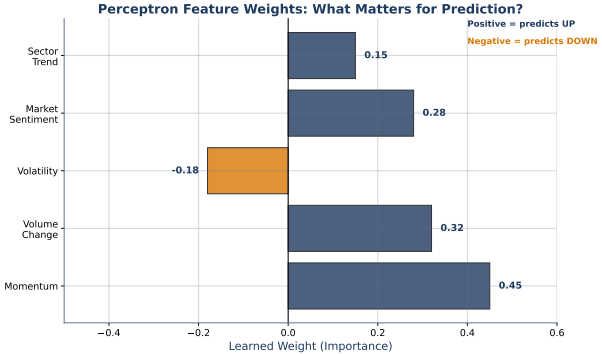


Solution: add hidden layers (multi-layer perceptron) for non-linear boundaries

Finance: Why Perceptrons Are Not Enough

Market direction prediction: up (1) or down (0) next day.

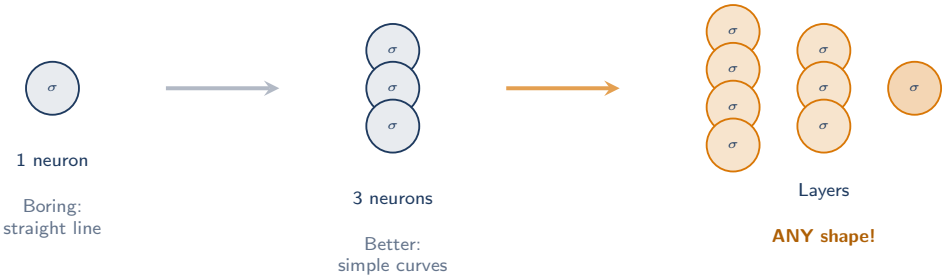
- Inputs: momentum and volatility features
- Linear boundary captures some signal, but accuracy is limited
- Non-linear regimes need deeper architectures



This motivates Part 2: MLPs, activations, and beyond

Part 2 – The Lego Insight

One brick is boring. But stacked bricks build anything.



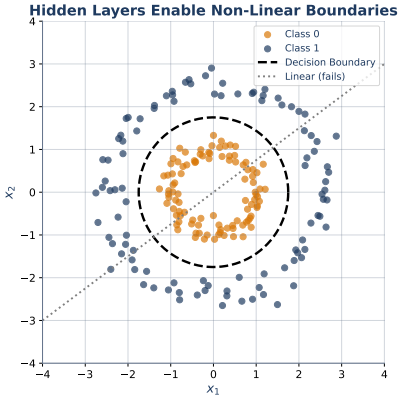
Individually simple neurons, combined in layers, can approximate **any function**.

This is the core idea behind deep learning: composing simple transformations.

MLP Architecture

Input → Hidden → Output

- **Input layer:** raw features
- **Hidden layers:** non-linear transformations
- **Output layer:** final prediction



Each connection has a weight. Each neuron has a bias. All learned during training.

Why Activation Functions Make Depth Meaningful

Every neuron makes a decision: how strongly do I respond?

Without a non-linear activation function, stacking layers is **useless**:

$$\underbrace{W_3 \cdot (W_2 \cdot (W_1 \cdot \mathbf{x}))}_{\text{3 linear layers}} = \underbrace{(W_3 W_2 W_1) \cdot \mathbf{x}}_{\text{= 1 linear layer!}}$$

The fix: add a non-linear function $g(\cdot)$ after each layer:

$$\mathbf{a} = g(\mathbf{W}\mathbf{x} + \mathbf{b})$$

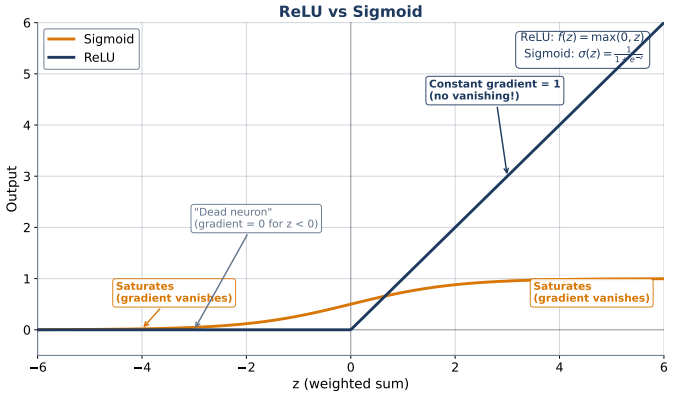
- Without g: 100 layers = 1 layer (just matrix multiplication)
- With g: each layer learns genuinely new transformations

Activation functions are what make depth meaningful. Without them, deep = shallow.

ReLU: The Modern Default Activation

ReLU$(x) = \max(0, x)$**: dead simple and effective**

- No vanishing gradient (gradient $= 1$ when active)
- Sigmoid: gradient $\to 0$ at extremes (deep nets cannot learn)
- Warning: "dying ReLU" – if always $z < 0$, neuron is permanently dead



Use ReLU for hidden layers in 99% of cases. It just works.

Output Activations: Match Task to Function

Hidden layers: always ReLU. Output layer: depends on your TASK.

Task	Output Neurons	Activation	Loss Function
Regression	1	Linear (none)	MSE
Binary classif.	1	Sigmoid	Binary CE
Multiclass (K)	K	Softmax	Categorical CE
Multi-label (K)	K	Sigmoid	Binary CE

MSE = mean squared error, $\frac{1}{n}\sum(y-\hat{y})^2$. **CE** (cross-entropy) measures how surprised the predicted probability is by the true label: confident wrong predictions are penalized heavily.

Key Rule: Loss function must match output activation!

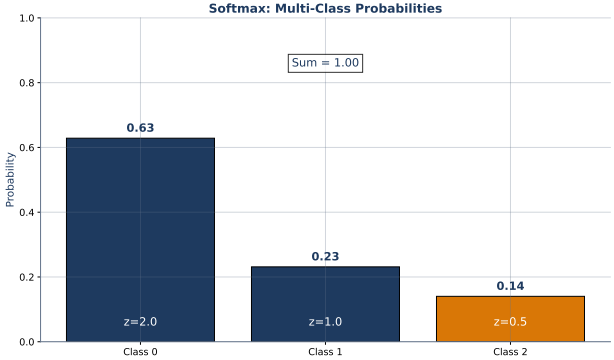
- Sigmoid: probability for ONE class (or independent labels)
- Softmax: probabilities for ALL classes (mutually exclusive, sum = 1)

Forgetting to match activation and loss is the #1 Keras beginner mistake.

Softmax for Multiclass

K output neurons for K classes, probabilities sum to 1

- Each neuron gives probability of one class
- $\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$
- Prediction: $\hat{y} = \arg\max_i \text{softmax}(z_i)$



Softmax generalizes sigmoid to multiple classes. Sigmoid is softmax with $K = 2$.

Keras: Building Networks in 5 Lines

The entire workflow:

```
1. model = Sequential([  Dense(64, 'relu', input_shape=(10,)),
       Dense(32, 'relu'),
       Dense(1, 'sigmoid')])

2. model.compile(optimizer='adam',
       loss='binary_crossentropy',
       metrics=['accuracy'])

3. model.fit(X_train, y_train, epochs=50)

4. model.evaluate(X_test, y_test)

5. model.predict(X_new)
```

That's it. Define → Compile → Fit → Evaluate → Predict.

Keras makes neural networks accessible. The hard part is choosing the architecture.

The Universal Approximation Theorem

A network with ONE hidden layer can approximate ANY continuous function.

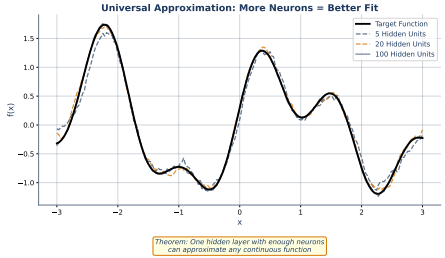
Cybenko (1989) / Hornik (1991)

What it says:

- Given enough neurons, a single hidden layer MLP can get arbitrarily close to any continuous function on a bounded domain

The catch:

- "Enough neurons" may mean **exponentially many**
- In practice, **depth beats width**: fewer total parameters needed

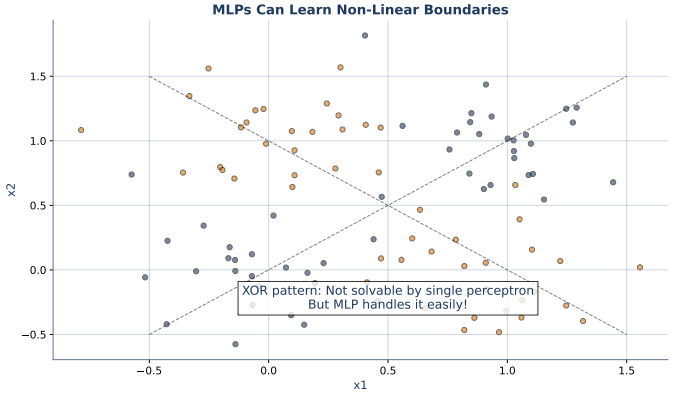


Theory: width suffices. Practice: depth is more efficient. This is why we go "deep."

Universal Approximation in Action

More neurons = better fit. XOR? No problem.

- Networks learn curves, circles, spirals: any shape
- The Lego tower is now mathematically justified

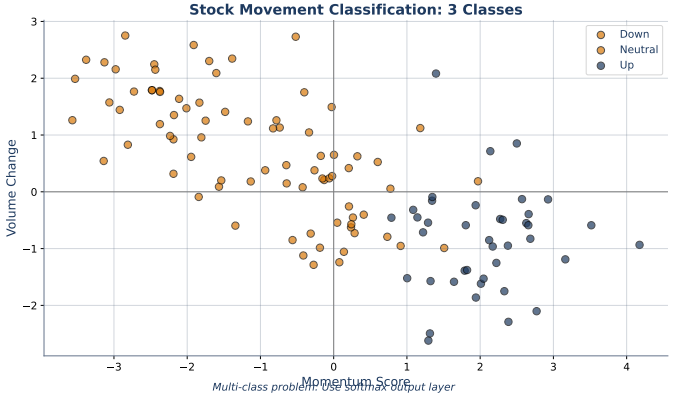


XOR, circles, spirals: all learnable with sufficient network capacity.

Finance: Market Regime Detection

3-class problem: bull / bear / sideways

- Inputs: volatility, momentum, correlation features
- Output: softmax over 3 regime classes
- Non-linear boundaries that logistic regression misses

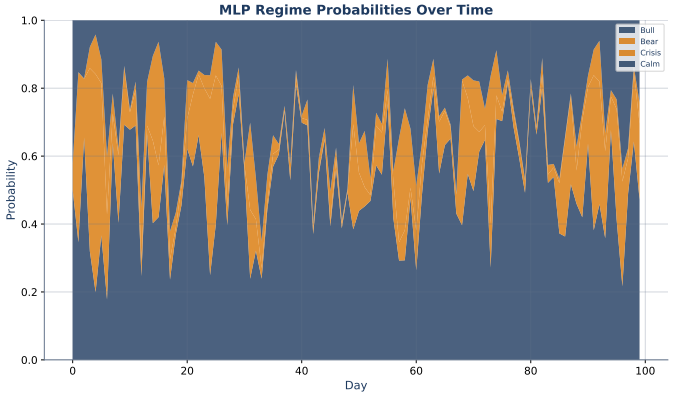


MLPs capture non-linear regime boundaries that linear models miss entirely.

Regime Probabilities

Softmax gives a probability distribution over regimes

- Not just "bull" or "bear": probabilities for each state
- Risk management: act differently when model is uncertain
- Smooth transitions between regimes (no hard switches)



Probability outputs enable nuanced trading strategies based on confidence.

Part 3 – The Challenge

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

This is the problem backpropagation solves.

After this part, 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

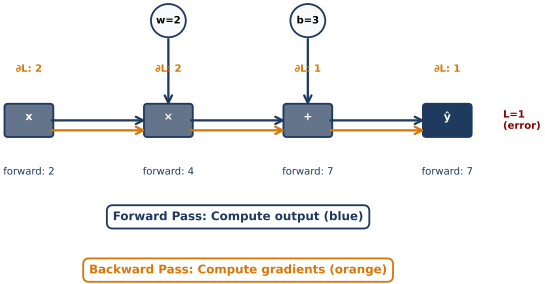
Finance Application: Training predictive models on financial data

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

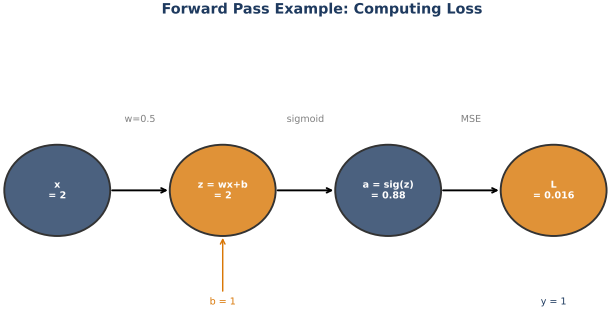
Each node asks: "How much did I contribute to the error?"



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.



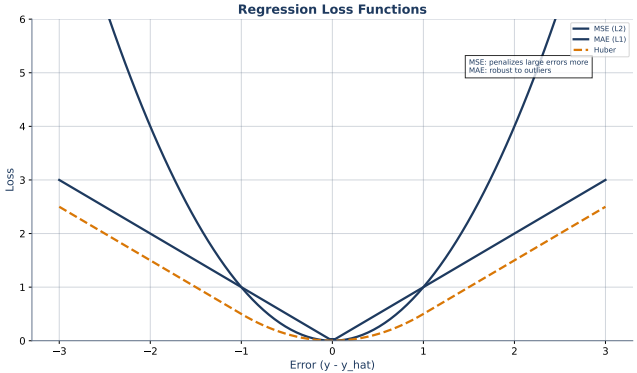
Trace the numbers: input $\times$ weight $+$ bias $\rightarrow$ activation $\rightarrow$ next layer

Step 2: Measuring Error – The Loss

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

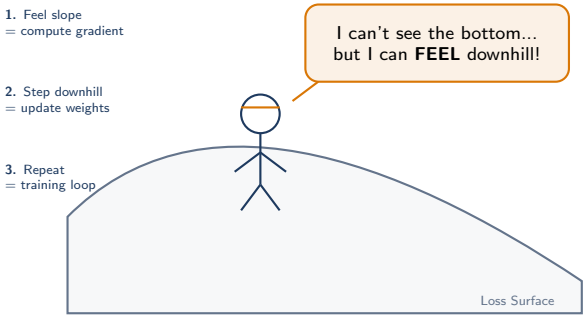
- **Regression:** MSE
- **Classification:** Cross-entropy

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



Loss = single number measuring prediction quality

The Blind Hill-Walker



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

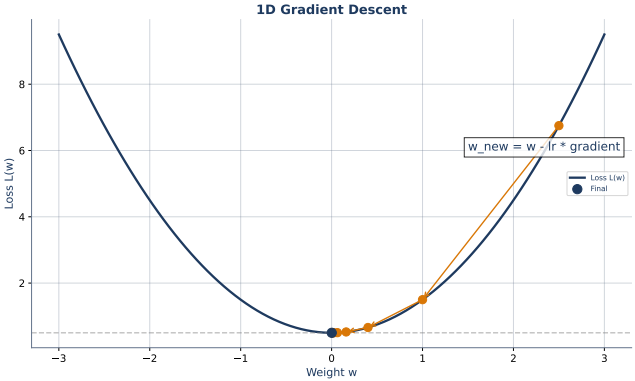
Gradient Descent: Walking Downhill

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

Update rule:

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

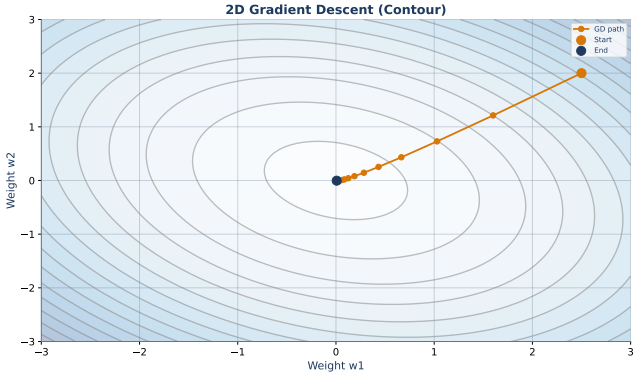
The slope tells direction; the step size tells how far.



The slope tells direction; the step size tells how far

The 2D Loss Landscape

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



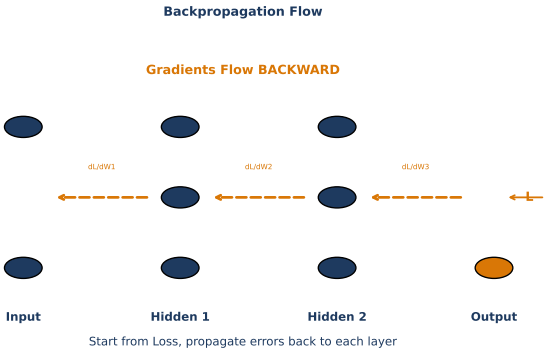
Always walk downhill.

Same principle in any dimension: follow the negative gradient

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

The Complete Training Loop

Putting it all together:

1. **Forward pass** → prediction
2. **Loss** → how wrong
3. **Backward pass** → blame each weight
4. **Update** → 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.

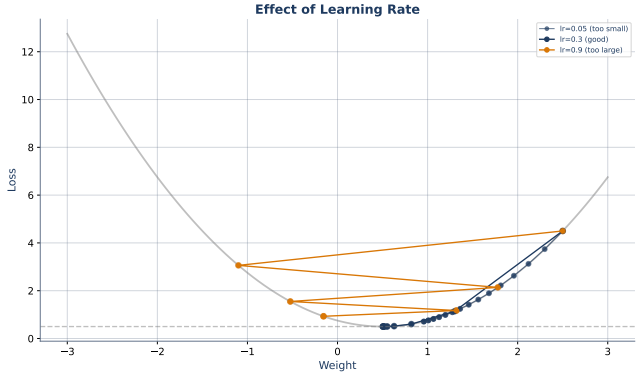
You define architecture; Keras computes all gradients automatically

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).

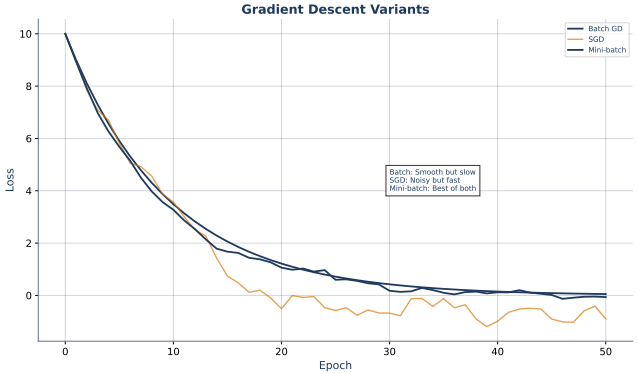


Start with 0.001 (Adam default). Adjust if training is unstable or too slow

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

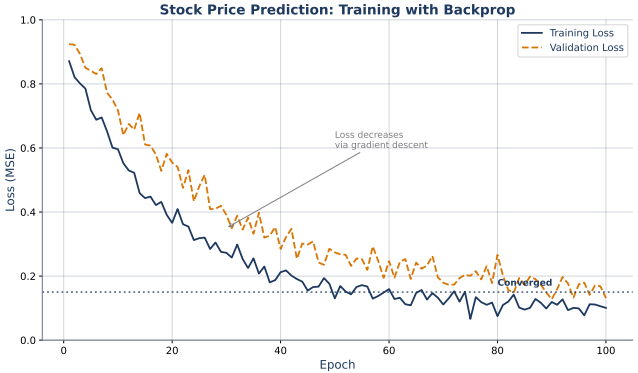
This is the **MOST IMPORTANT** diagnostic tool.

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

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 Part 4: how to **PREVENT** the network from learning too well.

When val loss diverges early, you need regularization. That is Part 4.

Part 4 – The Suspicious 99%

Scenario: Your model gets 99% accuracy on training data but 60% on new data. The CEO wants to deploy it tomorrow. What do you tell them?

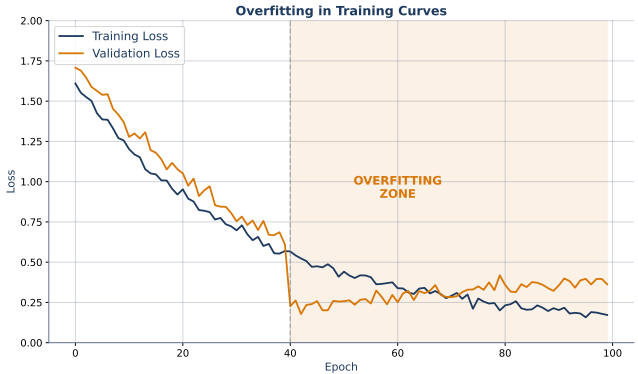
After this part, you will be able to:

- Recognize overfitting in learning curves
- Apply dropout to prevent co-adaptation
- Use early stopping to halt at the right time
- Apply L2 regularization to control weight magnitude

Is 99% training accuracy always good news?

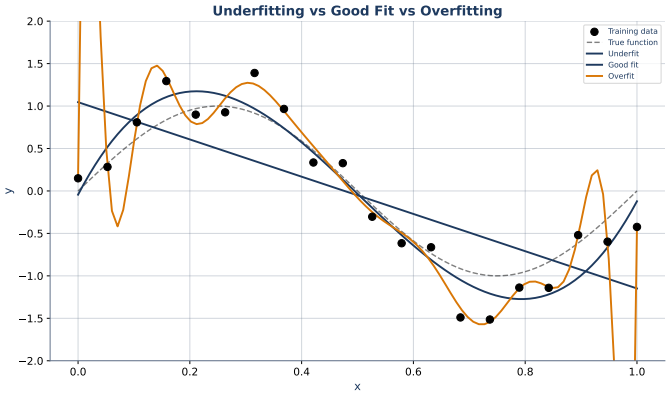
Overfitting: When Perfect Is Bad

The model is MEMORIZING the training data instead of learning general patterns.
Like a student who memorizes answers but cannot solve new problems.



Overfitting = memorizing noise instead of learning signal

Underfitting, Good Fit, Overfitting

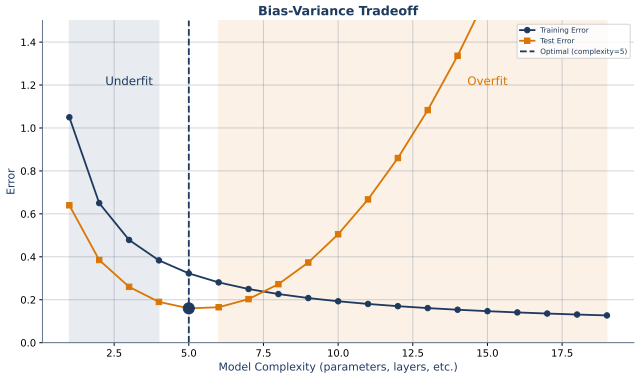


Too simple (underfitting). Just right (good fit). Too complex (overfitting).

The Goldilocks zone: complex enough to learn, simple enough to generalize

Bias-Variance: The Fundamental Tradeoff

High bias = underfitting. **High variance** = overfitting.
The sweet spot is in the middle.

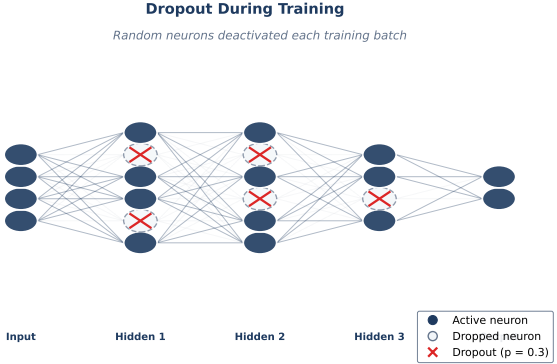


Three weapons ahead: Dropout, Early Stopping, L2 Regularization

Weapon 1: Regularization via Dropout

During training, randomly set some neurons to zero. Each batch sees a DIFFERENT sub-network.

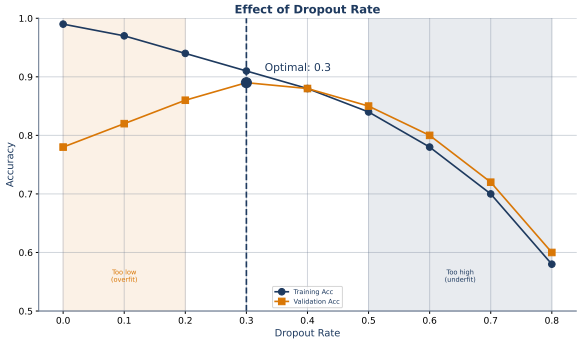
Regularization = any technique that reduces overfitting. Dropout is the first. Like studying with random pages missing: you learn core concepts, not page numbers.



Dropout forces the network to learn redundant, robust representations

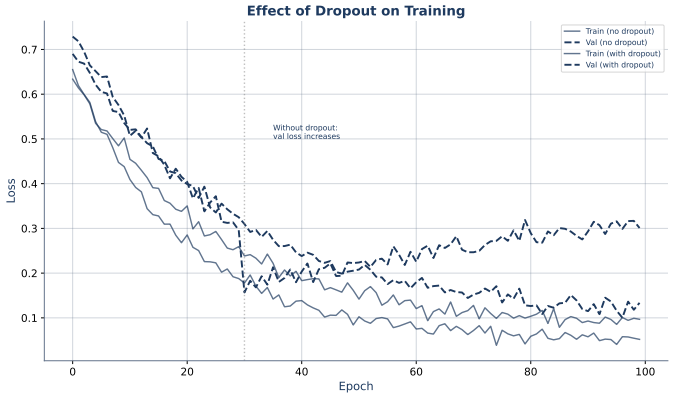
How Much Dropout?

Typical rates: 0.2 to 0.5 for hidden layers.
Keras: `Dropout(0.3)` between Dense layers. During INFERENCE, all neurons active: dropout only during training.



Keras handles train/inference modes automatically

Dropout's Effect on Curves



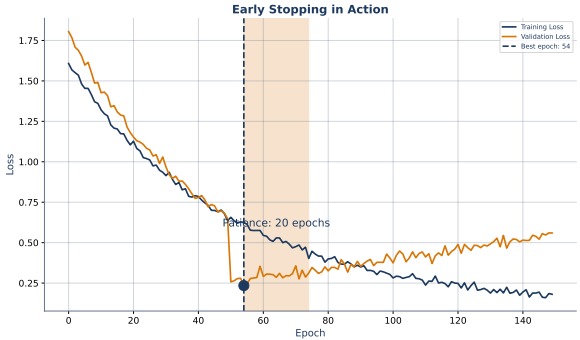
With dropout: training is slower but validation improves. The gap closes. You trade training speed for generalization. Always a good trade.

Slower training + better generalization = exactly what we want

Weapon 2: Early Stopping

Set epochs high (500), but STOP when validation loss stops improving.

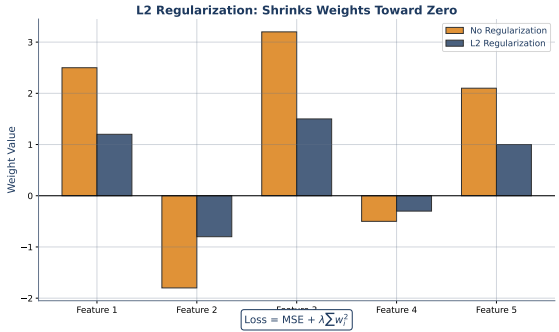
`EarlyStopping(patience=20, restore_best_weights=True)`



Best model = valley of the validation curve, not the last epoch

Weapon 3: L2 Regularization

Add penalty for large weights to loss.
Keras: `kernel_regularizer=l2(0.001)`

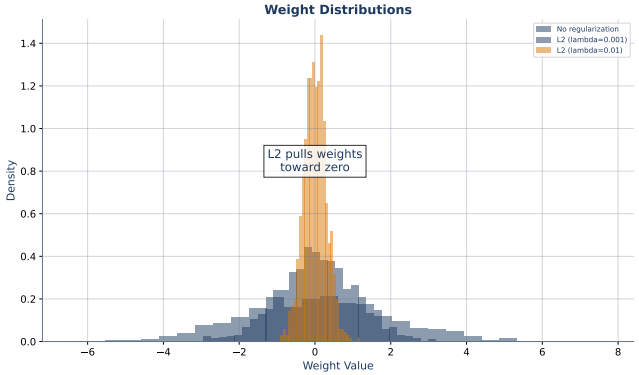


Same idea as Ridge regression: large weights = too much faith in one feature.

L2 in neural networks = Ridge regression for deep learning

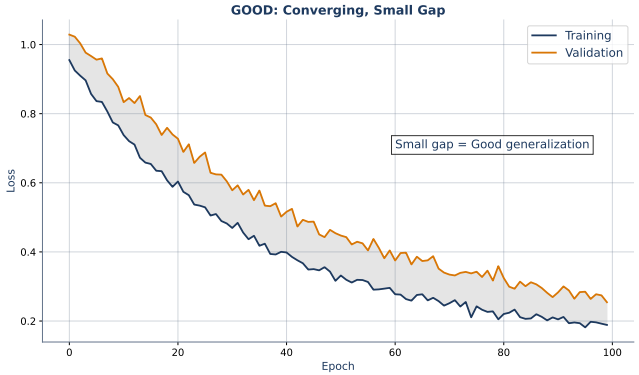
L2 Effect on Weights

Without L2: weights spread out, some extreme.
With L2: concentrated near zero.



Smaller weights → simpler model → better generalization

Good Training: Converging, Small Gap



This is what you are LOOKING for: both curves converge, small gap.

Goal: train and validation curves converge to similar low values

The Complete Keras Recipe

Put ALL weapons together in one model:
`Dense(64, 'relu')` → `Dropout(0.3)` → `Dense(32, 'relu')` → `Dropout(0.2)` → `Dense(1, 'sigmoid')`

Compile: `optimizer='adam', loss='binary_crossentropy'`

Callbacks:

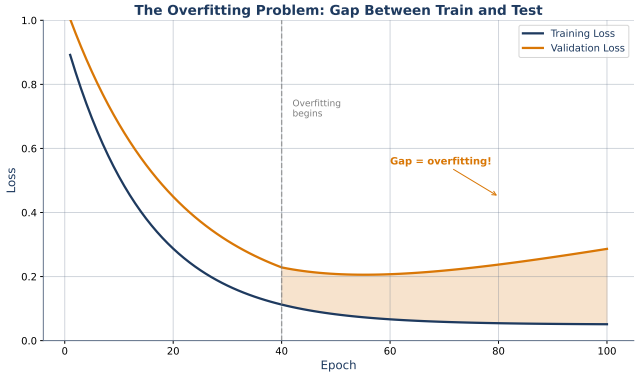
- `EarlyStopping(patience=20, restore_best_weights=True)`
- `ReduceLROnPlateau(patience=10, factor=0.5)`

Set epochs=500 and let callbacks decide when to stop.

The recipe: dropout layers + early stopping callback + L2 optional

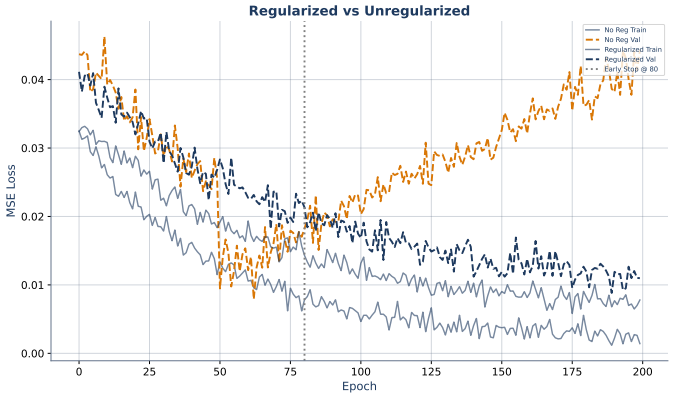
Finance: Fitting to Noise

Financial data is the HARDEST case for overfitting: noisy, non-stationary, limited samples.
Regularization is not optional: it is mandatory.



Financial ML mantra: regularize aggressively, validate relentlessly

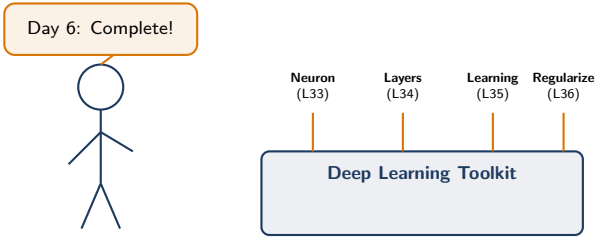
Regularized vs Unregularized



Dropout + early stopping + small network = your finance ML starter kit.

Simpler models often outperform complex ones on financial data

The Complete Deep Learning Toolkit



Always check your validation loss!

Four lessons, one complete toolkit. You are ready for deep learning.

Key Takeaways

1. **Perceptron** = logistic regression reframed as a single neuron (L33)
2. **Hidden layers + activation functions** = non-linear power (L34)
3. **Backpropagation** = trace blame backward, gradient descent corrects (L35)
4. **Overfitting** = memorizing noise; fight it with dropout, early stopping, L2 (L36)
5. **Finance:** regularize aggressively, noisy data demands it

---

One neuron to deep learning: build, stack, learn, regularize