Deep Learning

Neural Networks from Perceptrons to Modern Architectures

Lecture Companion Notes

Data Science with Python – BSc Course

Joerg Osterrieder
April 12, 2026

These notes accompany the deep learning lectures (L33–L36). They follow a problem-first structure: each section opens with a concrete challenge, builds intuition through visuals and analogies, then formalizes the concept with worked examples. Read before lecture for preparation, revisit after for deeper understanding.

Contents

1. From Linear to Nonlinear – Why We Need Deep Learning 3

2. The Perceptron – A Single Neuron 12

3. Stacking Neurons – Multilayer Perceptrons and Activation Functions 24

4. Teaching Networks – Loss Functions and Backpropagation 40

5. Why Deep Networks Overfit 55

6. Regularization for Deep Networks 63

7. Training Tricks and Pitfalls 75

8. Deep Learning in Finance – Use Cases and Limits 84

A. Solutions to Practice Problems 90

(c) Joerg Osterrieder 2026

1. From Linear to Nonlinear – Why We Need Deep Learning

Opening Problem: The Credit Risk Analyst's XOR Headache

You are a credit risk analyst at a mid-sized retail bank. For three years your logistic regression has been the bank's workhorse: give it a customer's debt-to-income ratio, employment length, credit history, and it returns a default probability. The model is calibrated, interpretable, and regulators approve of it. Life is good.

Then a new loan product arrives: buy-now-pay-later financing for gig-economy workers. You plug the old logistic regression into the new data and performance collapses. The AUC drops from 0.82 to 0.59. You check the data, fix the cleaning, add interaction terms—nothing helps. A junior colleague plots the two most informative features, "months of income stability" and "app spending velocity," and colors defaults red. The defaults form a diagonal band: defaults happen either when income is stable AND spending is high, or when income is unstable AND spending is low. Neither feature alone predicts default. The relationship is an interaction—literally, an XOR pattern.

Your logistic regression draws a single straight line. But no straight line can separate the red diagonal from the blue diagonal on a two-dimensional scatter. You need curved boundaries. You need a model that can learn that "high spending is bad for unstable workers but fine for stable ones." This section explains why classical linear models fail on such problems and introduces deep learning as the principled way to handle nonlinear interactions without writing out every interaction term by hand.

Discovery Question

Suppose you must classify four points on a 2D grid: $(0, 0)$ and $(1, 1)$ are class 0; $(0, 1)$ and $(1, 0)$ are class 1. Sketch the points. Can any straight line separate the two classes? If not, what kind of boundary would work—and how might a computer learn it without being told the shape in advance?

Linear Models and What They Cannot See

A linear classifier draws a hyperplane. In two dimensions the hyperplane is a line; in three dimensions, a flat plane; in higher dimensions, a flat slab. The classifier says: "positive on this side, negative on the other." For many problems that is enough. If a bank knows that higher income and longer employment both reduce default risk, a weighted sum with a threshold works beautifully.

Logistic regression is the most common linear classifier. It computes $z = w^\top x + b$ and passes z through a sigmoid $\sigma(z) = 1/(1 + e^{-z})$ to get a probability. The decision boundary is the set of points where $z = 0$—a hyperplane. Change the weights and the hyperplane tilts; change the bias and the hyperplane shifts; but it remains a flat slab.

Linear decision boundary: A hyperplane $\{x : w^\top x + b = 0\}$ that separates input space into two regions. Logistic regression, linear SVMs, and the single-layer perceptron all produce linear decision boundaries.

Feature interaction: A situation where the effect of one feature on the target depends on the value of another feature. "Income matters more for younger applicants than older ones" is an interaction. Linear models cannot capture interactions unless you explicitly construct the product $x_i \cdot x_j$ as a new feature.

The fundamental limit of linear models is easy to state: they can only represent linear functions of the input. If the true mapping from inputs to labels bends, twists, or contains XOR-like interactions, a linear classifier will have a non-zero error *no matter how much data you throw at it*. This is not a sample-size issue. It is a model-class issue.

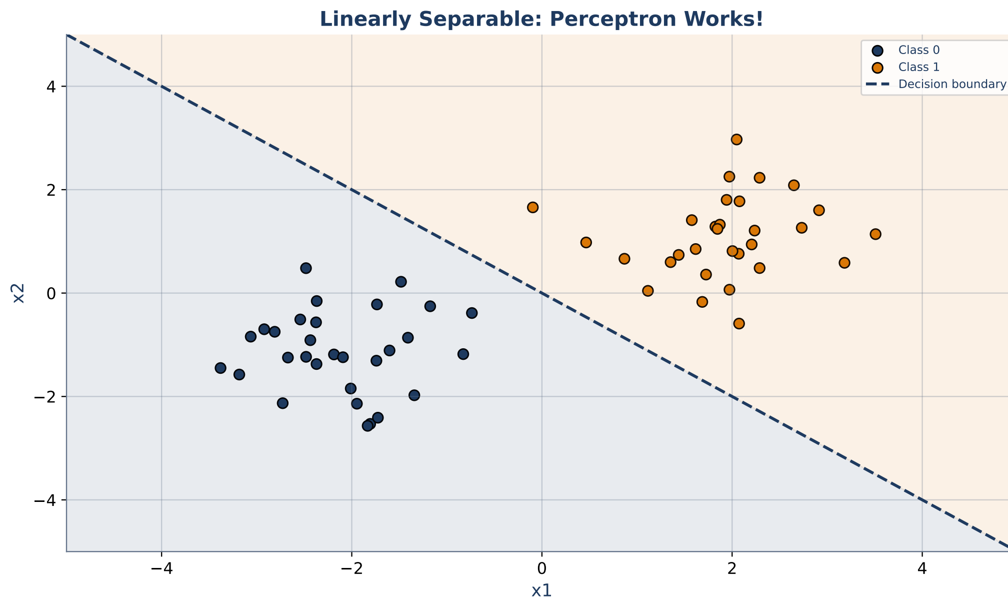

Figure 1: A linearly separable dataset. A single straight line cleanly divides the two classes. Logistic regression, linear SVM, and a single perceptron all solve this problem in a handful of iterations.

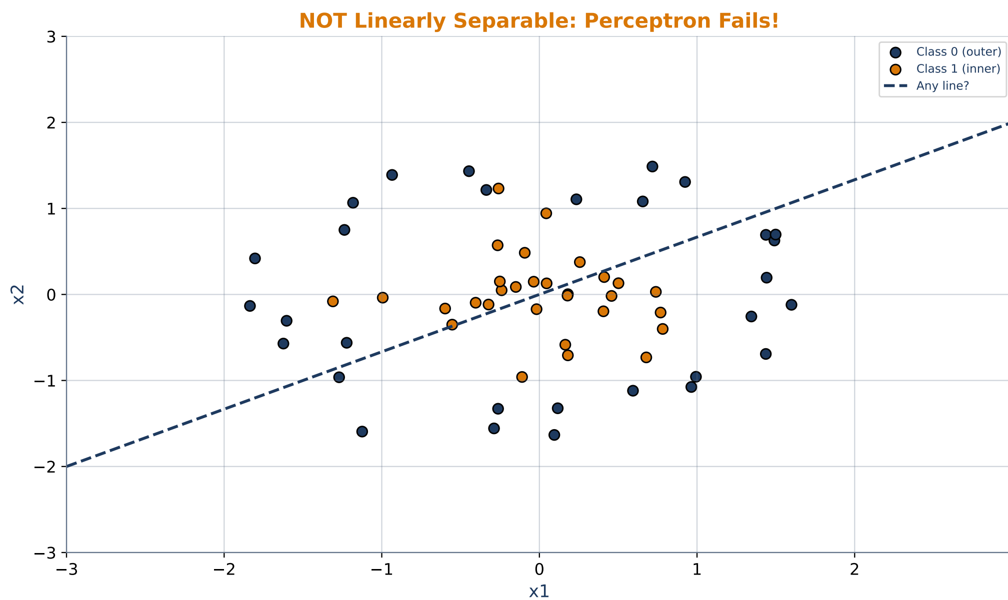

Figure 2: A non-linearly-separable dataset. No straight line separates the classes. Any linear classifier is doomed here; the hypothesis class is simply too small.

The XOR Problem: Four Points That Broke AI in 1969

The XOR function takes two binary inputs and outputs 1 when the inputs differ. It has four input combinations:

x_1	x_2	XOR(x_1, x_2)
0	0	0
0	1	1
1	0	1
1	1	0

Plot these four points in the (x_1, x_2) plane. The two 1-labels sit at opposite corners, and so do the two 0-labels. Any straight line that separates the two 1-labels from the two 0-labels must put at least one 0-labeled point on the wrong side. Try rotating the line. Try curving it—wait, linear means no curving. Try translating it. Every single line misclassifies at least one of the four points.

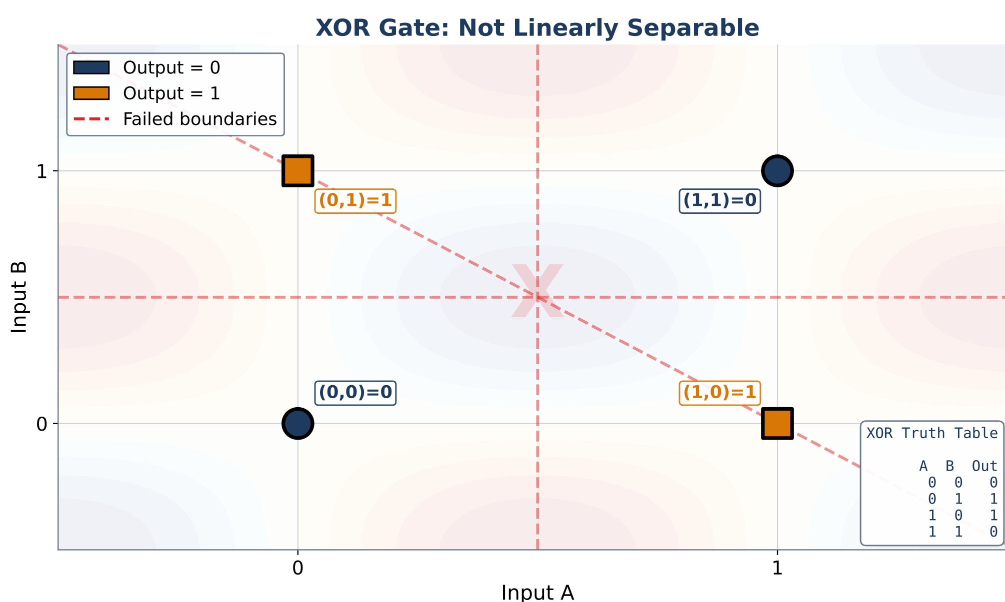

Figure 3: The XOR gate. The two classes sit at opposite corners of a square. No straight line can separate them. This four-point dataset broke the single-layer perceptron and triggered the first AI winter.

In 1969 Marvin Minsky and Seymour Papert published *Perceptrons*, a mathematically rigorous book that proved the single-layer perceptron cannot represent XOR. The proof itself takes half a page; the cultural impact took two decades. Funding for neural network research dried up almost overnight. "Connectionism" became a dirty word at many American universities. Researchers who continued working on multi-layer networks did so quietly, often publishing under different labels, until the 1986 backpropagation paper relit the fuse.

The remedy, in hindsight, is simple. Add one hidden layer with two neurons. The first hidden neuron fires when x_1 OR x_2 is active; the second fires when both are active; the output neuron takes (first hidden) AND NOT (second hidden). This two-stage computation is precisely XOR. By stacking neurons and introducing a nonlinearity between them, we recover the ability to represent functions that one layer cannot.

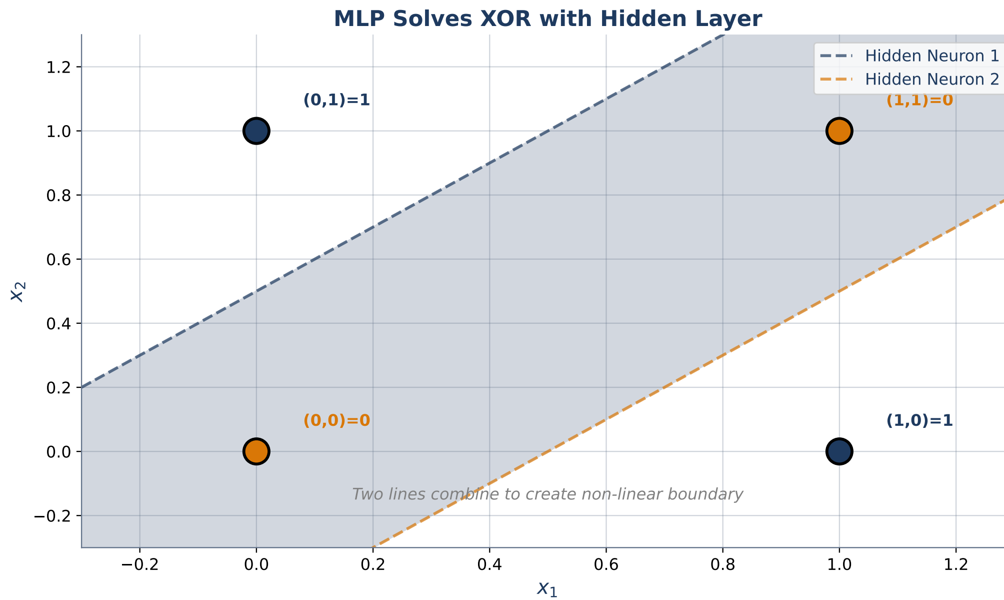

Figure 4: A two-layer network solves XOR. The hidden layer warps the input space so that classes become linearly separable in the transformed representation. Depth creates representational power.

From One Neuron to Many Layers

A single neuron computes $\hat{y} = \sigma(w^\top x + b)$, a logistic regression in disguise. Stacking neurons in a single layer with no nonlinearity between them still produces a linear model, because the composition of linear maps is linear. To gain representational power, we must apply a nonlinear activation function between layers.

Universal approximation: A property of neural networks with at least one hidden layer and a nonlinear activation: given enough hidden units, the network can approximate any continuous function on a compact domain to arbitrary accuracy. Formal statement in Hornik (1991) and earlier in Cybenko (1989).

Definition: Deep Neural Network

A deep neural network is a function $f_\theta : \mathbb{R}^{d_0} \to \mathbb{R}^{d_L}$ of the form

$$f_\theta(x) = \sigma_L(W_L \sigma_{L-1}(W_{L-1} \cdots \sigma_1(W_1 x + b_1) \cdots + b_{L-1}) + b_L)$$

where $L \geq 2$ is the number of layers, $W_\ell \in \mathbb{R}^{d_\ell \times d_{\ell-1}}$ and $b_\ell \in \mathbb{R}^{d_\ell}$ are learned parameters, and each σ_ℓ is a nonlinear activation (ReLU, sigmoid, tanh, etc.). "Deep" informally means $L \geq 2$; modern networks often have L in the tens or hundreds.

Key Formula: One Layer of Computation

A single hidden layer transforms its input $h^{(\ell-1)} \in \mathbb{R}^{d_{\ell-1}}$ into a new vector $h^{(\ell)} \in \mathbb{R}^{d_\ell}$:

$$h^{(\ell)} = \sigma\Big(W_\ell h^{(\ell-1)} + b_\ell\Big)$$

where:

- W_ℓ is a $d_\ell \times d_{\ell-1}$ weight matrix—each row is the weight vector of one neuron
- b_ℓ is a d_ℓ-dimensional bias vector—one bias per neuron
- σ is a nonlinearity applied element-wise

Plain English: Multiply the input by a matrix, add a bias, then bend the result with a nonlinear function. Repeat.

The upgrade path from classical ML to deep learning: add hidden layers, add nonlinearity, regain the ability to fit interactions.

Historical Background: Minsky, Papert, and the First AI Winter (1969)

In 1969 Marvin Minsky (MIT) and Seymour Papert (also MIT) published *Perceptrons: An Introduction to Computational Geometry*. The book analyzed what single-layer perceptrons can and cannot compute. Their most famous result was a proof that the perceptron cannot represent XOR, together with a broader argument that perceptrons cannot compute "global" properties of geometric figures (like connectedness) without prohibitively many units.

The book was mathematically correct but rhetorically lethal. Minsky and Papert were influential figures, and their pessimistic framing was read (perhaps too broadly) as a death knell for neural network research. Research funding shifted to symbolic AI—expert systems, logic programming, knowledge representation. Neural networks entered a two-decade hibernation known as the first AI winter.

Multi-layer networks could have solved XOR from day one—Minsky and Papert knew this. What was missing in 1969 was not the architecture but an efficient training algorithm. That arrived in 1986 when Rumelhart, Hinton, and Williams rediscovered and popularized backpropagation (earlier formulations date to Werbos 1974 and Kelley/Bryson in the 1960s control-theory literature). With backpropagation, deep networks could finally be trained. The ice age ended; connectionism returned.

Where Deep Learning Fits in the ML Toolbox

Deep learning is not a replacement for classical machine learning; it is one tool among many. Figure 6 shows where it sits. Linear and tree-based methods still dominate tabular data, small-sample problems, and any setting where interpretability is paramount. Deep learning pulls ahead on unstructured data—images, text, audio, sequences—and on tabular problems with massive training data and strong interactions.

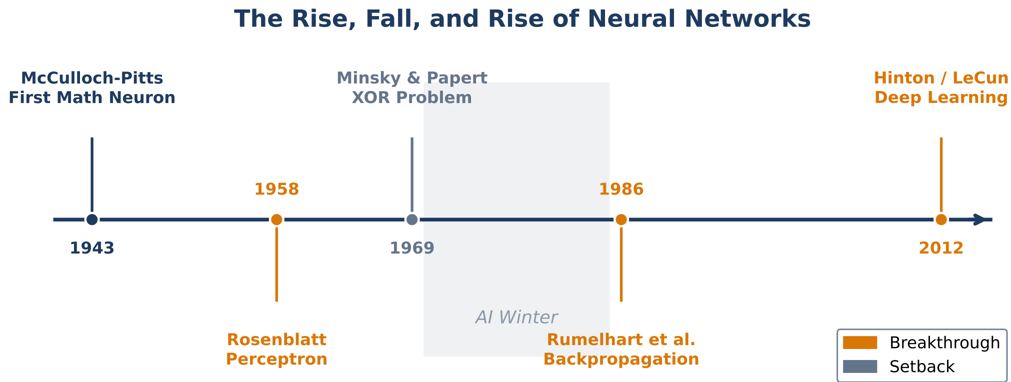

Figure 5: A timeline of neural network history. Two winters and three springs: the 1943 McCulloch–Pitts neuron, the 1958 perceptron, the 1969 Minsky–Papert critique, the 1986 backprop revival, and the 2012 ImageNet breakthrough that launched modern deep learning.

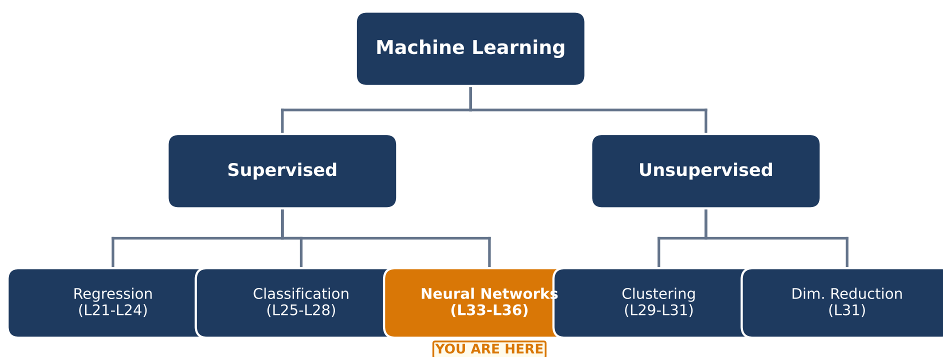

Figure 6: Where deep learning sits inside the broader ML family tree. It is one branch of supervised learning alongside linear models, trees, and kernel methods—not a strict superset.

Common Misconceptions about "Why We Need Deep Learning"

(1) "Deep learning always beats linear models." False. On tabular data with fewer than 10,000 rows, gradient boosting usually matches or beats a neural network with substantially less tuning. Deep learning shines on high-dimensional unstructured inputs.
(2) "Adding layers always helps." Stacking layers without nonlinearity is equivalent to one layer—composition of linear maps is linear. You need a nonlinear activation between every pair of layers.
(3) "Minsky and Papert killed neural networks forever." They pointed out a real limitation of single-layer perceptrons, which was correct. Multi-layer networks were never disproved; they just lacked an efficient training procedure until 1986.

Worked Examples

Worked Example 1: Why Logistic Regression Fails on XOR

Consider the four XOR points: $(0,0) \to 0$, $(0,1) \to 1$, $(1,0) \to 1$, $(1,1) \to 0$. A logistic regression model is $\hat{y} = \sigma(w_1x_1 + w_2x_2 + b)$.
Suppose we want $\hat{y} > 0.5$ exactly for the two points labeled 1. That requires:

- $w_1(0) + w_2(1) + b > 0 \Rightarrow w_2 + b > 0$
- $w_1(1) + w_2(0) + b > 0 \Rightarrow w_1 + b > 0$
- $w_1(0) + w_2(0) + b \leq 0 \Rightarrow b \leq 0$
- $w_1(1) + w_2(1) + b \leq 0 \Rightarrow w_1 + w_2 + b \leq 0$

Add the first two inequalities: $w_1 + w_2 + 2b > 0$, so $w_1 + w_2 > -2b \geq 0$ (since $b \leq 0$). Substituting into the fourth: $w_1 + w_2 + b > 0 + b = b$. But we needed $w_1 + w_2 + b \leq 0$, which means $b \geq w_1 + w_2 > 0$. Contradiction with $b \leq 0$. No choice of (w_1, w_2, b) satisfies all four constraints. Logistic regression cannot represent XOR.

Worked Example 2: A Hidden Layer Solves It

Introduce two hidden units h_1, h_2 with ReLU activations:

$$h_1 = \max(0, x_1 + x_2 - 0.5), \qquad h_2 = \max(0, x_1 + x_2 - 1.5).$$

Output: $\hat{y} = \sigma(h_1 - h_2 - 0.5)$.
Evaluate on the four XOR inputs:

- $(0,0)$: $h_1 = \max(0, -0.5) = 0$, $h_2 = \max(0, -1.5) = 0$, $\hat{y} = \sigma(-0.5) \approx 0.38 \to 0$. ✓
- $(0,1)$: $h_1 = \max(0, 0.5) = 0.5$, $h_2 = \max(0, -0.5) = 0$, $\hat{y} = \sigma(0) = 0.5$. Borderline.
- $(1,0)$: same as $(0,1)$ by symmetry. Borderline.
- $(1,1)$: $h_1 = \max(0, 1.5) = 1.5$, $h_2 = \max(0, 0.5) = 0.5$, $\hat{y} = \sigma(0.5) \approx 0.62$. Wrong!

Our manual weights are not perfect, but the point is that *some* setting of the six parameters (weights, biases) can solve XOR exactly. Gradient descent will find one. The hidden layer creates two new features (h_1, h_2) that make the problem linearly separable in the transformed space. This is the key insight of deep learning: depth builds representations.

Problem 1.1 (Easy) ⋆

Explain in one sentence each: (a) why linear regression cannot fit $y = x^2$; (b) why a logistic regression cannot separate XOR; (c) what property of neural networks lets them handle both.
Solution: see Appendix.

Problem 1.2 (Easy) ⋆

Given three features x_1, x_2, x_3, how many *pairwise* interaction terms $x_i x_j$ can you form? If a linear model needs all pairwise and all triple-wise interaction terms as explicit features to fit a nonlinear pattern, how many features does that become? Why does this motivate deep learning as d grows?
Solution: see Appendix.

Problem 1.3 (Medium) ⋆⋆

A dataset has 20 binary features. You suspect that the label depends on interactions involving any three of them simultaneously (e.g., $x_3 \wedge x_7 \wedge x_{12}$). How many such three-way interaction terms exist? Compare this count to the number of parameters in a fully connected neural network with 20 inputs, one hidden layer of 16 ReLU neurons, and one output neuron. Which is more economical?
Solution: see Appendix.

Problem 1.4 (Medium) ⋆⋆

A junior analyst stacks five linear layers $W_5(W_4(W_3(W_2(W_1x))))$ with no activations between them. Show that this is equivalent to a single linear transformation Wx. What single matrix W does the five-layer stack compute? What does this tell you about the role of nonlinearity in deep networks?
Solution: see Appendix.

Problem 1.5 (Hard) ⋆⋆⋆

State the universal approximation theorem (Cybenko 1989 or Hornik 1991) informally. Then explain, in your own words, why universal approximation does *not* imply that deep networks always outperform shallow networks in practice. Mention at least two practical reasons (e.g., number of parameters needed, optimization difficulty).
Solution: see Appendix.

Connecting Forward

We now know why linear models fail on problems with interactions—and we have seen the remedy: add hidden layers and nonlinear activations. Section 2 zooms in on the basic building block, the single neuron (perceptron). We will see it as a thin wrapper around logistic regression, understand its geometric interpretation as a hyperplane, and recover the 1958 learning rule that Rosenblatt originally proposed. That sets up Section 3, where we stack neurons into multilayer perceptrons that can represent anything linear models cannot.

Key Takeaway: Linear models cannot represent interactions; deep networks regain this ability

by stacking layers with nonlinear activations—the XOR problem is the smallest example of this fundamental distinction.

(c) Joerg Osterrieder 2026

2. The Perceptron – A Single Neuron

Opening Problem: Rosenblatt's Photoperceptron and Modern Credit Scoring

In 1958 Frank Rosenblatt wired 400 cadmium-sulfide photocells to a bank of potentiometers and built the Mark I Perceptron at Cornell Aeronautical Laboratory. The machine filled a room. Its job: look at a 20×20 pixel image of a printed letter and decide whether it was an A or not. The New York Times ran a story on July 8, 1958, with the headline "New Navy Device Learns by Doing"—and predicted machines that could "walk, talk, see, write, reproduce itself and be conscious of its existence." The hype was spectacular. The actual machine could recognize a single letter with moderate accuracy.

Sixty-eight years later a credit-card issuer needs to decide, for every new applicant, whether to approve or decline. The features are different—debt-to-income, months at current address, credit utilization—but the math is identical to Rosenblatt's perceptron. Multiply each feature by a weight, sum them, add a bias, pass through an activation. If the activation clears a threshold, approve; otherwise decline. The same formula that classified typed letters in 1958 classifies loan applications in 2026. This section studies that formula in detail, derives its learning rule, and analyzes its geometric interpretation.

Discovery Question

A biological neuron receives electrical signals from many dendrites, sums them in the cell body, and fires an output pulse down the axon if the sum exceeds a threshold. If we had to mimic this with math, what are the minimal ingredients? How few parameters can we get away with?

Biology as Inspiration, Not Blueprint

A real neuron has dendrites, a soma, an axon, and synaptic clefts. Dendrites receive neurotransmitters from upstream neurons. The soma integrates the incoming electrochemical signal. When the integrated potential exceeds a threshold, an action potential propagates down the axon and releases neurotransmitters into the next cleft. Real neurons have refractory periods, dynamic synaptic weights, and firing rates that depend on timing.

An artificial neuron (perceptron) keeps three ideas and discards the rest:

1. **Weighted inputs.** Each incoming feature x_i is multiplied by a weight w_i that represents how much this neuron cares about that feature.
2. **Aggregation.** The weighted inputs are summed. A bias b shifts the baseline up or down.
3. **Threshold/activation.** The sum is passed through a function σ that squashes or gates the output.

Everything else—timing, refractory periods, three-dimensional geometry of dendrites—is thrown away. That is why we say a perceptron is *inspired by* biology, not a model of it.

The Perceptron Formula

A perceptron with n inputs is parameterized by a weight vector $w = (w_1, \dots, w_n)$ and a scalar bias b. Given an input $x = (x_1, \dots, x_n)$, it computes:

$$z = w_1x_1 + w_2x_2 + \cdots + w_nx_n + b = w^\top x + b.$$

Then it applies an activation σ:

$$\hat{y} = \sigma(z).$$

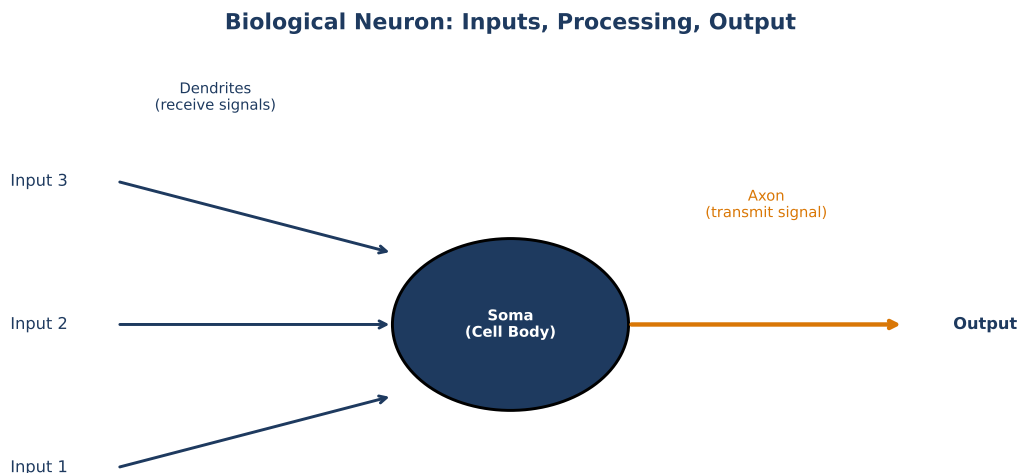

Figure 7: A biological neuron. Dendrites receive signals, the soma integrates them, the axon transmits the output. Real neurons are electrochemical marvels; artificial neurons borrow only the aggregate-then-threshold structure.

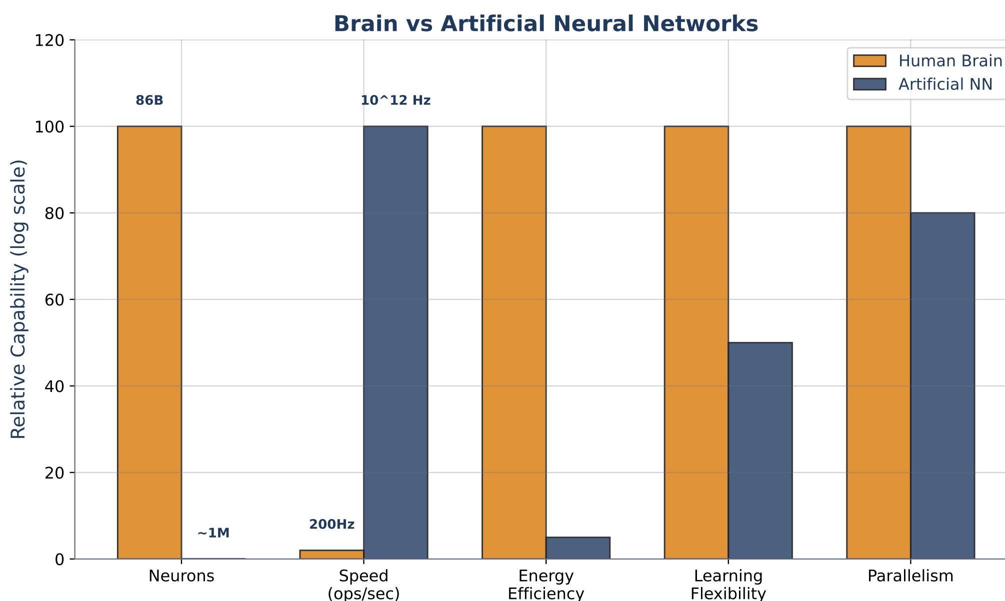

Figure 8: Brain vs. artificial neural network. The brain has ∼86 billion neurons with ∼10,000 synapses each. A credit-scoring MLP has maybe 10,000 parameters total. The analogy is loose but the computational motif is shared.

In the original 1958 perceptron, σ was the step (Heaviside) function: $\sigma(z) = 1$ if $z \geq 0$, else 0. Modern perceptrons almost always use a smooth activation like sigmoid or ReLU, because smooth activations allow gradient-based learning.

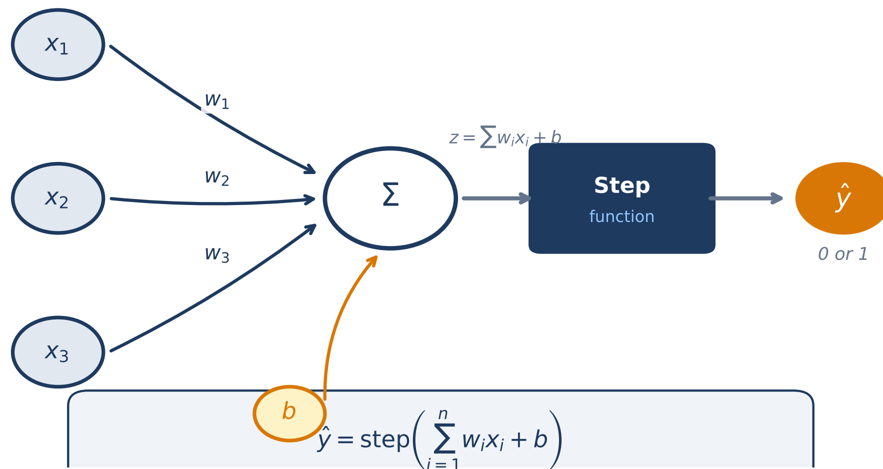

Figure 9: The perceptron computes a weighted sum, adds a bias, and passes the result through an activation function. This is logistic regression viewed as a single-neuron neural network.

Key Formula: The Perceptron

For input $x \in \mathbb{R}^n$, weight vector $w \in \mathbb{R}^n$, bias $b \in \mathbb{R}$, and activation $\sigma : \mathbb{R} \to \mathbb{R}$:

$$\hat{y} = \sigma\left(\sum_{i=1}^{n} w_i x_i + b\right) = \sigma(w^\top x + b).$$

Activations used in practice:

- **Step:** $\sigma(z) = \mathbf{1}[z \geq 0]$ (original 1958 form; binary output; not differentiable)
- **Sigmoid:** $\sigma(z) = 1/(1+e^{-z})$ (smooth; output in $(0, 1)$; interpretable as probability)
- **Tanh:** $\sigma(z) = \tanh(z)$ (smooth; output in $(-1, 1)$; zero-centered)
- **ReLU:** $\sigma(z) = \max(0, z)$ (piecewise linear; the modern default for hidden layers)

Plain English: Multiply every input by a weight, sum them up, add a baseline, then squash through a nonlinear function.

The Role of the Bias

The bias b is the neuron's "default activation" when all inputs are zero. Without a bias, the decision boundary $w^\top x + b = 0$ must pass through the origin. With a bias, the boundary can sit anywhere. Removing the bias is almost always a bad idea; it forces the model to represent problems that have no natural origin (like temperature in Celsius vs. Fahrenheit) through contorted weight configurations.

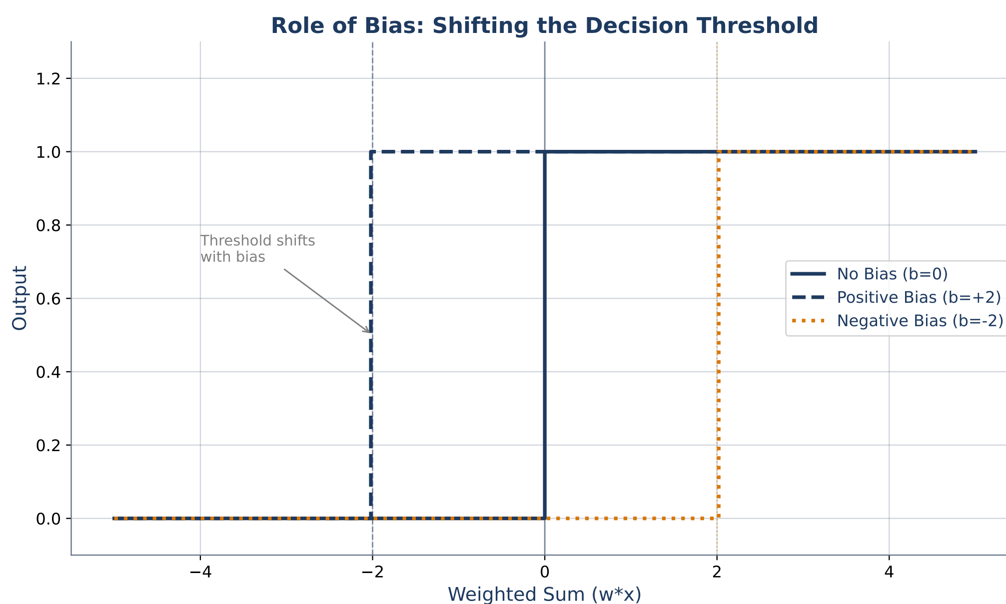

Figure 10: The role of the bias. Without a bias, the decision boundary must pass through the origin. Adding a bias lets the boundary shift freely, which is essential for most real problems.

Activation Functions: Step vs. Sigmoid

Rosenblatt's original perceptron used the step function. The step has a clean biological interpretation (fire / don't fire), but it is not differentiable at zero and has zero gradient everywhere else. Zero gradient means gradient-based learning cannot work; you can only train step-perceptrons with the original Rosenblatt rule (below). For modern networks we replace the step with a smooth sigmoid, which has a well-defined derivative $\sigma'(z) = \sigma(z)(1 - \sigma(z))$ everywhere.

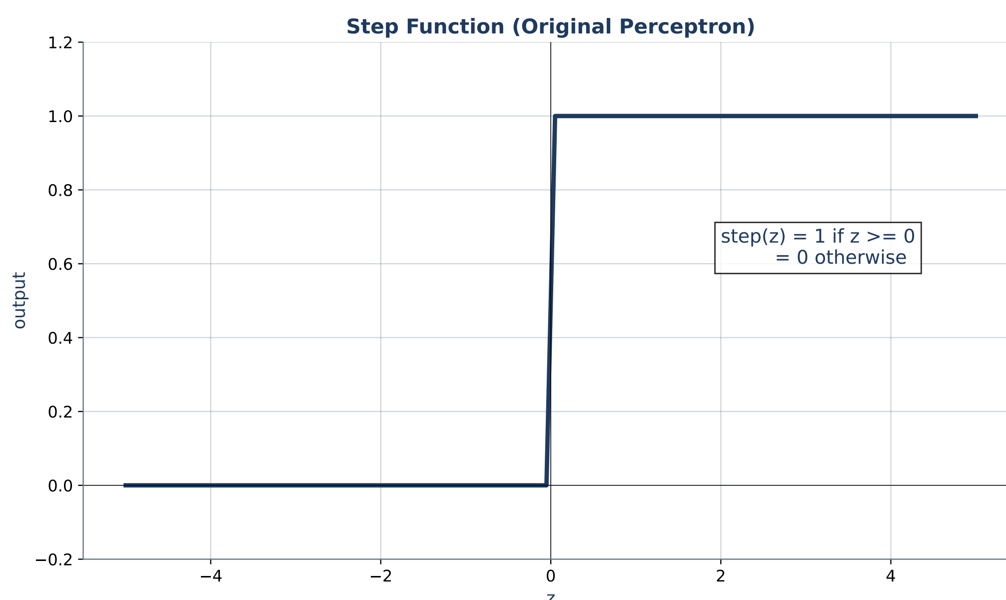

Figure 11: The step function. Output jumps from 0 to 1 at $z = 0$. Not differentiable; zero gradient away from the jump. Used by Rosenblatt in 1958.

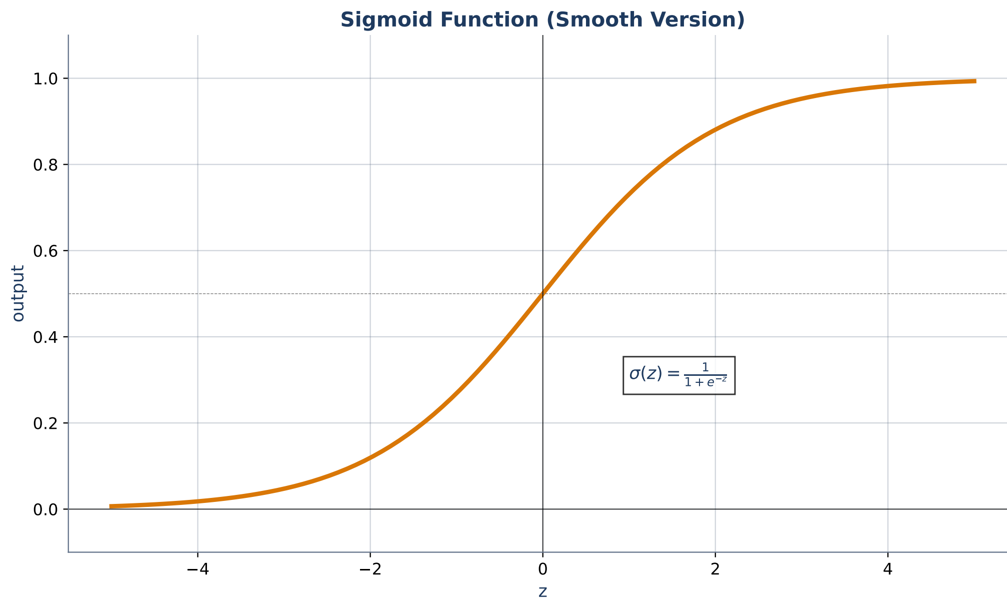

Figure 12: The sigmoid function. A smooth S-curve with outputs in $(0, 1)$ and a non-zero gradient everywhere. This is what modern gradient-based learning requires.

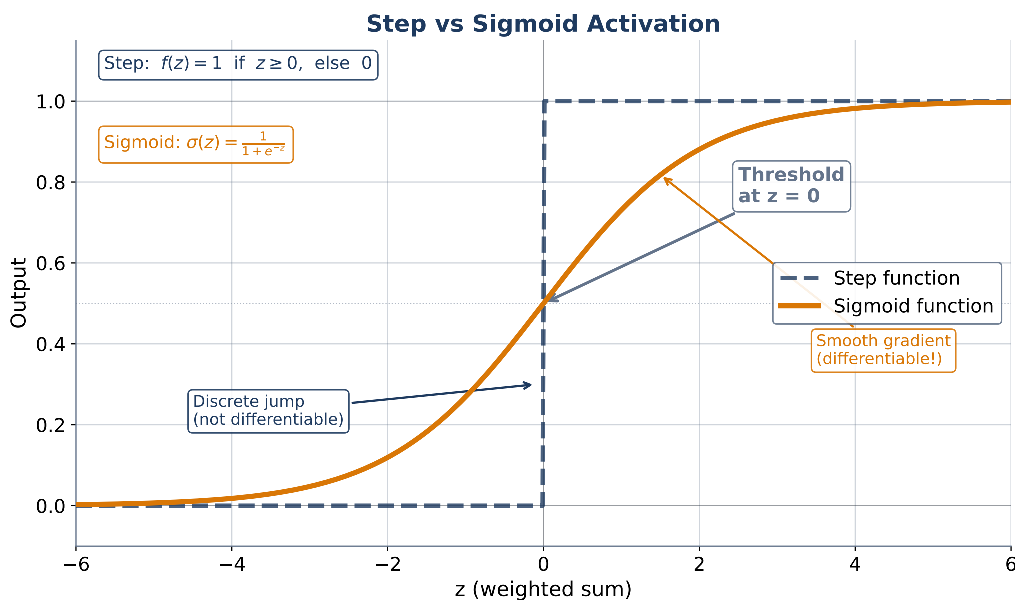

Figure 13: Step vs. sigmoid, overlaid. The step is the $\lim_{a\to\infty}$ of $\sigma(az)$. Sigmoid is a smooth, differentiable relaxation suitable for backpropagation.

Geometric Interpretation: The Perceptron Is a Hyperplane

The equation $w^\top x + b = 0$ describes a hyperplane in $\mathbb{R}^n$. Points with $w^\top x + b > 0$ lie on one side (the "positive" side), and points with $w^\top x + b < 0$ lie on the other. The perceptron's prediction depends on which side of the hyperplane the input falls on.

Decision boundary: The locus of input points where the perceptron's prediction flips. For a single perceptron with any monotonic activation, the decision boundary is the hyperplane $\{x : w^\top x + b = 0\}$.

Margin: The distance from a correctly classified point to the decision boundary. Larger margins usually indicate more confident, more generalizable classification. The margin of point x with label $y \in \{-1, +1\}$ is $y(w^\top x + b)/\|w\|$.

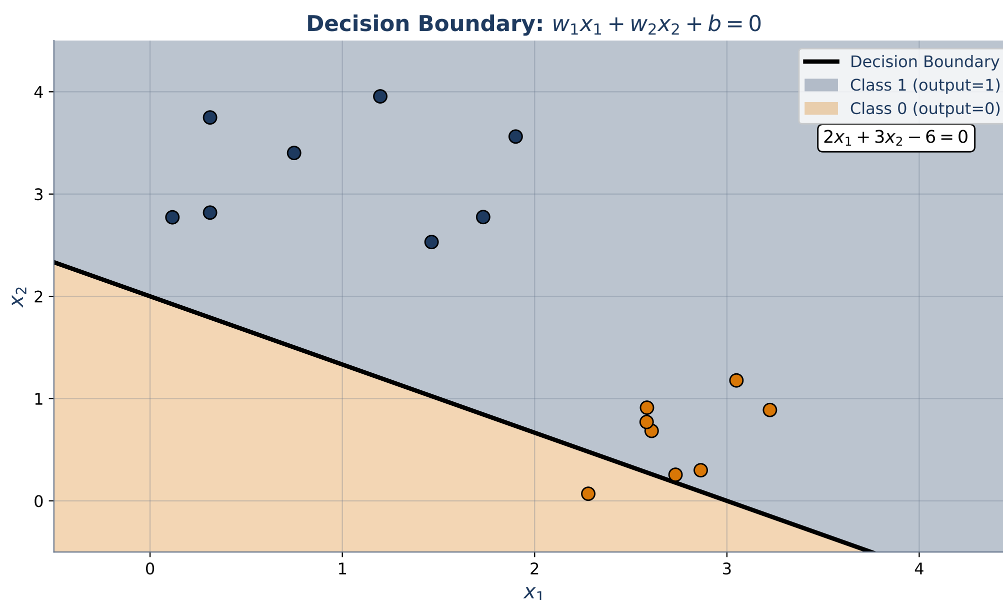

Figure 14: The perceptron's decision boundary is the hyperplane $w^\top x + b = 0$. The weight vector w is normal to the hyperplane; the bias b shifts it from the origin.

The Perceptron Learning Rule (Rosenblatt 1958)

Rosenblatt proposed a simple iterative rule for training a step-perceptron on labeled data. For each training example (x, y) with $y \in \{0, 1\}$:

$$w \leftarrow w + \eta(y - \hat{y})x, \qquad b \leftarrow b + \eta(y - \hat{y}),$$

where $\eta > 0$ is the learning rate. If the perceptron predicts correctly ($\hat{y} = y$), the update is zero: don't fix what isn't broken. If the perceptron predicts 0 when the label is 1, add ηx to w (nudge the weight vector toward the pattern). If the perceptron predicts 1 when the label is 0, subtract ηx.

Key Formula: Perceptron Learning Rule

For each training example (x, y), with learning rate $\eta > 0$:

$$w \leftarrow w + \eta(y - \hat{y})x, \qquad b \leftarrow b + \eta(y - \hat{y}).$$

Perceptron Convergence Theorem (Novikoff 1962): If the training data is linearly separable with margin $\gamma > 0$ and input norms bounded by R, the Rosenblatt rule converges in at most $(R/\gamma)^2$ mistakes. If the data is not linearly separable, the rule oscillates forever.

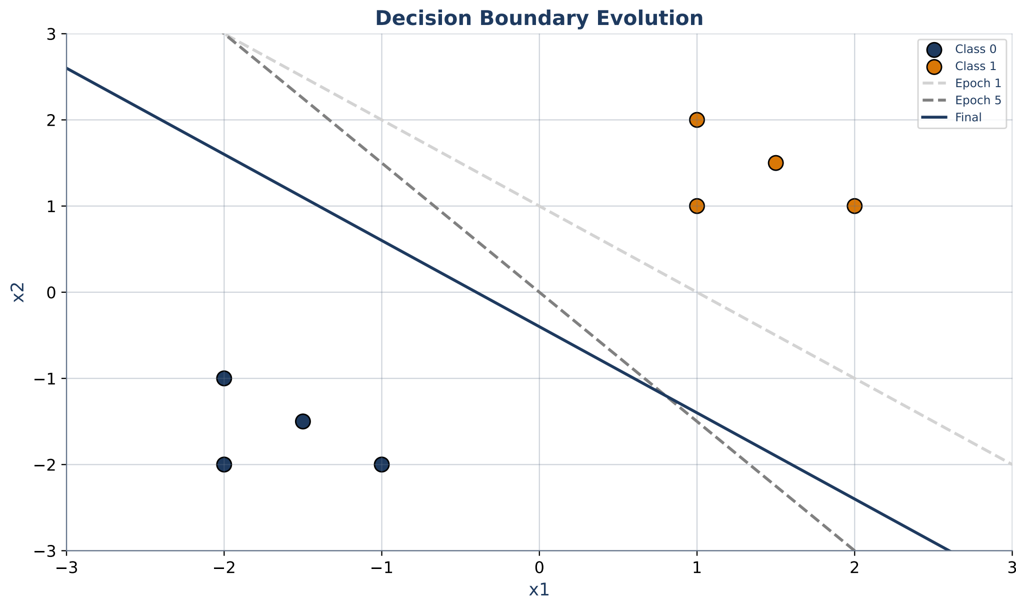

Figure 15: Evolution of the perceptron's decision boundary during training. Each mistake triggers an update that rotates the hyperplane toward correctly classifying the offending point. After enough iterations on linearly separable data, the hyperplane stabilizes.

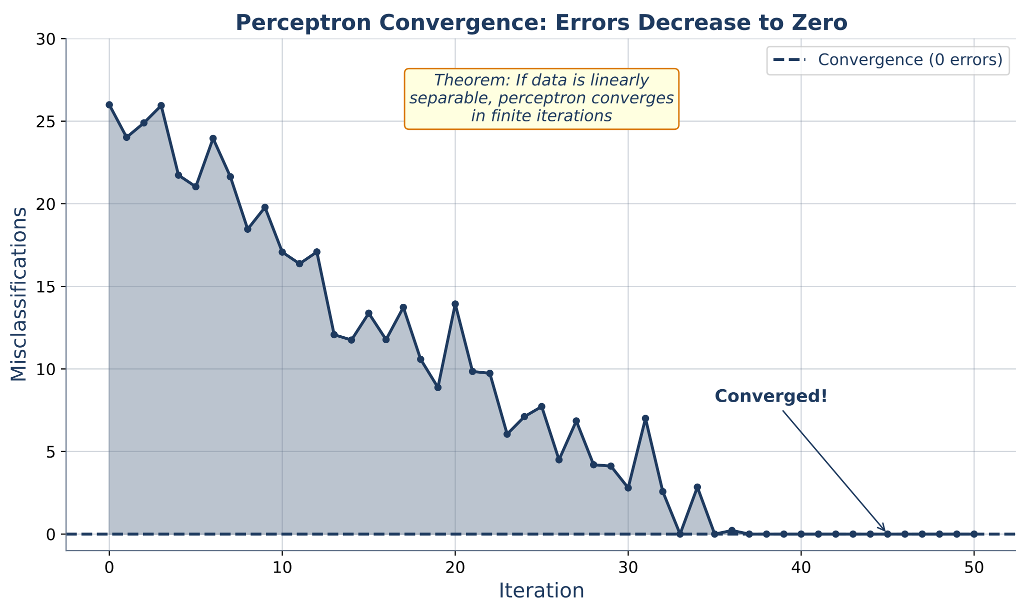

Figure 16: The perceptron convergence theorem. If the data is linearly separable with margin γ, the Rosenblatt rule makes at most $(R/\gamma)^2$ mistakes, where R bounds the input norms.

The AND, OR, and XOR Gates

The three classic Boolean gates are a useful testbed for single perceptrons:

- **AND:** Output 1 iff both inputs are 1. Linearly separable.
- **OR:** Output 1 iff at least one input is 1. Linearly separable.
- **XOR:** Output 1 iff exactly one input is 1. *Not* linearly separable.

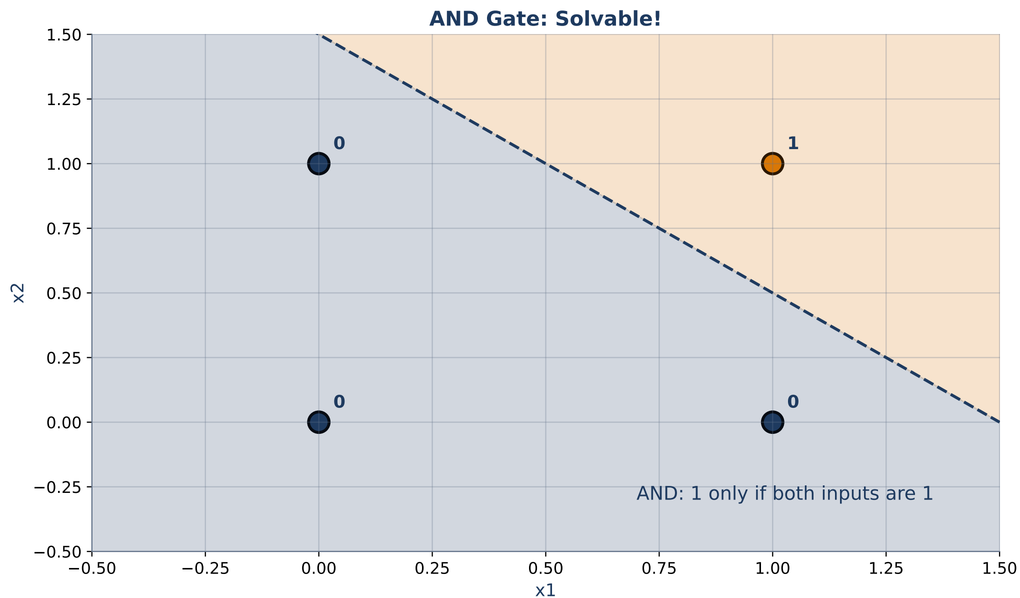

Figure 17: AND and OR gates are linearly separable. A single perceptron handles them easily.

A perceptron with weights $w_1 = w_2 = 1$ and bias $b = -1.5$ computes AND: output is 1 only when $x_1 + x_2 > 1.5$, i.e., both are 1. With the same weights but bias $b = -0.5$, it computes OR. For XOR, as we showed in Section 1, no such weights exist.

Definition: Single-Layer Perceptron

A single-layer perceptron is the function $f : \mathbb{R}^n \to \{0, 1\}$ (or $\mathbb{R}$) defined by

$$f(x) = \sigma(w^\top x + b)$$

with learnable parameters $w \in \mathbb{R}^n$, $b \in \mathbb{R}$, and a fixed activation σ. It is equivalent to logistic regression when σ is the sigmoid and training uses cross-entropy loss with gradient descent. Its decision boundary is always a hyperplane, which limits it to linearly separable problems.

Common Misconceptions about Perceptrons

(1) "A perceptron is fundamentally different from logistic regression." It is not. With a sigmoid activation and cross-entropy loss, the two are identical. The difference is historical: perceptrons emphasize the biological analogy and the step activation, while logistic regression emphasizes the probabilistic interpretation.
(2) "The bias is optional." It is not. Without a bias, the decision boundary must pass through the origin, which is rarely appropriate. Always include a bias.
(3) "The perceptron convergence theorem applies to any data." It applies only when the training data is *linearly separable*. On non-separable data (e.g., XOR), the Rosenblatt rule oscillates forever without converging.

Worked Examples

Worked Example 1: Training an AND-Gate Perceptron by Hand

Training data: $(0,0) \to 0$, $(0,1) \to 0$, $(1,0) \to 0$, $(1,1) \to 1$. Start with $w = (0,0)$, $b = 0$, $\eta = 1$, step activation.
Iteration 1 on $(0,0) \to 0$. Compute $z = 0$, $\hat{y} = \sigma(0) = 1$ (step function outputs 1 when $z \geq 0$). Error: $y - \hat{y} = 0 - 1 = -1$. Update: $w \leftarrow (0,0) + 1 \cdot (-1) \cdot (0,0) = (0,0)$; $b \leftarrow 0 + 1 \cdot (-1) = -1$.
Iteration 2 on $(0,1) \to 0$. $z = 0 + 0 + (-1) = -1$. $\hat{y} = 0$. Error $= 0$. No update.
Iteration 3 on $(1,0) \to 0$. $z = 0 + 0 + (-1) = -1$. $\hat{y} = 0$. Error $= 0$. No update.
Iteration 4 on $(1,1) \to 1$. $z = 0 + 0 + (-1) = -1$. $\hat{y} = 0$. Error $= 1$. Update: $w \leftarrow (0,0) + 1 \cdot 1 \cdot (1,1) = (1,1)$; $b \leftarrow -1 + 1 = 0$.
Iteration 5 on $(0,0) \to 0$. $z = 0$. $\hat{y} = 1$. Error $= -1$. Update: $b \leftarrow 0 - 1 = -1$. Now $w = (1,1)$, $b = -1$.
Continue a few more passes. The algorithm settles on something like $w = (1,1)$, $b = -1.5$, which gives $z = x_1 + x_2 - 1.5$. Only $(1,1)$ produces $z > 0$; all others give $z < 0$. Correct classification.

Worked Example 2: A Single-Neuron Credit Scorer

A bank uses a single sigmoid perceptron for a simple credit decision. Features: standardized debt-to-income ratio x_1 and standardized employment length x_2 (in years from the mean). Weights after training: $w_1 = -1.5$ (higher DTI hurts), $w_2 = 0.8$ (longer employment helps). Bias: $b = 0.3$.
Applicant A: DTI = 0.5 standard deviations above average (worrying), employment = 1.0 standard deviation above average (good).

$$z = -1.5(0.5) + 0.8(1.0) + 0.3 = -0.75 + 0.8 + 0.3 = 0.35.$$

$\hat{y} = \sigma(0.35) = 1/(1 + e^{-0.35}) \approx 0.587$. Probability of non-default $\approx 59\%$. Approve borderline.
Applicant B: DTI = 2.0 (very high), employment = -0.5 (short).

$$z = -1.5(2.0) + 0.8(-0.5) + 0.3 = -3.0 - 0.4 + 0.3 = -3.1.$$

$\hat{y} = \sigma(-3.1) \approx 0.043$. Probability of non-default $\approx 4\%$. Decline.
The single perceptron gives interpretable, smooth decisions on this two-feature problem. But notice the weights apply uniformly—if the interaction between DTI and employment matters (e.g., "high DTI is fine if employment is very long"), this single neuron cannot capture it. We need hidden layers.

Historical Background: Rosenblatt and the Mark I Perceptron (1958)

Frank Rosenblatt was a psychologist at Cornell Aeronautical Laboratory. In 1957 he proposed the perceptron as a mathematical model of biological learning. In 1958 he built the Mark I Perceptron, a physical machine with 400 photocells and a bank of motor-driven potentiometers that adjusted themselves during training. The machine was front-page news.
Rosenblatt published the theory in *Psychological Review* (1958) and elaborated in *Principles of Neurodynamics* (1962). His convergence theorem (later proved more cleanly by Novikoff in 1962) guaranteed that, on linearly separable data, the perceptron rule finds a separating hyperplane in finite time.
The 1969 Minsky–Papert critique was devastating—not because it was wrong, but because Rosenblatt had oversold. The Mark I was limited, and the press had turned him into a prophet of conscious machines. When the limits became public, the backlash was severe. Rosenblatt died in a boating accident in 1971 at age 43, long before the 1986 backpropagation revival vindicated his core ideas.
Every modern neural network is a descendant of Rosenblatt's perceptron. The learning rule has been replaced by gradient descent; the step activation has been replaced by ReLU; single units have been replaced by layers of thousands. But the core motif—weighted sum, add bias, pass through nonlinearity—is the same Rosenblatt drew on a napkin in 1957.

Problem 2.1 (Easy) ⋆

A perceptron has weights $w = (0.5, -1.0, 2.0)$ and bias $b = -0.3$. Compute the output z before activation for input $x = (2, 1, 0.5)$. Does the step function produce 0 or 1?
Solution: see Appendix.

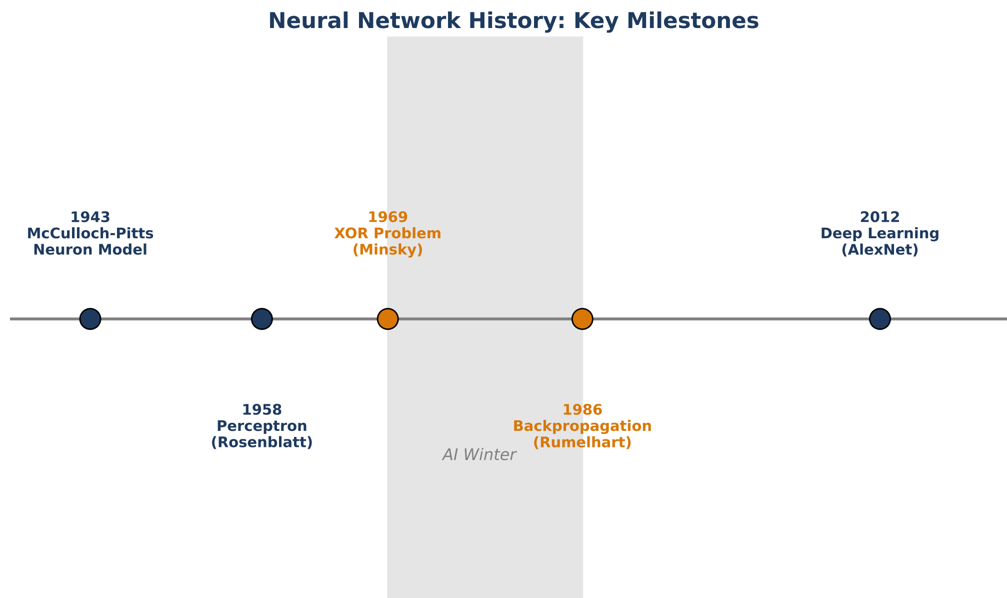

Figure 18: Key milestones in the development of artificial neurons: McCulloch–Pitts (1943, binary threshold), Rosenblatt (1958, trainable perceptron), Widrow–Hoff (1960, continuous gradient learning), and the later backpropagation era.

Problem 2.2 (Easy) ⋆

Design weights and a bias for a two-input perceptron that computes the OR function. Verify on all four inputs.
Solution: see Appendix.

Problem 2.3 (Medium) ⋆⋆

A perceptron has $w = (1, 1)$ and $b = -1.5$ with step activation. It sees the training example $(x, y) = ((2, 0), 0)$ (expected output 0). What is its current prediction? Apply one update of the Rosenblatt rule with $\eta = 0.5$ and state the new weights and bias.
Solution: see Appendix.

Problem 2.4 (Medium) ⋆⋆

The NAND gate has truth table: $(0, 0) \to 1$, $(0, 1) \to 1$, $(1, 0) \to 1$, $(1, 1) \to 0$. Is it linearly separable? If yes, give weights and a bias that implement it. If no, explain why.
Solution: see Appendix.

Problem 2.5 (Hard) ⋆ ⋆ ⋆

Prove the perceptron convergence theorem in the margin formulation. Specifically: assume training examples (x_i, y_i) with $y_i \in \{-1, +1\}$, $\|x_i\| \le R$, and a vector w^* with $\|w^*\| = 1$ and $y_i(w^*)^\top x_i \ge \gamma > 0$ for all i. Show that the perceptron rule with $\eta = 1$ makes at most $(R/\gamma)^2$ mistakes. (*Hint:* Track both $w_t^\top w^*$ (grows at least γ per mistake) and $\|w_t\|^2$ (grows at most R^2 per mistake). Combine with Cauchy–Schwarz.)
Solution: see Appendix.

(c) Joerg Osterrieder 2026

Connecting Forward

A single perceptron is a hyperplane classifier with a simple learning rule and a clean convergence guarantee on linearly separable data. But Section 1 already showed its Achilles' heel: XOR and any other non-separable problem. Section 3 lifts this restriction by stacking perceptrons into multilayer networks. We will see the universal approximation theorem made precise, meet the activation functions that make deep networks trainable, and solve XOR explicitly with a two-neuron hidden layer.

Key Takeaway: A perceptron is a hyperplane classifier; its Rosenblatt learning rule converges on linearly separable data but oscillates forever on XOR-like problems—this limitation motivated the multilayer architectures we build next.

(c) Joerg Osterrieder 2026

3. Stacking Neurons – Multilayer Perceptrons and Activation Functions

Opening Problem: The Regime-Detection Analyst

A systematic hedge fund runs a single trading strategy that works beautifully in “low-volatility trending” markets and disastrously in “high-volatility choppy” markets. The portfolio manager wants a regime-classification model: take the past 20 days of market features (realized volatility, average return, skewness, VIX level, yield-curve slope) and output a probability that tomorrow will be a low-volatility trending regime.
The analyst first tries logistic regression. Some features help individually (high VIX $\rightarrow$ lower probability of trending), but the joint pattern is fundamentally nonlinear. Low VIX AND steep yield curve is one regime; low VIX AND flat yield curve is another. The interactions matter more than the main effects. Logistic regression gives AUC ≈ 0.62—not good enough to act on.
The analyst needs a model that automatically discovers interactions without being told which to look for. She wants to throw in the 20 raw features, let the model build its own internal representations, and output a regime probability. That is exactly what a multilayer perceptron (MLP) does. Section 3 explains how.

Discovery Question

One perceptron gives one hyperplane. Two perceptrons give two hyperplanes. If you feed the output of the first two perceptrons into a third perceptron, what kind of decision regions can you carve out of the input space? How does this change if there are 100 hidden perceptrons instead of 2?

From One Layer to Many

A multilayer perceptron (MLP) is what you get when you stack perceptrons into layers and apply a nonlinear activation between layers. The first layer—called the *input layer*—holds the raw features. Every intermediate layer is a *hidden layer*. The last layer is the *output layer*, which produces the prediction.

Hidden layer: Any layer of neurons between the input and output layers of a neural network. The term “hidden” reflects the fact that these layers’ activations are not directly observed in the training data—only inputs and outputs are given. Hidden layers learn intermediate representations.

Feedforward network: A neural network where information flows in one direction, from input to output, with no cycles. The MLP is the canonical feedforward architecture. Recurrent networks (RNNs, LSTMs) are NOT feedforward—they have loops.

Each hidden neuron computes $h_j = \sigma(w_j^\top x + b_j)$, exactly like the perceptron in Section 2. But now we have multiple such neurons in parallel, stacked into a layer. The output of layer ℓ becomes the input to layer $\ell + 1$. The composition of many such layers, with nonlinearities in between, produces complex decision boundaries that no single layer could.

Forward Propagation: Computing the Output

Given an input x, forward propagation computes the network’s output by evaluating each layer in sequence. For a three-layer network (input, one hidden, output):

$$h = \sigma_h(W_1 x + b_1), \qquad \hat{y} = \sigma_y(W_2 h + b_2).$$

For a general L-layer network: let $h^{(0)} = x$ and

$$h^{(\ell)} = \sigma_\ell(W_\ell h^{(\ell-1)} + b_\ell), \qquad \ell = 1, 2, \ldots, L.$$

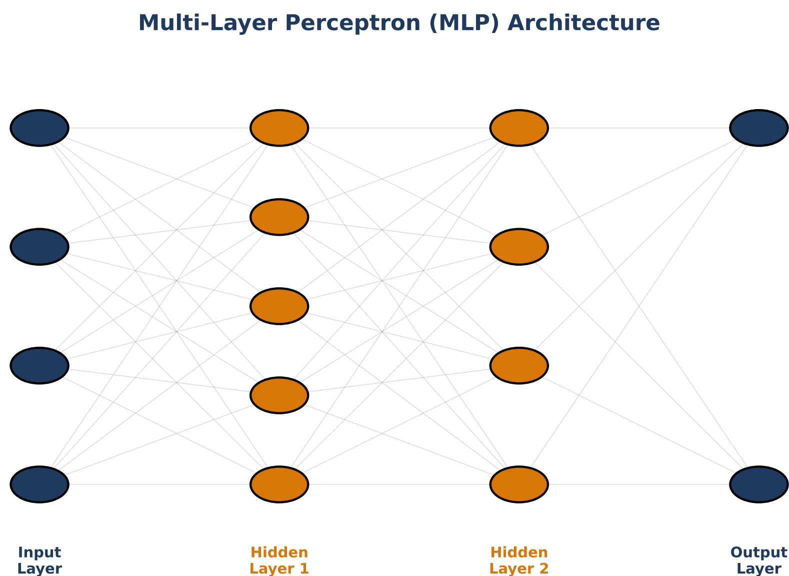

Figure 19: A multilayer perceptron (MLP). Input, hidden, and output layers are fully connected. Each neuron in a layer receives from every neuron in the previous layer.

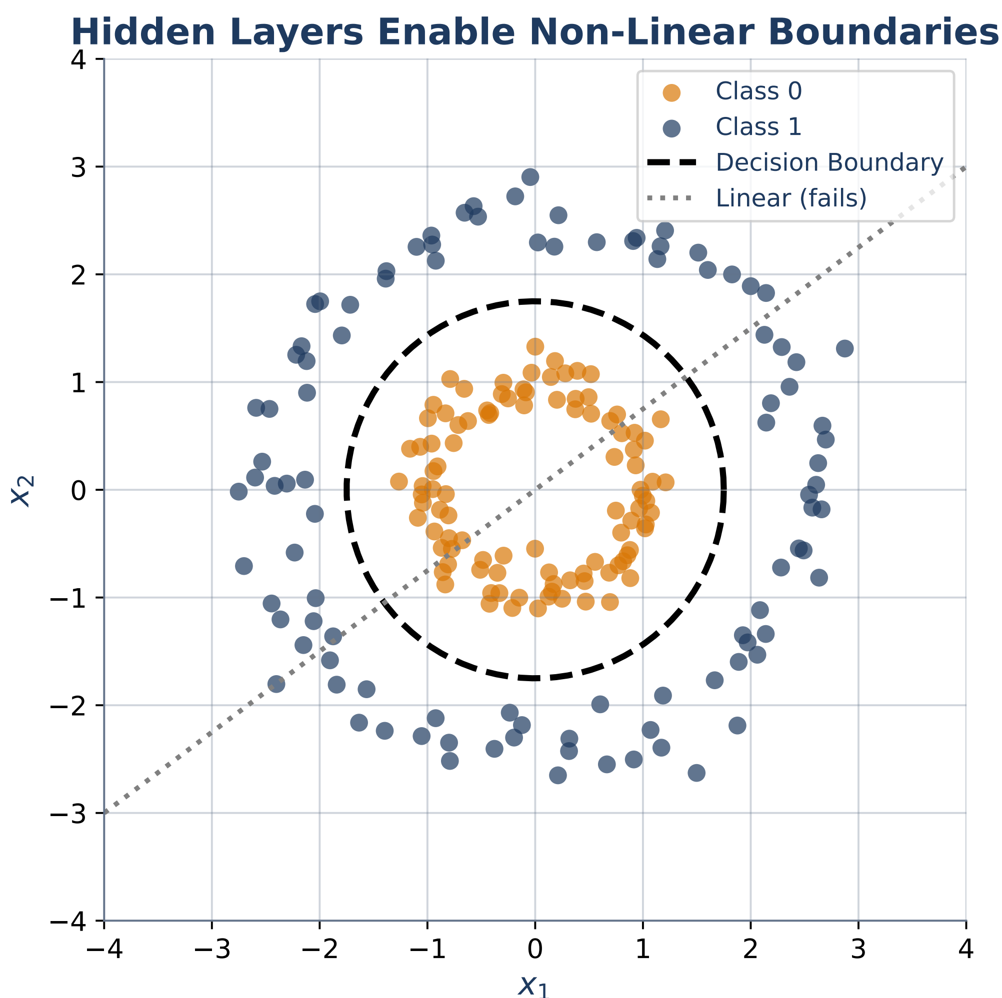

Figure 20: Why hidden layers matter. The first hidden layer detects simple features (edges in image data, threshold crossings in tabular data). Deeper layers combine these into richer patterns.

The final output is $\hat{y} = h^{(L)}$.

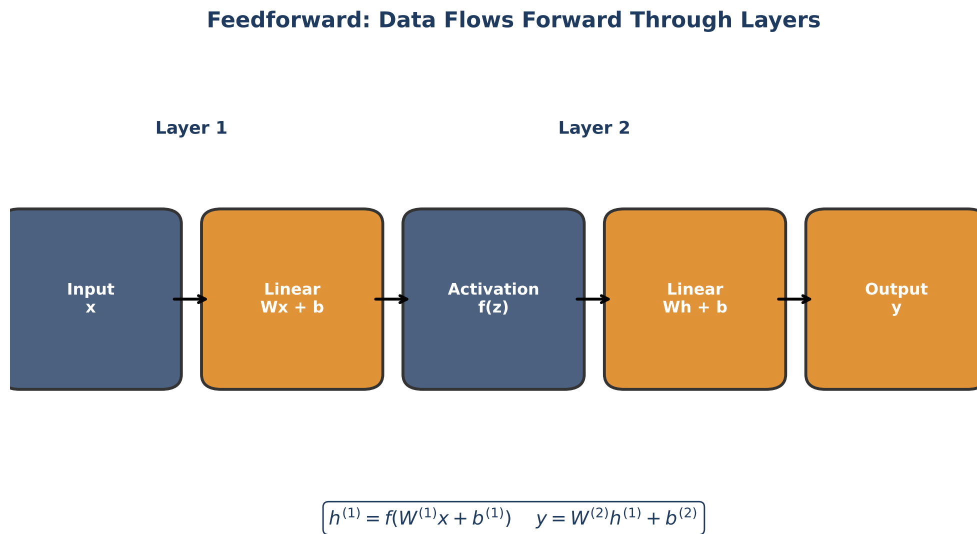

Figure 21: Forward propagation through an MLP. Each layer performs a linear transformation followed by a nonlinear activation. The input is transformed step by step until it becomes the prediction.

Key Formula: Forward Propagation

For a neural network with layers $\ell = 1, \dots, L$, weight matrices W_ℓ, bias vectors b_ℓ, and activations σ_ℓ:

$$h^{(0)} = x, \qquad z^{(\ell)} = W_\ell h^{(\ell-1)} + b_\ell, \qquad h^{(\ell)} = \sigma_\ell(z^{(\ell)}).$$

The network's output is $\hat{y} = h^{(L)}$.
Parameter count: layer ℓ has $d_\ell \cdot d_{\ell-1}$ weights and d_ℓ biases, where d_ℓ is the number of neurons. Total parameters:

$$P = \sum_{\ell=1}^{L} (d_\ell \cdot d_{\ell-1} + d_\ell).$$

A network with input dimension 20, one hidden layer of 64 neurons, and output dimension 1 has $P = (64 \cdot 20 + 64) + (1 \cdot 64 + 1) = 1344 + 65 = 1409$ parameters.

The Universal Approximation Theorem

The headline theoretical result of the 1980s–1990s neural network literature is the universal approximation theorem. Multiple versions exist; here is a representative one.

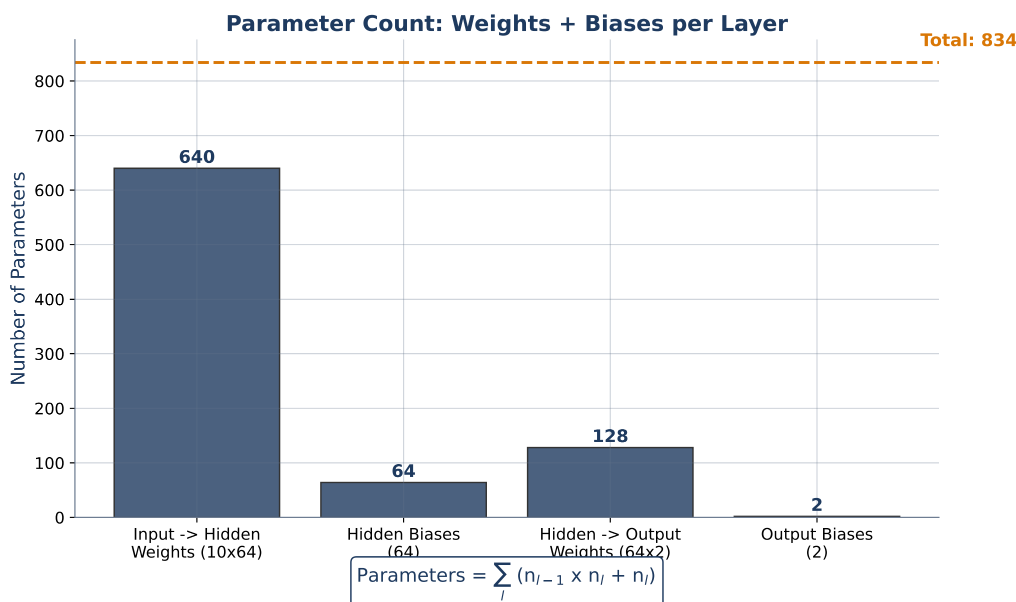

Figure 22: Parameter count formula. Each fully connected layer contributes $d_\ell \cdot d_{\ell-1}$ weights plus d_ℓ biases. Deep, wide networks can easily reach millions of parameters.

Theorem (Hornik 1991, simplified)

Let $\sigma : \mathbb{R} \to \mathbb{R}$ be any continuous, non-constant, bounded, monotonically increasing function (e.g., sigmoid). For any continuous function $f : [0,1]^n \to \mathbb{R}$ and any $\epsilon > 0$, there exists an MLP with one hidden layer of finite size m and weights (W, b) such that

$$\sup_{x \in [0,1]^n} |f(x) - \hat{f}_{W,b}(x)| < \epsilon.$$

In words: a one-hidden-layer network with enough neurons can approximate any continuous function on a bounded domain to arbitrary accuracy.

Universal approximation is a *existence* result, not a *learnability* result. It says the network *can* represent the target function, not that gradient descent will find the right weights in reasonable time. It also does not promise a small number of neurons: approximating complex functions to high accuracy may require a huge m. In practice, deeper networks (more layers, each with moderate width) often approximate the same function more efficiently than very wide single-layer networks.

Activation Functions: Sigmoid, Tanh, ReLU, and Friends

The nonlinear activation is the engine that gives MLPs their power. Without it, even a 100-layer network collapses to a single linear map. Choose the wrong activation and the network becomes untrainable (vanishing gradients) or unstable (exploding outputs). Here are the usual suspects.

Sigmoid: $\sigma(z) = 1/(1+e^{-z})$. Output in $(0,1)$. Historical default. Downside: for $|z|$ large, $\sigma'(z) \approx 0$—this is the **vanishing gradient problem**. Backpropagation through a deep sigmoid network attenuates gradients exponentially; the early layers learn almost nothing.

Tanh: $\tanh(z) = (e^z - e^{-z})/(e^z + e^{-z})$. Output in $(-1,1)$, zero-centered. Slightly better than sigmoid because its output has zero mean, but still suffers from vanishing gradients.

ReLU: $\sigma(z) = \max(0, z)$. The modern default for hidden layers. Gradient is exactly 1 for $z > 0$ and 0 for $z < 0$. No vanishing gradient on the positive side. But a "dead ReLU" can happen: if a neuron's input is always negative, its gradient is always zero and it never learns

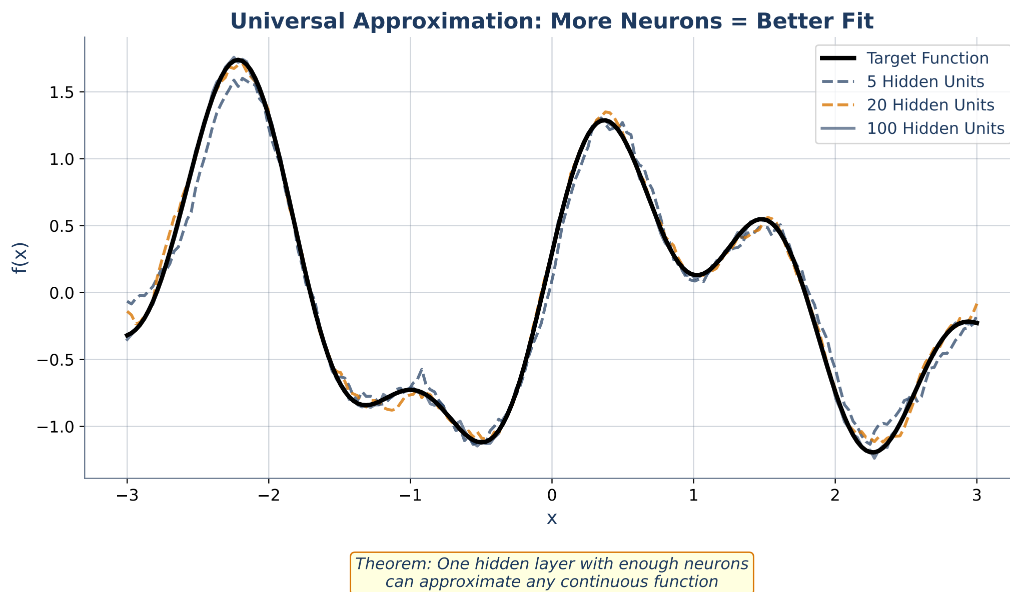

Figure 23: Universal approximation, illustrated. With enough hidden neurons, a single-layer MLP can match any continuous target function. Curved boundaries, periodic patterns, discontinuities—all within reach.

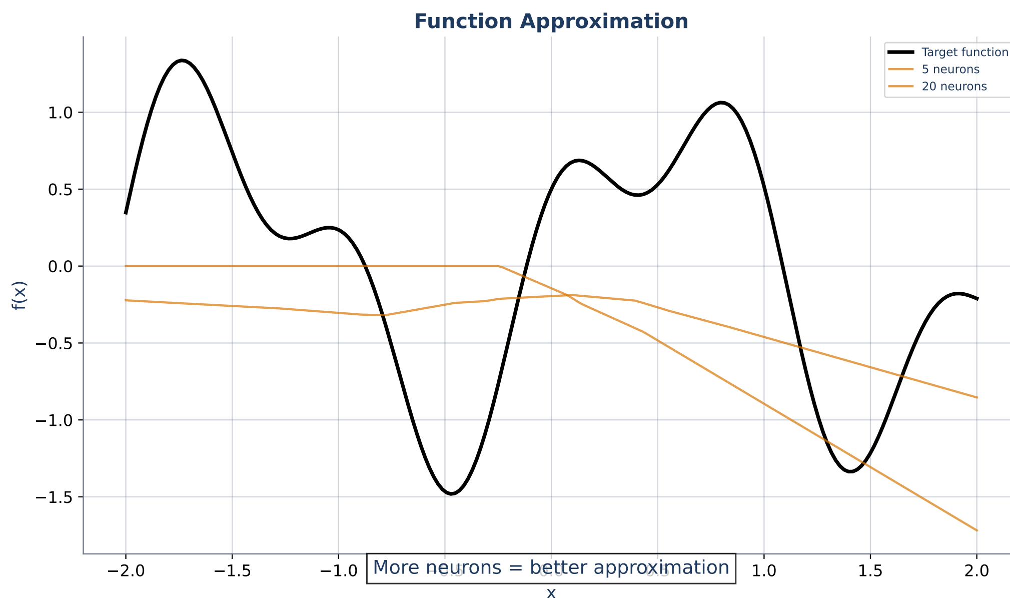

Figure 24: Approximating a complicated function with a one-hidden-layer MLP. As the number of hidden units grows, the approximation converges to the target.

(c) Joerg Osterrieder 2026

again.

Leaky ReLU: $\sigma(z) = \max(\alpha z, z)$ with small $\alpha \approx 0.01$. Avoids dead neurons by giving a small negative slope.

ELU: $\sigma(z) = z$ for $z > 0$ and $\alpha(e^z - 1)$ for $z \leq 0$. Smoother than ReLU near zero; self-normalizing variants (SELU) can stabilize deep networks.

Softmax (for output layer in multiclass): $\sigma(z)_k = e^{z_k} / \sum_j e^{z_j}$. Maps a K-vector to a probability distribution over K classes. Not used in hidden layers.

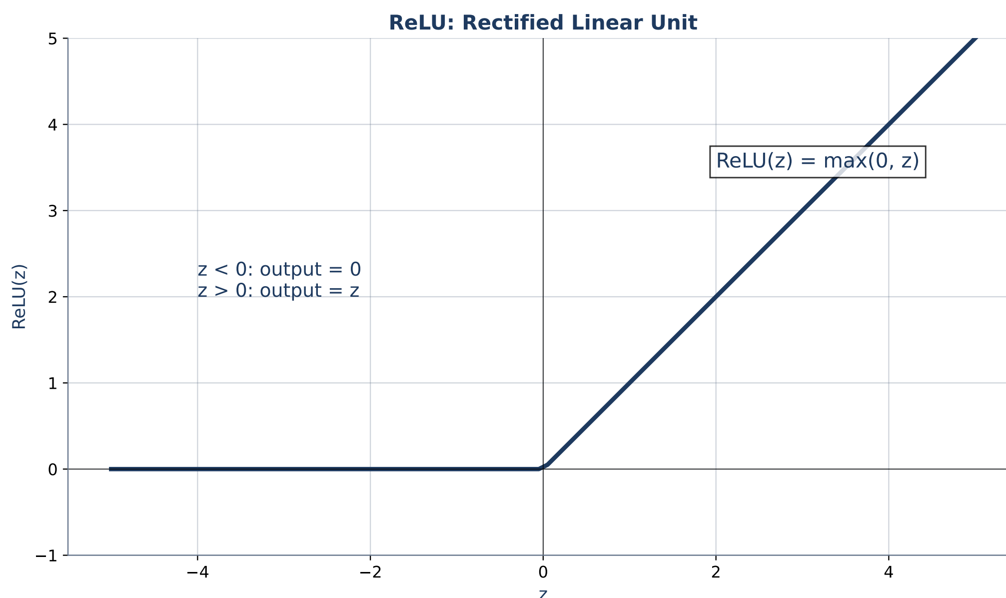

Figure 25: The ReLU activation: $\max(0, z)$. Linear for positive inputs, zero for negatives. Simple, fast, and surprisingly effective.

Key Formula: Common Activations and Their Derivatives

- **Sigmoid:** $\sigma(z) = \dfrac{1}{1 + e^{-z}}, \quad \sigma'(z) = \sigma(z)(1 - \sigma(z)) \in (0, 0.25]$.
- **Tanh:** $\tanh(z), \quad \tanh'(z) = 1 - \tanh^2(z) \in (0, 1]$.
- **ReLU:** $\mathrm{relu}(z) = \max(0, z), \quad \mathrm{relu}'(z) = \mathbf{1}[z > 0]$.
- **Leaky ReLU:** $\mathrm{lrelu}(z) = \max(\alpha z, z), \quad \mathrm{lrelu}'(z) = \alpha$ for $z \leq 0$, 1 for $z > 0$.
- **Softmax:** $\sigma(z)_k = \dfrac{e^{z_k}}{\sum_j e^{z_j}}, \quad \dfrac{\partial \sigma_k}{\partial z_j} = \sigma_k(\delta_{kj} - \sigma_j)$.

Sigmoid's maximum derivative is 0.25; after many layers, gradients get multiplied many times by a number ≤ 0.25 and vanish. ReLU's derivative is 0 or 1; no shrinking except at dead neurons.

Why ReLU Won

When the 2006–2012 deep learning revival happened, ReLU was the activation that made it work on deep architectures. Three reasons:

1. **Gradient flow.** ReLU's gradient is 1 for positive inputs. Stacked 100 deep, gradients stay roughly constant rather than collapsing to zero.

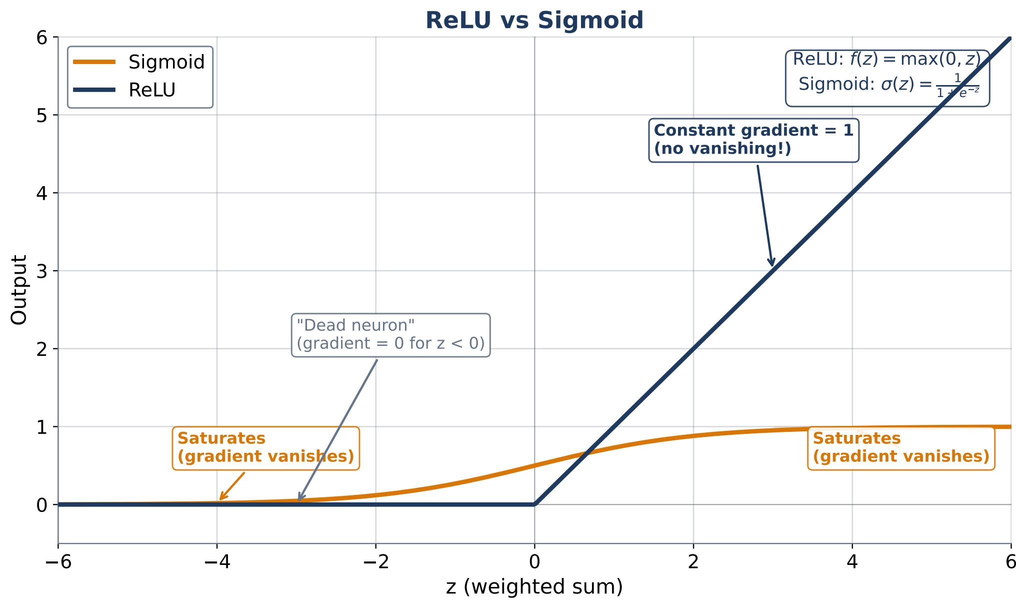

Figure 26: ReLU vs. sigmoid, overlaid. ReLU's gradient is 1 in the positive regime; sigmoid's gradient saturates near zero for large $|z|$. This gradient-preserving property is why ReLU won.

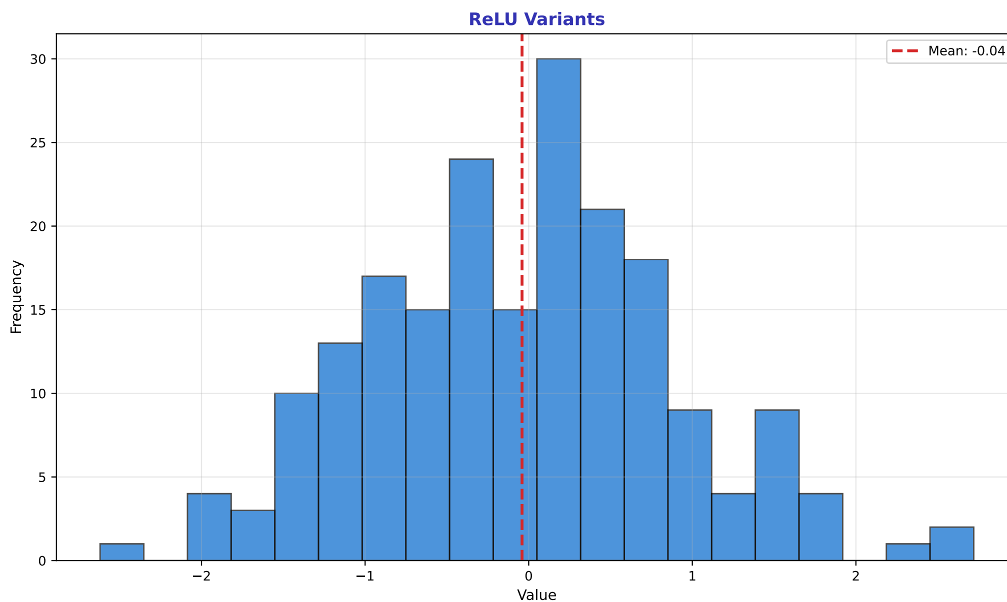

Figure 27: ReLU variants: Leaky ReLU, ELU, GELU, and others. Each addresses the "dead neuron" issue with a different trade-off between smoothness and sparsity.

2. **Sparsity.** Roughly half of ReLU neurons output zero for any given input. Sparse activations ease optimization and improve generalization (implicit regularization).

3. **Computational speed.** $\max(0, z)$ is a single comparison; $1/(1+e^{-z})$ requires an exponential. At scale, this matters.

Softmax is used for the output layer in multiclass classification. It is *not* used for hidden layers because it couples all its outputs—changing one neuron's input changes the outputs of all. In hidden layers you want independent nonlinear activations, not coupled ones.

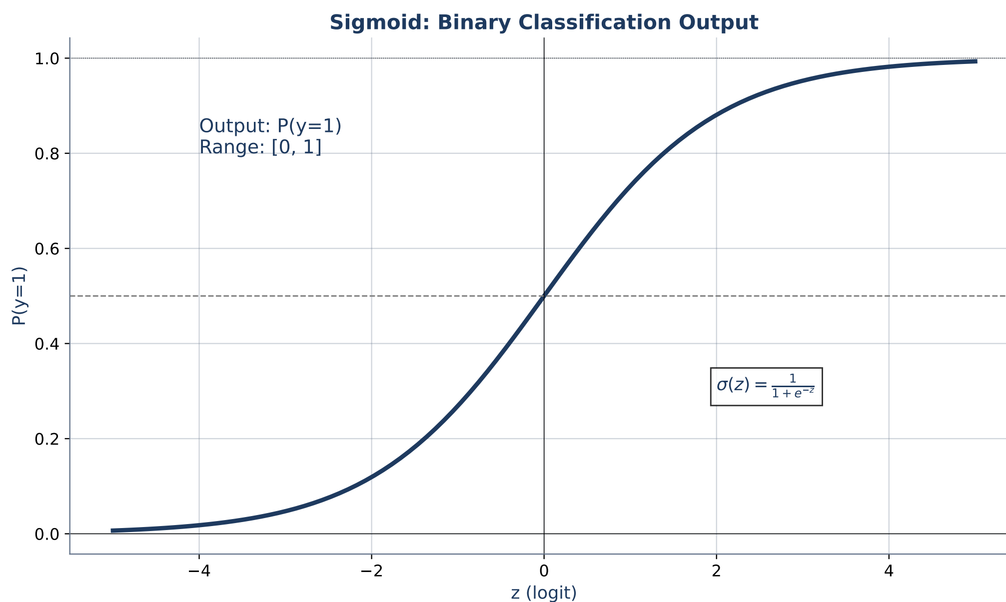

Figure 28: Sigmoid at the output for binary classification. Maps the network's pre-activation score to a probability in $(0, 1)$.

Architectural Choices: Width and Depth

Two knobs control model capacity: the **width** of each layer (number of neurons) and the **depth** of the network (number of layers). For a fixed parameter budget, deeper networks usually outperform wider ones on real problems. Width adds neurons that see the same input; depth adds hierarchies of features.

Definition: Multilayer Perceptron (MLP)

An MLP is a feedforward neural network with at least one hidden layer, where each layer is fully connected (every neuron in layer ℓ receives from every neuron in layer $\ell - 1$). Formally:

1. Input layer: $h^{(0)} = x \in \mathbb{R}^{d_0}$.

2. Hidden layers: $h^{(\ell)} = \sigma(W_\ell h^{(\ell-1)} + b_\ell)$ for $\ell = 1, \ldots, L-1$ with $W_\ell \in \mathbb{R}^{d_\ell \times d_{\ell-1}}$, nonlinear activation σ.

3. Output layer: $\hat{y} = \sigma_y(W_L h^{(L-1)} + b_L)$, where σ_y is sigmoid (binary classification), softmax (multiclass), or identity (regression).

By the universal approximation theorem (Hornik 1991), an MLP with one hidden layer and enough neurons can approximate any continuous function on a bounded domain.

(c) Joerg Osterrieder 2026

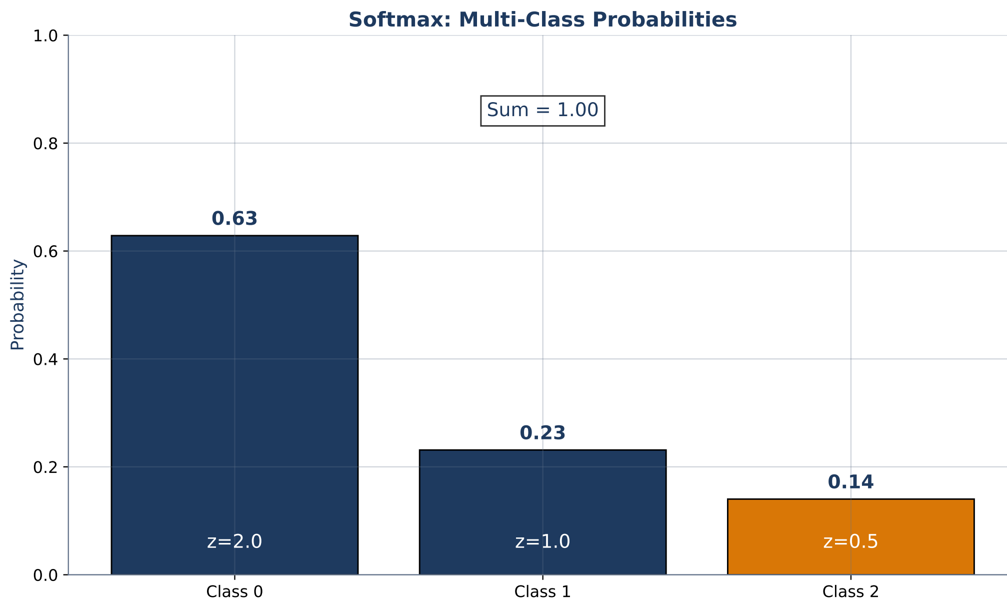

Figure 29: Softmax at the output for multiclass classification. Maps a vector of scores to a probability distribution. Larger scores get exponentially more probability mass.

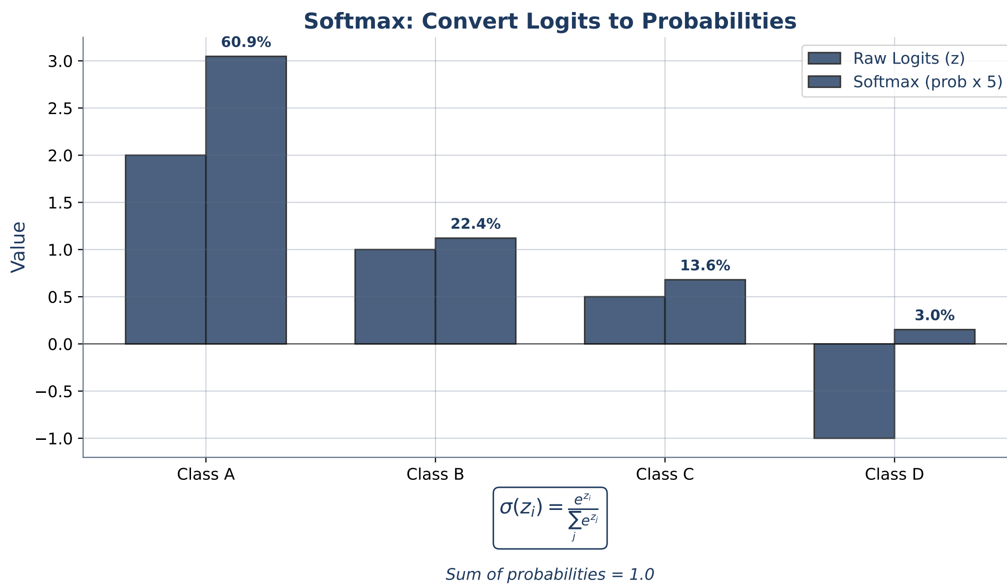

Figure 30: The softmax formula: exponentiate each score, then normalize. Translation-invariant: $\text{softmax}(z + c) = \text{softmax}(z)$ for any constant c.

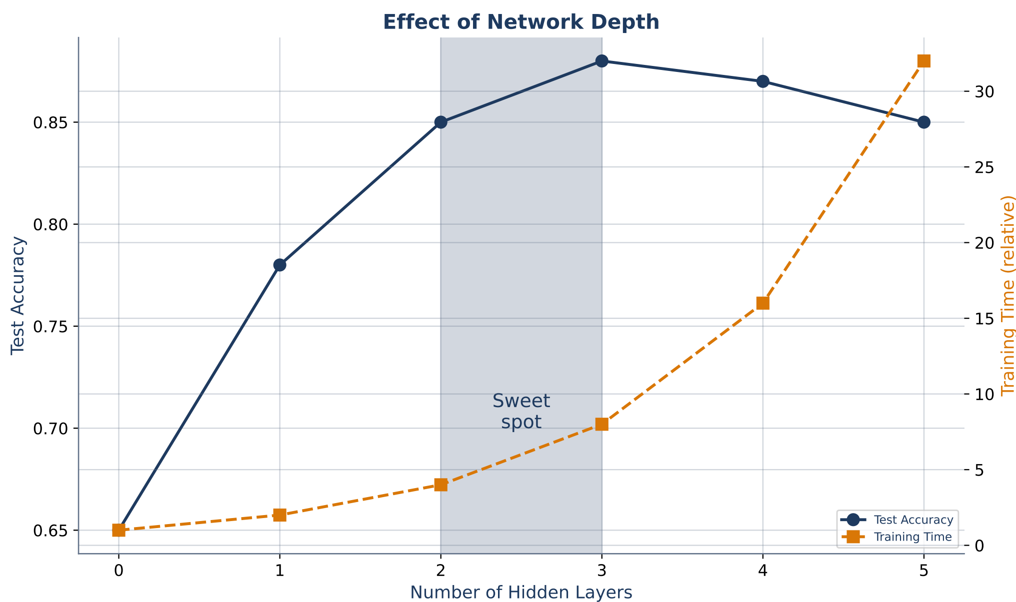

Figure 31: Effect of network depth on decision boundary complexity. Deeper networks build more nuanced boundaries by composing simple pieces.

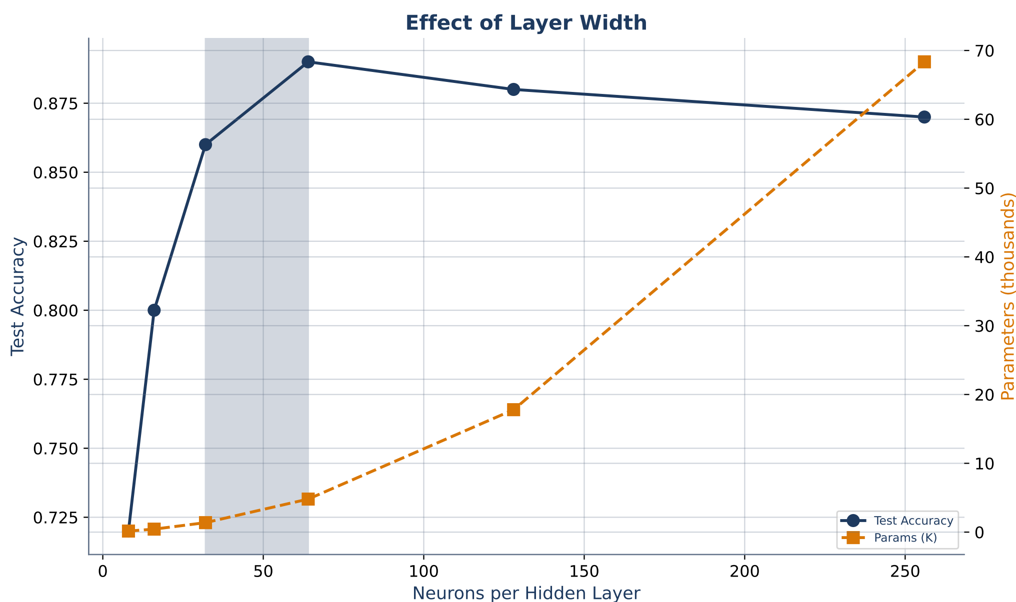

Figure 32: Effect of layer width (neurons per layer). Wider layers can represent more patterns at each level of abstraction but are prone to overfit.

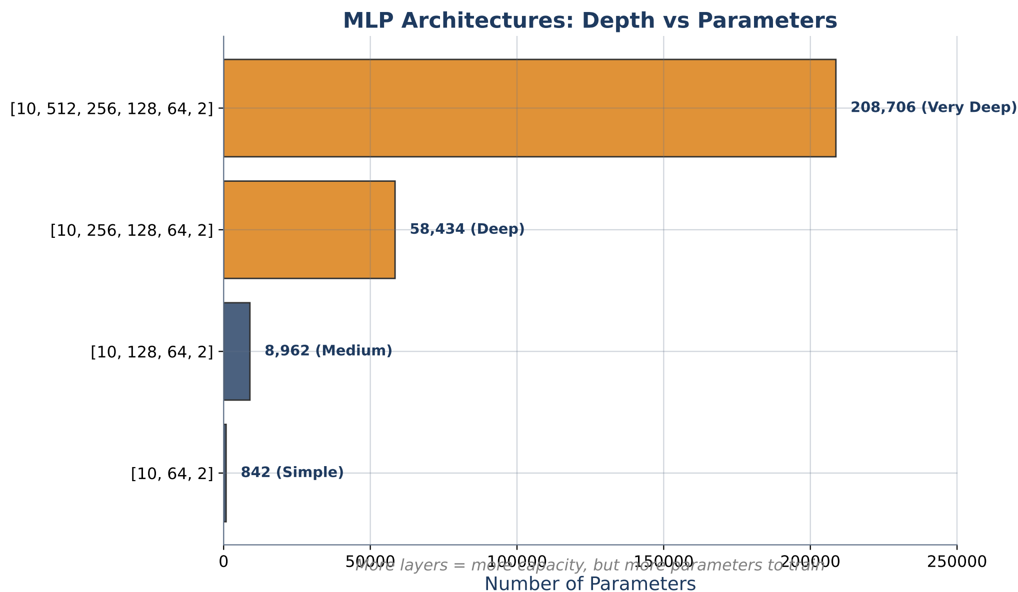

Figure 33: Sample architectures for different problem sizes: tiny (20 input, 16 hidden, 1 output), medium (100 input, two hidden layers of 64), and large (a deep MLP for a complex regression problem).

Common Misconceptions about MLPs and Activations

(1) "Deeper is always better." Adding depth increases capacity but also increases optimization difficulty and overfit risk. In practice, 2–5 hidden layers suffice for most tabular problems; 10–100+ are needed for images and text, mediated by convolutions or attention rather than pure MLP stacks.
(2) "Sigmoid is fine for hidden layers." No. In deep networks, sigmoid's gradient saturates for $|z| > 3$ or so, and backpropagation through many sigmoid layers causes gradients to vanish. Use ReLU (or a variant) for hidden layers; reserve sigmoid for the output of binary classifiers.
(3) "Universal approximation means MLPs always win." The theorem guarantees representational capacity, not optimization success. In practice, convolutional networks beat MLPs on images and gradient boosting beats MLPs on tabular data, because the right inductive bias matters as much as raw capacity.

Worked Examples

Worked Example 1: Solving XOR with Two Hidden Neurons

Let us solve XOR explicitly with an MLP that has 2 inputs, 2 hidden neurons with ReLU, and 1 output neuron with a linear activation.
Choose weights:

$$W_1 = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}, \quad b_1 = \begin{pmatrix} 0 \\ -1 \end{pmatrix}, \quad W_2 = \begin{pmatrix} 1 & -2 \end{pmatrix}, \quad b_2 = 0.$$

Let the hidden layer compute $h_j = \max(0, W_1 x + b_1)_j$.

- $x = (0,0)$: $z_1 = (0,0) + (0,-1) = (0,-1)$. $h = (\max(0,0), \max(0,-1)) = (0,0)$. Output $= 1(0) - 2(0) = 0$. ✓
- $x = (0,1)$: $z_1 = (1,1) + (0,-1) = (1,0)$. $h = (1,0)$. Output $= 1(1) - 2(0) = 1$. ✓
- $x = (1,0)$: $z_1 = (1,1) + (0,-1) = (1,0)$. $h = (1,0)$. Output $= 1$. ✓
- $x = (1,1)$: $z_1 = (2,2) + (0,-1) = (2,1)$. $h = (2,1)$. Output $= 1(2) - 2(1) = 0$. ✓

Perfect. Two hidden neurons plus a linear output solve XOR. The first hidden neuron computes an OR-like pattern; the second computes an AND-like pattern; the output subtracts $2\times$AND from OR to produce XOR. This is exactly the kind of representation a trained MLP would discover on its own.

Worked Example 2: Counting Parameters in a Finance MLP

A classification MLP for credit risk has input dimension 20 (features), two hidden layers of 64 neurons with ReLU, and one output neuron with sigmoid. Compute the total number of trainable parameters.

- Layer 1: $W_1 \in \mathbb{R}^{64\times 20}$, $b_1 \in \mathbb{R}^{64}$. Params: $64 \cdot 20 + 64 = 1344$.
- Layer 2: $W_2 \in \mathbb{R}^{64\times 64}$, $b_2 \in \mathbb{R}^{64}$. Params: $64 \cdot 64 + 64 = 4160$.
- Output: $W_3 \in \mathbb{R}^{1\times 64}$, $b_3 \in \mathbb{R}$. Params: $64 + 1 = 65$.

Total: $1344 + 4160 + 65 = 5569$ parameters.
For a training set of, say, 10,000 loan applications, the ratio $P/n = 0.56$. This is already in overfit-risk territory; we will need regularization (Section 6). Compare with a deeper network: 20 input, three hidden layers of 128, output 1: $(128 \cdot 20 + 128) + 2(128 \cdot 128 + 128) + (128 + 1) = 2688 + 33024 + 129 = 35841$. With 10k samples, $P/n = 3.58$—deeply in overfit territory.

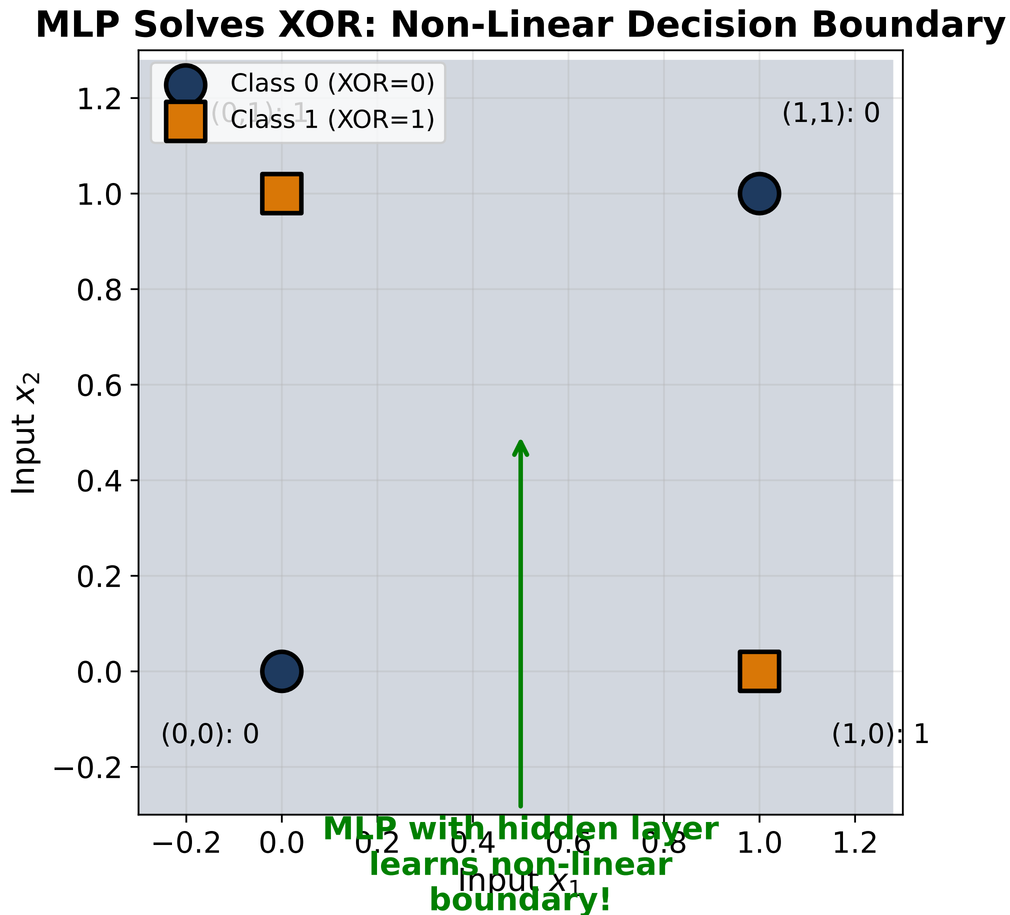

Figure 34: The MLP's solution to XOR. The hidden layer warps the 2D input plane into a 2D feature plane where the classes become linearly separable.

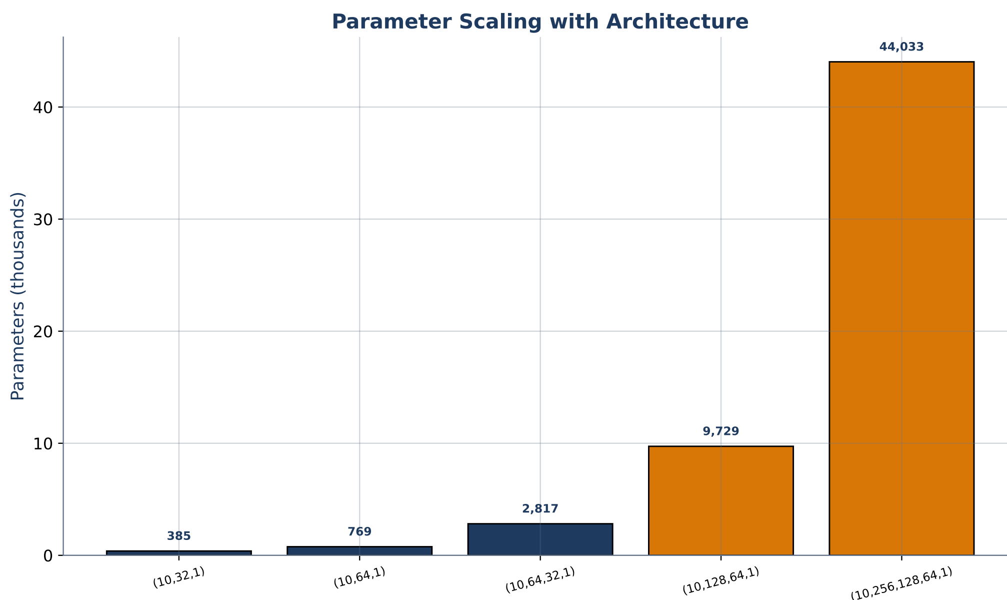

Figure 35: Parameter count scales quadratically with width and linearly with depth. A 10-layer MLP with 256 neurons each has roughly $10 \cdot (256^2 + 256) \approx 660$k parameters.

(c) Joerg Osterrieder 2026

Historical Background: Hornik, Cybenko, and Universal Approximation (1989–1991)

In 1989 George Cybenko at the University of Illinois published "Approximation by superpositions of a sigmoidal function" in *Mathematics of Control, Signals, and Systems*. The paper proved that a neural network with one hidden layer of sigmoidal units and a linear output layer can approximate any continuous function on a compact domain to arbitrary accuracy, provided the number of hidden units is allowed to grow.

Two years later, Kurt Hornik at TU Wien generalized the result. His 1991 paper in *Neural Networks* showed that the specific choice of activation does not matter much: any continuous, non-constant, bounded, monotonically increasing function works. Hornik's proof was cleaner and more general than Cybenko's, and it is the version most commonly cited today.

What these theorems said, and what they did *not* say, shaped a generation of research. They said: representation is not the bottleneck. They did not say: training will be easy, or deep is better than wide, or SGD will find the global optimum. Those practical questions remained open for another two decades. The 2012 ImageNet breakthrough showed that, with enough compute and data, deep networks could be trained in practice—but the theoretical guarantee that they could in principle represent anything goes back to Cybenko and Hornik.

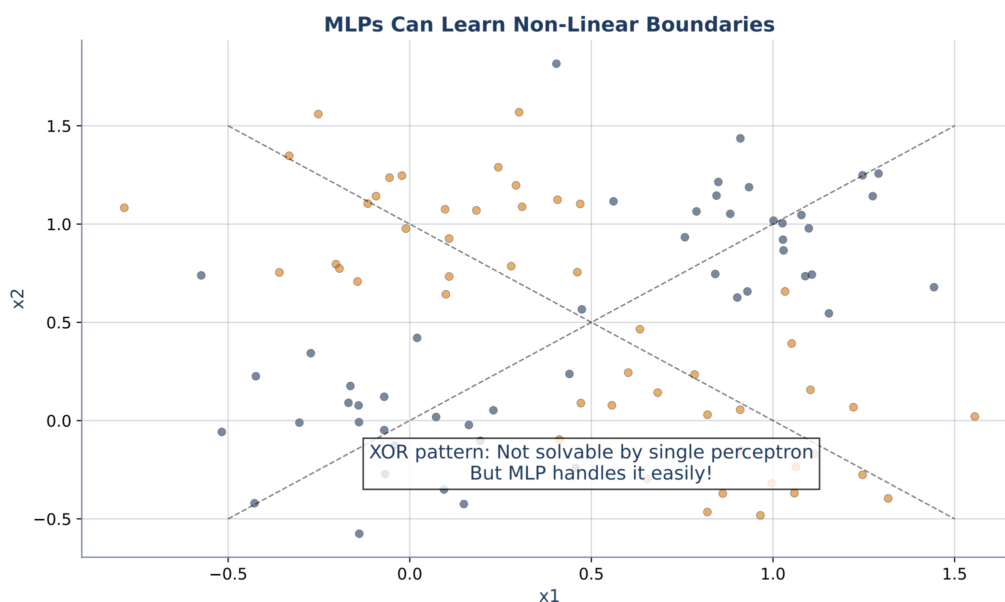

Figure 36: MLPs can learn to match a complicated target function. As the hidden layer widens, the approximation tightens. With 100 hidden neurons, the fit is essentially perfect.

Problem 3.1 (Easy) ⋆

An MLP has input size 10, one hidden layer of 32 ReLU neurons, and output size 1 (sigmoid). Count the total trainable parameters.
Solution: see Appendix.

(c) Joerg Osterrieder 2026

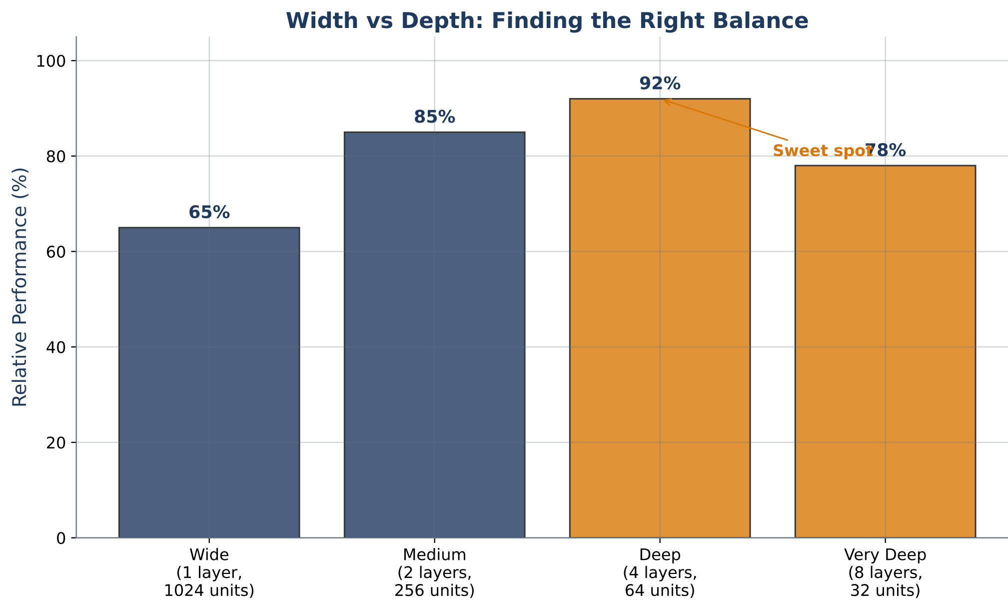

Figure 37: Practical implications of universal approximation. Representational power is there; the art is finding weights that actually achieve it within a reasonable training budget.

Problem 3.2 (Easy) ⋆

State three reasons why ReLU is preferred over sigmoid for hidden layers in deep networks.
Solution: see Appendix.

Problem 3.3 (Medium) ⋆⋆

Compute the output of softmax applied to the vector $(2.0, 1.0, -1.0)$. Show each step. What probability does each class receive?
Solution: see Appendix.

Problem 3.4 (Medium) ⋆⋆

A three-layer MLP has weights W_1, W_2, W_3, biases b_1, b_2, b_3, and uses the identity function as its activation at every layer (no nonlinearity). Write the network's output as a single linear expression $Wx + b$. What does this imply about the role of activation functions?
Solution: see Appendix.

Problem 3.5 (Hard) ⋆⋆⋆

Prove that a one-hidden-layer MLP with ReLU activations can exactly represent any continuous piecewise-linear function on $\mathbb{R}$ with K breakpoints, using at most $K+1$ hidden neurons. (*Hint:* Express the target as a sum of shifted ReLU functions.) Sketch how the representation generalizes to $\mathbb{R}^n$ with linear hyperplane breakpoints.
Solution: see Appendix.

Connecting Forward

We can now write down an MLP, count its parameters, and understand why ReLU activations enable deep networks. But we have not explained how the weights are found. Section 4 introduces

loss functions (how to measure error) and backpropagation (how to compute the gradient of that error with respect to every weight in the network). Once we can compute gradients, we can apply stochastic gradient descent and actually train the networks we have been specifying.

Key Takeaway: An MLP is a composition of linear-then-nonlinear layers; with enough hidden neurons and a nonlinear activation (ReLU in practice), it can approximate any continuous function—but representational capacity alone does not guarantee successful training.

(c) Joerg Osterrieder 2026

4. Teaching Networks – Loss Functions and Backpropagation

Opening Problem: How Does the Network Know It Is Wrong?

You have built an MLP with 5569 parameters for credit risk classification. You initialize all weights to small random numbers and feed in a training batch of 100 applicants. The network spits out probabilities—mostly around 0.5, since the weights are random. Some applicants who defaulted get predicted probability 0.4 (far from the correct answer of 1); some who paid back get predicted probability 0.6 (also wrong).
Now you need to *improve* these weights. You have 5569 knobs to turn. Some tiny adjustment in $W_1[3, 7]$ might reduce the error on defaulter 42 but increase the error on non-defaulter 83. You cannot try every combination—that is $\mathbb{R}^{5569}$ of possibilities. You need a procedure that, for each knob, tells you: "turn this one up, turn that one down, by exactly this much." That procedure is gradient descent, and the tool that computes the gradients efficiently is backpropagation. Section 4 develops both from first principles.

Discovery Question

You are standing on a hilly landscape in thick fog. You cannot see the bottom of the valley, but you can feel the slope under your feet. What local procedure would get you to the bottom? Now imagine the landscape has 5569 dimensions instead of 2. Does your procedure still work? What could go wrong?

Loss Functions: Measuring Error

Before we can improve the weights, we need a numerical score for "how wrong is the network." That score is the *loss function* $L(\theta)$, where θ collects all weights and biases. Training consists of minimizing L over θ.

Loss function: A scalar function that measures the discrepancy between predictions and true labels. Lower loss means better predictions. The choice of loss depends on the task: MSE for regression, cross-entropy for classification.

Regression: Mean Squared Error (MSE). For true values y_i and predictions $\hat{y}_i$, with N examples:

$$L_{\text{MSE}}(\theta) = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2.$$

MSE is differentiable, interpretable (it is the variance of residuals when $\hat{y} = \bar{y}$), and has the important property that the optimal constant predictor is the mean. Absolute error (MAE) and Huber loss are alternatives that are more robust to outliers.

Binary classification: Binary Cross-Entropy (BCE), a.k.a. log loss.

$$L_{\text{BCE}}(\theta) = -\frac{1}{N}\sum_{i=1}^{N}\left[y_i \log \hat{y}_i + (1 - y_i)\log(1 - \hat{y}_i)\right].$$

BCE has a maximum likelihood interpretation: minimizing it is equivalent to maximizing the likelihood under a Bernoulli model. It is the loss paired with a sigmoid output layer.

Multiclass classification: Categorical Cross-Entropy (CCE).

$$L_{\text{CCE}}(\theta) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{K} y_{i,k}\log \hat{y}_{i,k},$$

where $y_{i,k}$ is the one-hot encoding of the true class and $\hat{y}_{i,k}$ is the softmax output for class k.

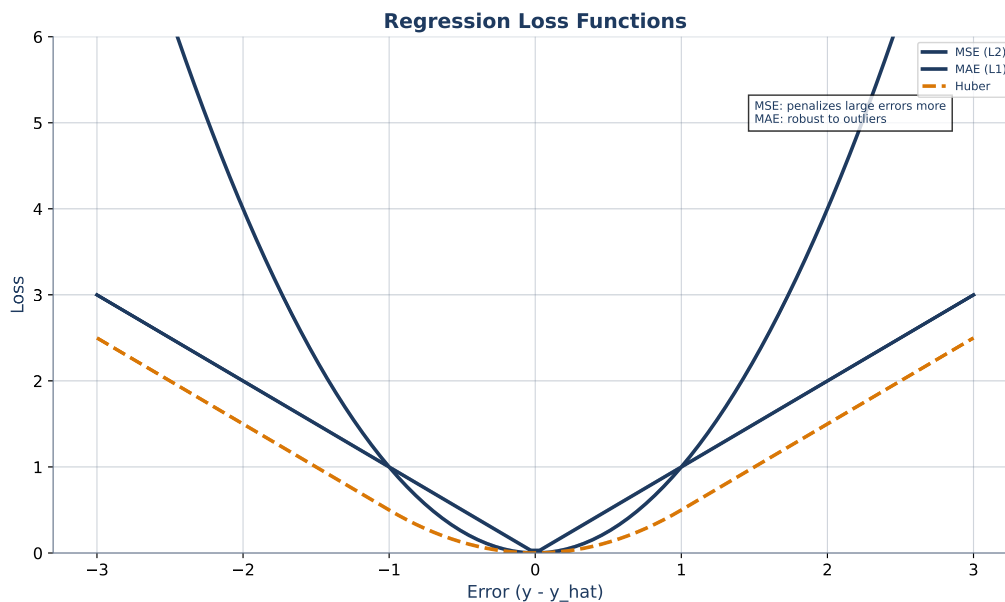

Figure 38: Three regression losses compared: MSE (quadratic), MAE (linear), and Huber (quadratic near zero, linear in the tails). MSE punishes large errors more aggressively; MAE is robust to outliers.

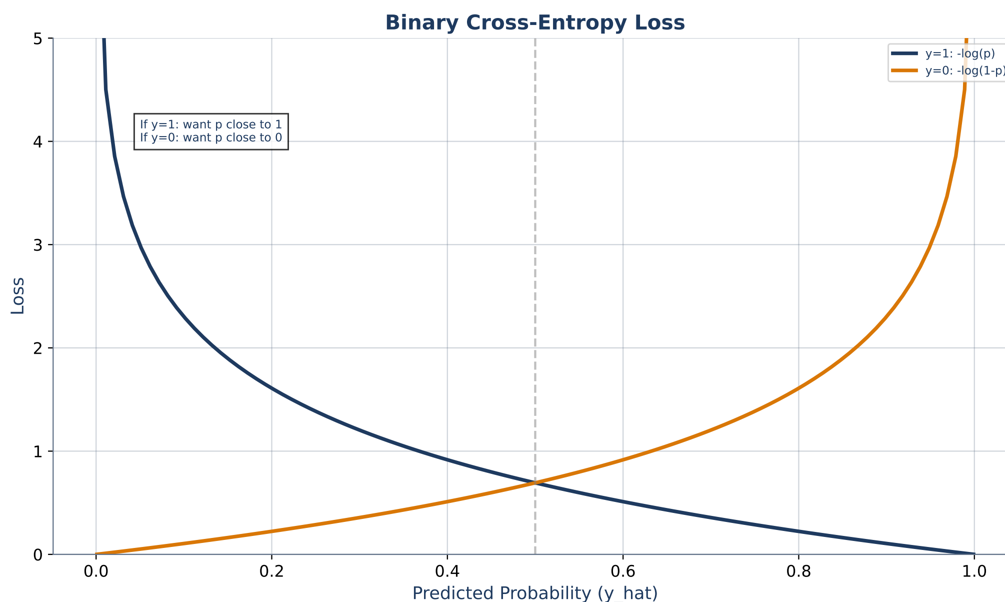

Figure 39: Binary cross-entropy as a function of the predicted probability when the true label is 1. Loss is zero when $\hat{y} = 1$ and blows up to $+\infty$ as $\hat{y} \to 0$. Confident wrong answers are punished severely.

Key Formula: Matching Loss to Output Activation

Task	Output activation	Loss
Regression	Identity	Mean Squared Error (MSE)
Binary classification	Sigmoid	Binary Cross-Entropy (BCE)
Multiclass classification	Softmax	Categorical Cross-Entropy (CCE)
Count regression	Exp/Softplus	Poisson Negative Log-Likelihood

Plain English: the loss should match the probabilistic assumption behind the output. Mismatches (like MSE with sigmoid) train slowly or not at all.

Gradient Descent: Rolling Downhill

With a differentiable loss, we can use gradient descent. For a single parameter w, the gradient $\partial L/\partial w$ tells us the direction of steepest increase. To decrease L, move w in the opposite direction:

$$w \leftarrow w - \eta \frac{\partial L}{\partial w},$$

where $\eta > 0$ is the **learning rate**. Apply the same rule to every parameter in θ simultaneously.

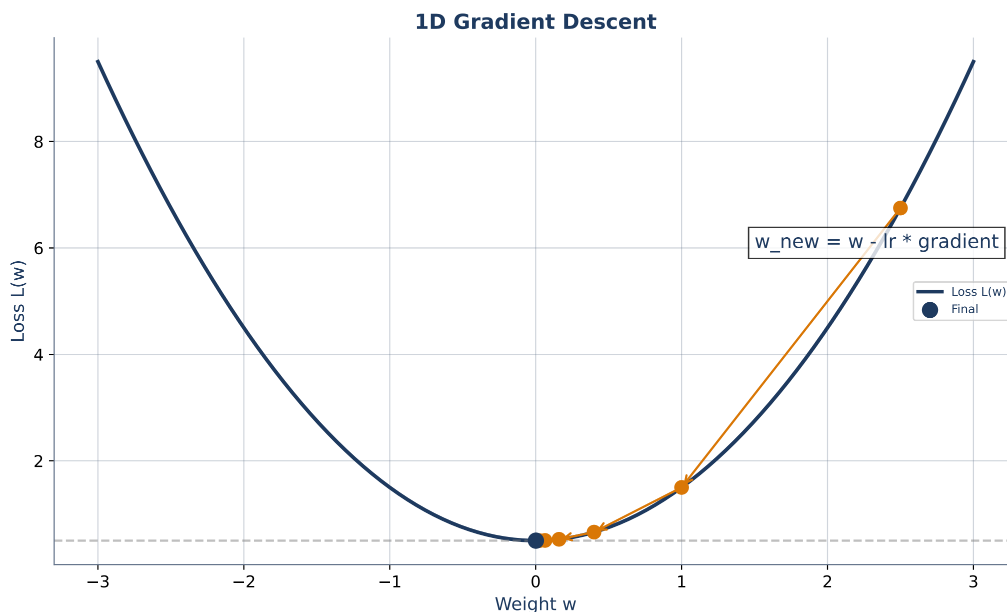

Figure 40: Gradient descent in one dimension. Starting from any point, follow the negative gradient to the nearest minimum. For convex loss, this minimum is global; for neural networks, only local.

(c) Joerg Osterrieder 2026

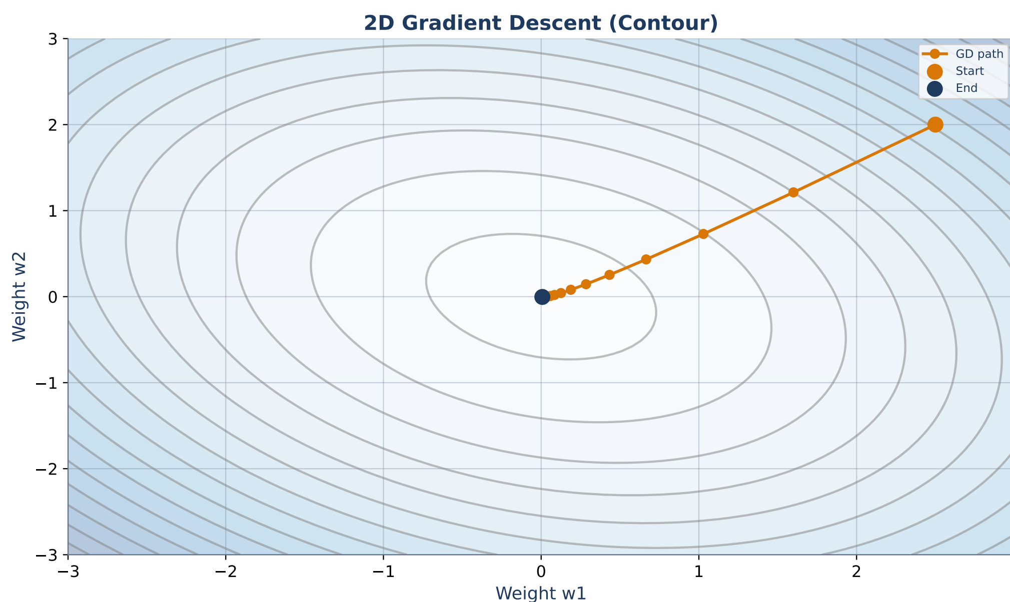

Figure 41: Gradient descent in two dimensions on a quadratic bowl. The trajectory zig-zags when the loss surface is elongated; momentum-based optimizers (below) dampen this oscillation.

Key Formula: Gradient Descent Update

For parameter vector θ and loss L:

$$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla_\theta L(\theta^{(t)}).$$

Here $\nabla_\theta L$ is the vector of partial derivatives; η is the learning rate.

- η too small: convergence is painfully slow.
- η too large: updates overshoot, loss diverges to infinity.
- Typical values: 10^{-4} to 10^{-2} for Adam; 10^{-3} to 10^{-1} for SGD.

The Chain Rule, Applied to a Composition

To compute $\partial L/\partial w$ for an intermediate weight w in a deep network, we need the chain rule. For a composition $f \circ g$, the chain rule says:

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

Extended to a chain of functions $L = \ell \circ f_L \circ f_{L-1} \circ \cdots \circ f_1$ with intermediate variables $h^{(1)}, \ldots, h^{(L)}$:

$$\frac{\partial L}{\partial h^{(\ell)}} = \frac{\partial L}{\partial h^{(L)}} \cdot \prod_{k=\ell+1}^{L} \frac{\partial h^{(k)}}{\partial h^{(k-1)}}.$$

Each factor $\partial h^{(k)}/\partial h^{(k-1)}$ is a Jacobian of layer k with respect to its input. Multiplying these Jacobians from output back to input is backpropagation.

Backpropagation: Efficient Gradient Computation

Naively, computing $\partial L/\partial w$ for every weight via finite differences would cost one forward pass per weight—billions of passes for a modern network. Backpropagation computes *all* gradients in

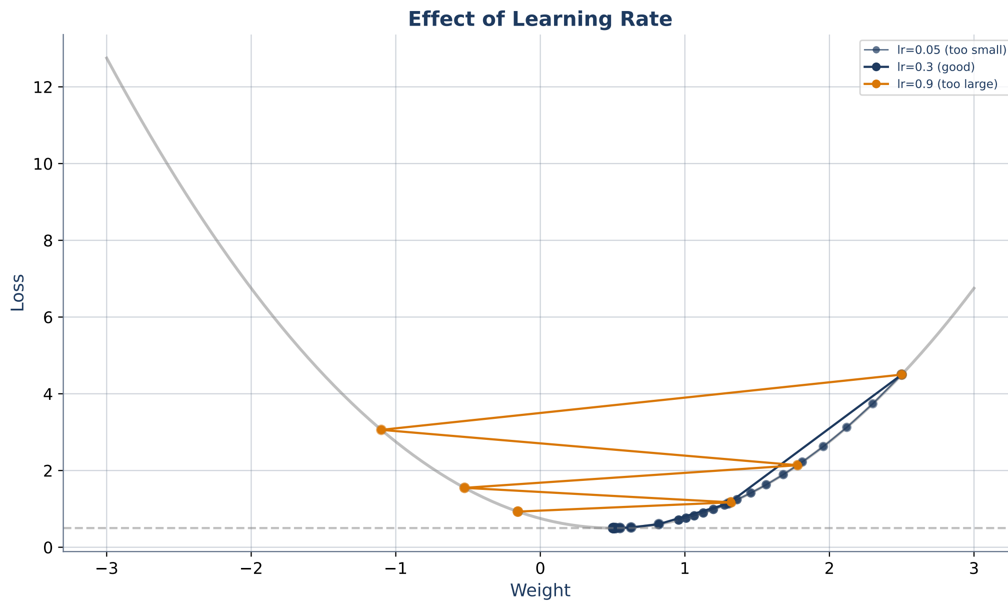

Figure 42: Effect of learning rate. Too small: stuck in molasses. Too large: the loss diverges. Just right: smooth, monotonic descent.

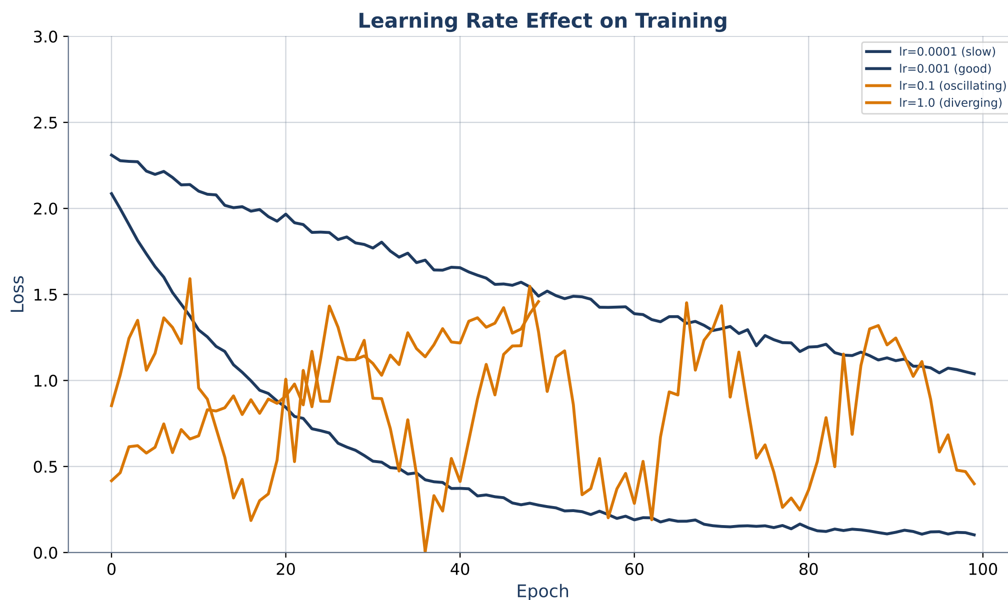

Figure 43: Loss curves at three different learning rates. A 10× larger learning rate can accelerate convergence by 10× or derail training entirely.

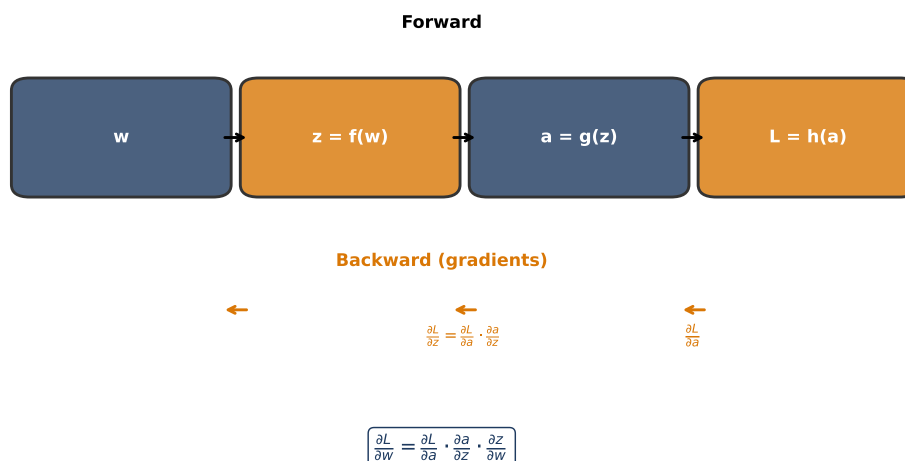

Figure 44: The chain rule visualized. Gradients flow backward through the computation graph, multiplying local Jacobians at each node.

essentially two passes: one forward, one backward. Its complexity is $O(P)$ rather than $O(P^2)$, which is the difference between "possible" and "impossible."

Definition: Backpropagation

Backpropagation is an algorithm for computing $\nabla_\theta L$ for a feedforward neural network. It has two phases:
Forward pass: compute and cache $z^{(\ell)} = W_\ell h^{(\ell-1)} + b_\ell$ and $h^{(\ell)} = \sigma(z^{(\ell)})$ for every layer $\ell = 1, \dots, L$.
Backward pass: Initialize $\delta^{(L)} = \partial L / \partial z^{(L)}$. For $\ell = L-1, L-2, \dots, 1$, compute

$$\delta^{(\ell)} = (W_{\ell+1}^\top \delta^{(\ell+1)}) \odot \sigma'(z^{(\ell)}),$$

where $\odot$ is element-wise multiplication. Then the gradients are:

$$\frac{\partial L}{\partial W_\ell} = \delta^{(\ell)} (h^{(\ell-1)})^\top, \qquad \frac{\partial L}{\partial b_\ell} = \delta^{(\ell)}.$$

The total cost is roughly $2\times$ the forward pass—a remarkable efficiency.

SGD, Mini-Batches, and Full-Batch

Computing the loss exactly requires a forward pass over the entire training set. For 100 million training examples, that is prohibitive. Three variants of gradient descent differ in how much data they use per step.

Batch (full-batch) gradient descent: compute the gradient using all N training examples, take one step. Deterministic and stable but slow; one step per epoch.

Stochastic gradient descent (SGD): pick one random example, compute its gradient, take a step. Very fast per step but very noisy. The noise has an unexpected benefit: it helps escape shallow local minima and saddle points.

Mini-batch SGD: pick a random subset of size B (the "batch size"), compute the mean gradient over the batch, take a step. Standard practice: B between 32 and 512. Combines

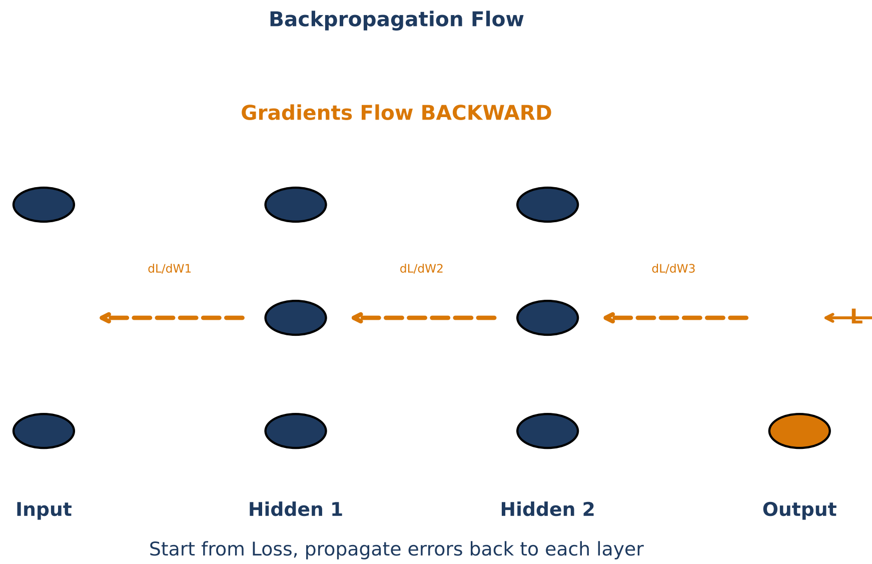

Figure 45: Backpropagation flow. Forward pass (blue) computes activations; backward pass (orange) computes gradients from output to input by repeated Jacobian multiplication.

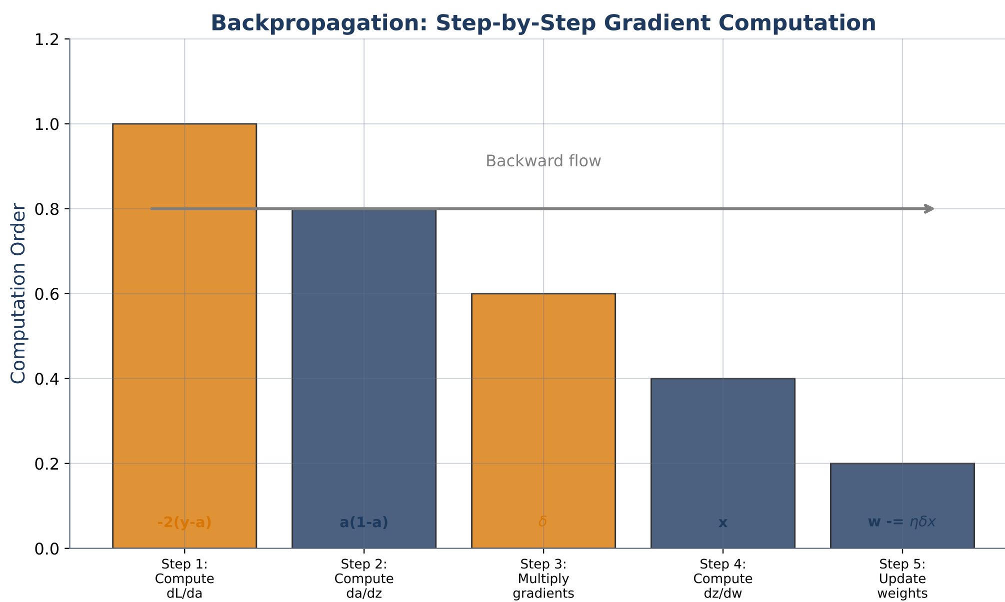

Figure 46: Backpropagation step by step. At each layer, the gradient is a product of the incoming error signal and the local derivative.

(c) Joerg Osterrieder 2026

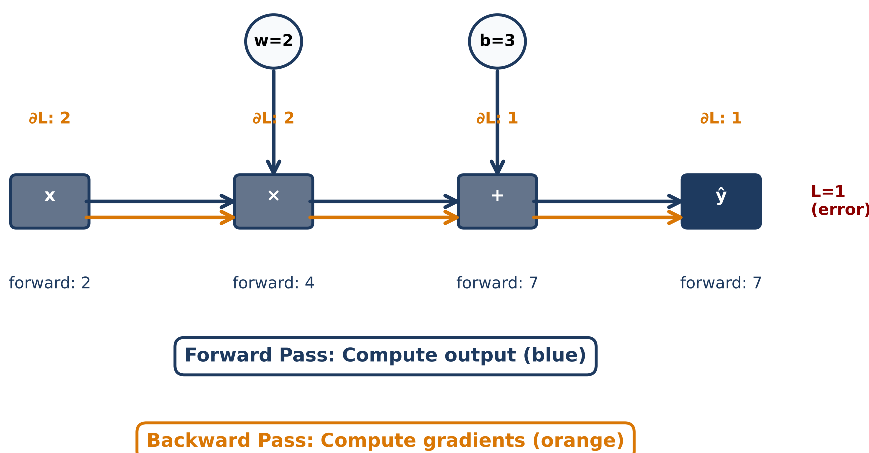

Figure 47: The computation graph of a small network. Backprop traverses this graph in reverse topological order, accumulating gradients.

SGD's exploration with batch's stability.

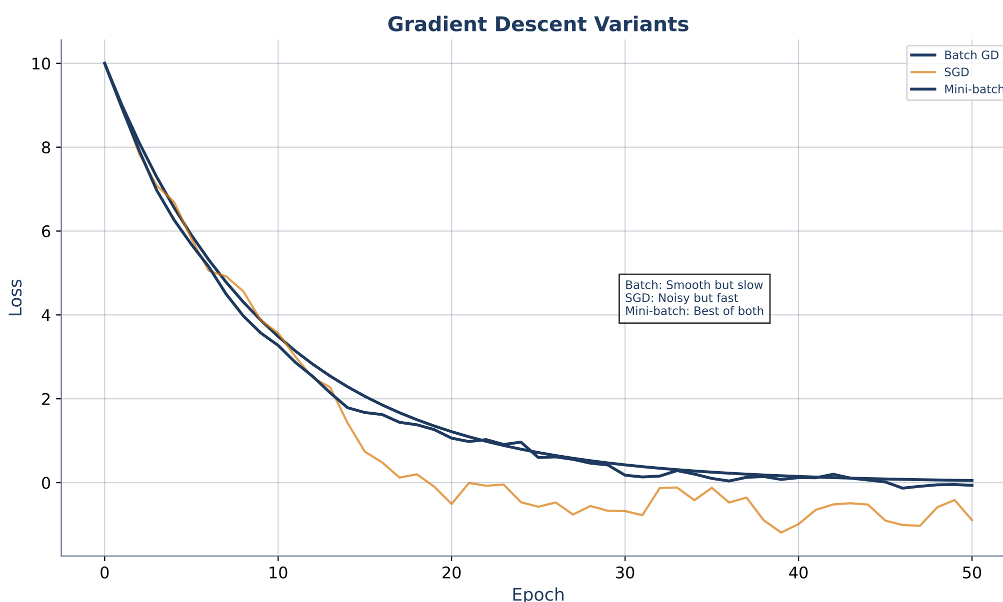

Figure 48: Batch vs. mini-batch vs. stochastic gradient descent. Full batch gives the true gradient but is slow. SGD is noisy but fast. Mini-batch is the standard compromise.

Momentum, Adam, and Modern Optimizers

Plain SGD is slow on loss surfaces that are ill-conditioned—stretched ellipses rather than round bowls. Momentum-based optimizers keep a running average of past gradients:

$$v^{(t+1)} = \beta v^{(t)} + (1-\beta)\nabla L(\theta^{(t)}), \qquad \theta^{(t+1)} = \theta^{(t)} - \eta v^{(t+1)}.$$

With $\beta \approx 0.9$, momentum smooths noisy gradients and accelerates in consistently descending directions.

(c) Joerg Osterrieder 2026

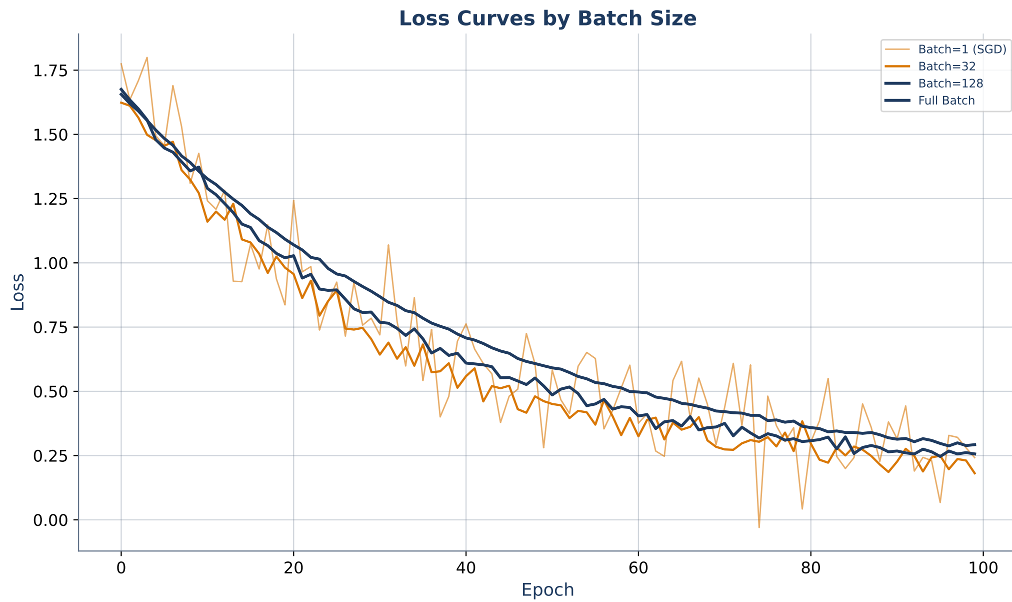

Figure 49: Loss curves for different batch sizes. Smaller batches show more noise per step but often generalize better; larger batches are smoother but can get stuck in sharp minima.

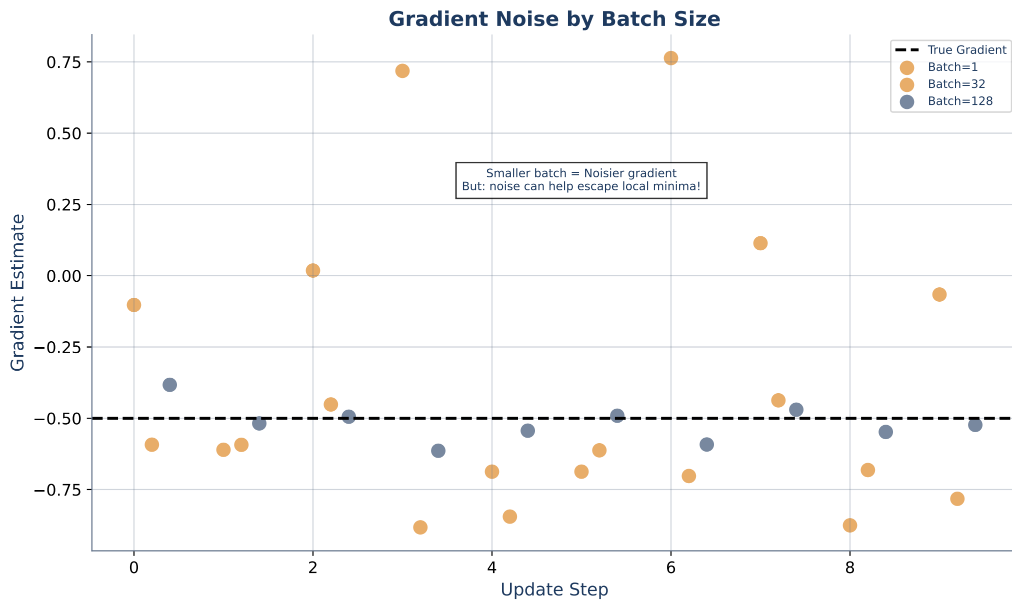

Figure 50: Gradient noise scales like $1/\sqrt{B}$ where B is batch size. Doubling the batch halves the variance of the gradient estimate.

Adam (Kingma and Ba, 2014) combines momentum with per-parameter adaptive learning rates:

$$m^{(t)} = \beta_1 m^{(t-1)} + (1-\beta_1)g^{(t)}, \qquad v^{(t)} = \beta_2 v^{(t-1)} + (1-\beta_2)(g^{(t)})^2,$$

$$\theta^{(t+1)} = \theta^{(t)} - \eta \frac{\hat{m}^{(t)}}{\sqrt{\hat{v}^{(t)}} + \epsilon},$$

where $\hat{m}, \hat{v}$ are bias-corrected versions and $g^{(t)} = \nabla L(\theta^{(t)})$. Typical hyperparameters: $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$.

Adam works well out of the box, which is why it is often the default. For convex problems or when asymptotic convergence matters, plain SGD with momentum can generalize slightly better, but Adam's robustness wins most practical shootouts.

Common Misconceptions about Loss and Backprop

(1) "Backprop is the same as gradient descent." Backprop computes gradients; gradient descent uses them to update weights. They are distinct. You can use the gradients computed by backprop with Adam, RMSprop, or any other optimizer.
(2) "MSE is fine for classification." No. Pairing MSE with a sigmoid output creates a non-convex loss with flat regions where gradients vanish. Use BCE (or CCE for multiclass)—the resulting loss has a single minimum and well-behaved gradients.
(3) "A larger batch always trains faster." Per step, yes. Per epoch, also faster. But larger batches sometimes hurt generalization by converging to sharp minima. Standard practice: batch size 32–256, linear-scale the learning rate with batch size.

Worked Examples

Worked Example 1: Backprop Through a 2-2-1 Network by Hand

A tiny network: 2 inputs, 2 hidden neurons with sigmoid, 1 output with sigmoid. Weights:

$$W_1 = \begin{pmatrix} 0.5 & 0.1 \\ -0.3 & 0.8 \end{pmatrix}, b_1 = \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \quad W_2 = (0.4, -0.6), b_2 = 0.2.$$

Input: $x = (1, 0.5)$. True label: $y = 1$. Loss: BCE.
Forward pass.

$$z_1^{(1)} = 0.5 \cdot 1 + 0.1 \cdot 0.5 = 0.55, \quad h_1^{(1)} = \sigma(0.55) \approx 0.634.$$

$$z_2^{(1)} = -0.3 \cdot 1 + 0.8 \cdot 0.5 = 0.1, \quad h_2^{(1)} = \sigma(0.1) \approx 0.525.$$

$$z^{(2)} = 0.4 \cdot 0.634 - 0.6 \cdot 0.525 + 0.2 = 0.254 - 0.315 + 0.2 = 0.139.$$

$$\hat{y} = \sigma(0.139) \approx 0.535.$$

BCE loss: $L = -\log(0.535) \approx 0.625$.
Backward pass. $\dfrac{\partial L}{\partial z^{(2)}} = \hat{y} - y = 0.535 - 1 = -0.465.$
Output gradients:

$$\frac{\partial L}{\partial W_2} = \delta^{(2)} \cdot (h^{(1)})^\top = (-0.465)(0.634, 0.525) = (-0.295, -0.244).$$

$$\frac{\partial L}{\partial b_2} = -0.465.$$

Backprop to hidden: $\delta^{(1)} = W_2^\top \delta^{(2)} \odot \sigma'(z^{(1)})$.

$$W_2^\top \delta^{(2)} = (0.4 \cdot -0.465, -0.6 \cdot -0.465) = (-0.186, 0.279).$$

$$\sigma'(0.55) = 0.634(1 - 0.634) = 0.232. \qquad \sigma'(0.1) = 0.525(1 - 0.525) = 0.249.$$

$$\delta^{(1)} = (-0.186 \cdot 0.232, 0.279 \cdot 0.249) = (-0.0432, 0.0696).$$

Hidden gradients:

$$\frac{\partial L}{\partial W_1} = \delta^{(1)} x^\top = \begin{pmatrix} -0.0432 \cdot 1 & -0.0432 \cdot 0.5 \\ 0.0696 \cdot 1 & 0.0696 \cdot 0.5 \end{pmatrix} = \begin{pmatrix} -0.043 & -0.022 \\ 0.070 & 0.035 \end{pmatrix}.$$

$$\frac{\partial L}{\partial b_1} = (-0.0432, 0.0696).$$

With $\eta = 0.1$, the updated weights move slightly in the direction that reduces BCE loss on this one example.

(c) Joerg Osterrieder 2026

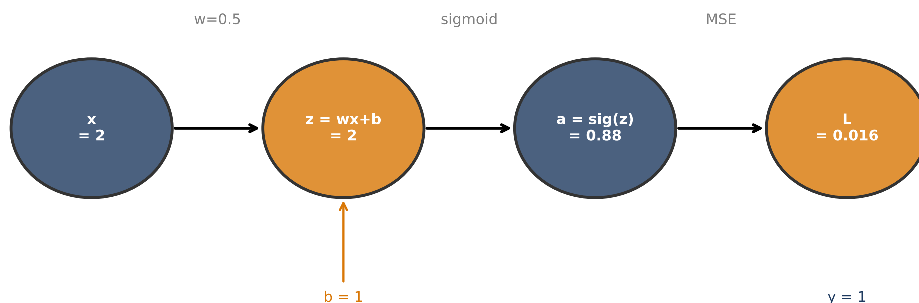

Figure 51: Numerical backpropagation example. All intermediate quantities are shown explicitly to make the computation concrete.

Worked Example 2: Return Prediction on S&P 500

A finance researcher trains an MLP to predict tomorrow's S&P 500 return from today's features (volatility, momentum, VIX level). The target is a small number—say, +0.005 or −0.003. The researcher uses MSE.

Early in training, the network outputs roughly zero. MSE contribution from an example with $y_i = 0.005$, $\hat{y}_i = 0.000$ is $(0.005)^2 = 2.5 \times 10^{-5}$. The loss number looks tiny, but the gradient pointing away from zero is $2(y_i - \hat{y}_i) = 0.01$—large in relative terms. Training progresses. After 50 epochs, the network predicts $\hat{y}_i \approx 0.003$ for this example. New MSE contribution: $(0.002)^2 = 4 \times 10^{-6}$. Gradient: 0.004. Smaller.

Because returns are small, MSE values are small too—but it is the *relative* fit that matters. Finance practitioners often report R^2 or directional accuracy instead of raw MSE for this reason. The bigger pitfall: stock returns are very noisy (SNR ≈ 0.01), so even a well-trained network will have $R^2 \approx 0.05$—and that is an outstanding result in practice.

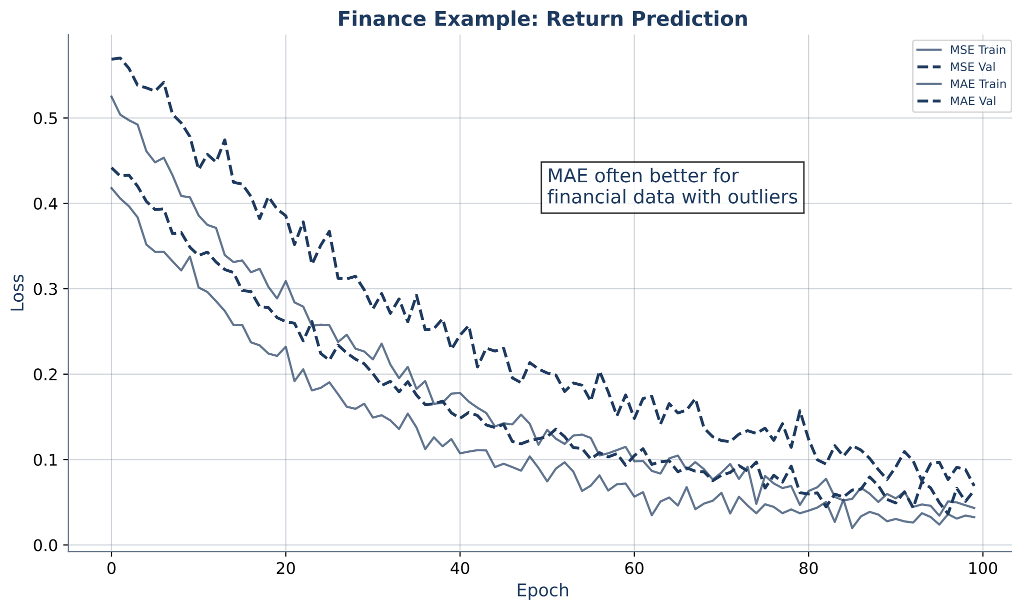

Figure 52: Return-prediction MLP. Predictions are a small, noisy signal; directional accuracy is often more informative than raw MSE for evaluating finance models.

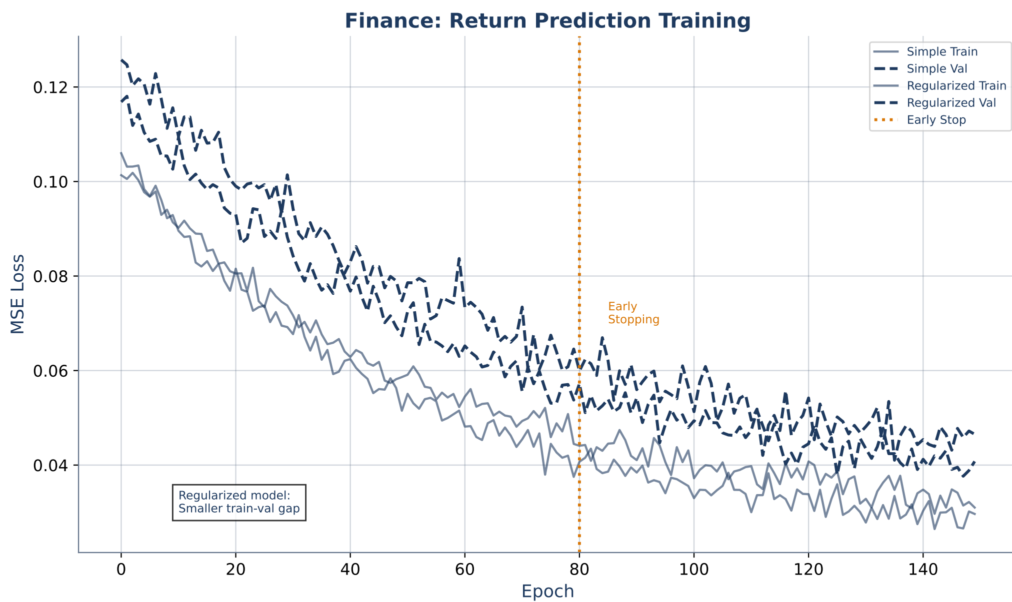

Figure 53: Training and validation loss for return prediction. The small validation-training gap is a good sign; wide gaps would indicate overfitting.

Historical Background: Rumelhart, Hinton, and Williams (1986)

Backpropagation was independently discovered several times. Paul Werbos proposed the method in his 1974 PhD thesis at Harvard, applying it to economic forecasting. The idea drew on earlier work in optimal control (Kelley 1960, Bryson 1961) and automatic differentiation. But Werbos's work was overlooked by the neural network community for more than a decade.

In 1986, David Rumelhart, Geoffrey Hinton, and Ronald Williams published "Learning representations by back-propagating errors" in *Nature*. They rediscovered backpropagation in the context of neural networks and demonstrated that it could train multi-layer perceptrons to solve problems Minsky and Papert's 1969 book had declared off-limits. The paper used XOR as a pedagogical example and showed backprop solving it in a handful of iterations.

The 1986 paper ended the first AI winter. Neural network research returned to respectability in the late 1980s and early 1990s. The real explosion waited for another 26 years: 2012, when AlexNet won the ImageNet competition with a convolutional neural network trained by backprop on two GPUs. That was the moment backpropagation graduated from "clever trick" to "workhorse of modern AI."

Problem 4.1 (Easy) ⋆

Compute the MSE loss for predictions $\hat{y} = (2.1, 3.9, 5.2)$ and true values $y = (2.0, 4.0, 5.0)$.
Solution: see Appendix.

Problem 4.2 (Easy) ⋆

The binary cross-entropy loss for one example is $-y \log \hat{y} - (1 - y) \log(1 - \hat{y})$. Compute the loss when $y = 1$ and $\hat{y} = 0.9$. Compute it again for $\hat{y} = 0.1$. Why is the second loss so much larger?
Solution: see Appendix.

Problem 4.3 (Medium) ⋆⋆

A single sigmoid output neuron has weight $w = 2.0$, bias $b = -1.0$, and receives input $x = 0.5$. True label $y = 1$. BCE loss. Compute the gradient of the loss with respect to w and b. Apply one gradient-descent update with $\eta = 0.5$.
Solution: see Appendix.

Problem 4.4 (Medium) ⋆⋆

Explain why Adam often trains faster than plain SGD on ill-conditioned loss surfaces. What specific mechanisms in the Adam update help? When might vanilla SGD be preferable?
Solution: see Appendix.

Problem 4.5 (Hard) ⋆⋆⋆

Derive the backpropagation update equations for a 3-layer MLP with sigmoid activations in the hidden layer and a sigmoid output unit, using BCE loss. Show that the output-layer gradient simplifies to $\hat{y} - y$ when using the BCE + sigmoid combination.
Solution: see Appendix.

Connecting Forward

With forward propagation, a differentiable loss, and backpropagation to compute gradients, we can train any MLP on any differentiable objective. Section 5 asks the next question: we have a procedure that drives training loss to near zero; but does that mean the model generalizes? We will see that deep networks are acutely prone to overfitting, introduce the bias-variance decomposition in this context, and set up Section 6's treatment of regularization.

Key Takeaway: Backpropagation computes gradients efficiently by traversing the computation graph backward once per step; gradient descent then uses those gradients to improve weights—these two ingredients train every deep network in use today.

(c) Joerg Osterrieder 2026

5. Why Deep Networks Overfit

Opening Problem: The Beautiful Backtest That Bankrupted the Fund

A quantitative research team at a small hedge fund trains an MLP with three hidden layers of 256 neurons each to predict daily directional returns for 50 S&P 500 stocks. They use five years of daily features—technical indicators, microstructure statistics, macro variables—and get a training accuracy of 78%. The backtest looks glorious: Sharpe ratio 3.2, maximum drawdown 4%, win rate 62%. The partners decide to allocate $50 million. In the first week of live trading, the strategy loses 2%. In the second week, another 3%. By the end of the month, it is down 11%, the Sharpe ratio has collapsed to −0.5, and the partners pull the plug. An autopsy reveals the culprit: the model had 450,000 parameters and was trained on roughly 60,000 daily observations (50 stocks times five years of 252 trading days). Ratio of parameters to data: 7.5 to 1. The network had effectively memorized the training set. In-sample fit was spectacular; out-of-sample behavior was indistinguishable from random.

This section diagnoses exactly what happened. We revisit the bias-variance tradeoff in the context of deep networks, quantify why deep networks are uniquely prone to overfit, and set up the regularization techniques of Section 6.

Discovery Question

You train a network and plot training loss vs. epoch. It falls smoothly toward zero. You conclude: the network is working. But your mentor asks, "What about validation loss?" You plot it too. For the first 10 epochs, validation loss falls with training loss. After epoch 20, validation loss starts rising while training loss continues to fall. What is happening? At which epoch should you have stopped?

Signs of Overfitting

Overfitting has a distinctive signature: the model fits the training set very well but generalizes poorly. You see it in three places.

1. Training loss vs. validation loss. Both start high. Both fall initially. At some point, the training loss keeps falling while the validation loss flattens and then rises. The gap between them widens. The model is memorizing the training set.

2. Large gap between training and test metrics. Training accuracy 95%, test accuracy 68% is a red flag. Difference of 27 percentage points means the model has learned patterns that do not exist outside the training set.

3. Unstable predictions on small input changes. An overfit model tends to have large weight magnitudes, which translate to huge output swings for small input shifts. A robust model is relatively flat.

Bias-Variance Decomposition, Applied to Deep Networks

The expected test error of a regression model decomposes into three parts:

$$\mathbb{E}[(y-\hat{y})^2] = \underbrace{(\mathbb{E}[\hat{y}]-y)^2}_{\text{bias}^2} + \underbrace{\mathbb{E}[(\hat{y}-\mathbb{E}[\hat{y}])^2]}_{\text{variance}} + \underbrace{\sigma^2_{\text{noise}}}_{\text{irreducible}}.$$

Bias is the error from a systematically wrong model. A linear model fitted to XOR has high bias. *Variance* is the error from the model varying too much across training samples. A deep network trained on a small dataset has high variance. Irreducible noise is the error floor set by the data-generating process.

(c) Joerg Osterrieder 2026

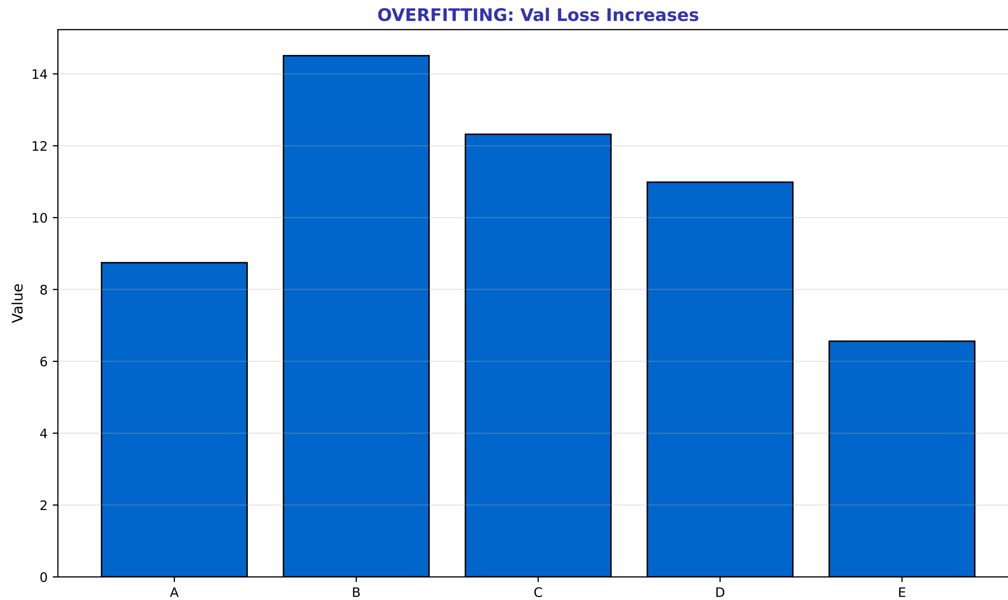

Figure 54: The classic overfitting pattern. Training loss (blue) drops monotonically; validation loss (orange) drops then rises. The crossover is where the model stops generalizing.

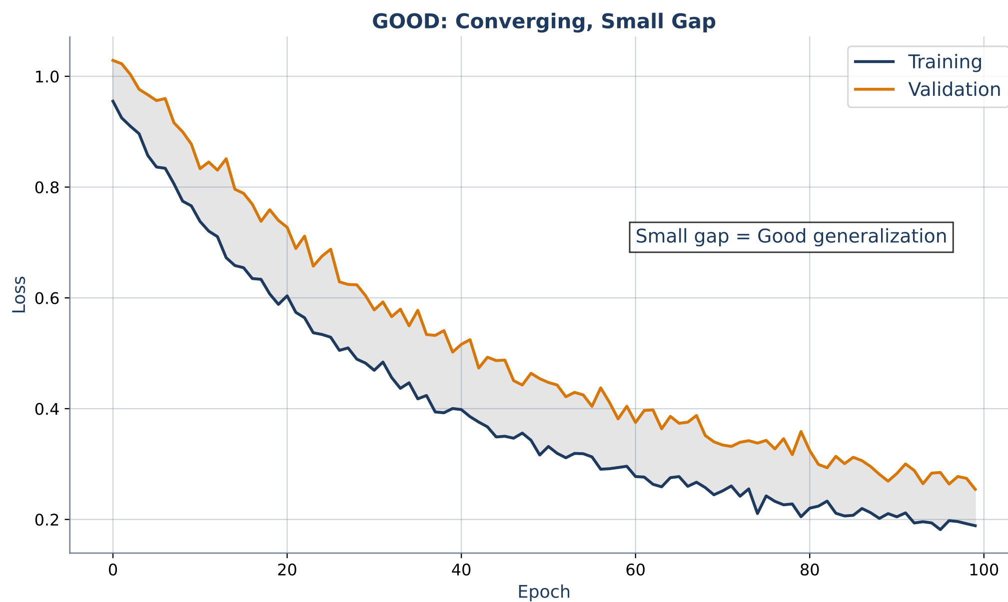

Figure 55: A healthy training curve. Training and validation losses fall together with a small, stable gap between them.

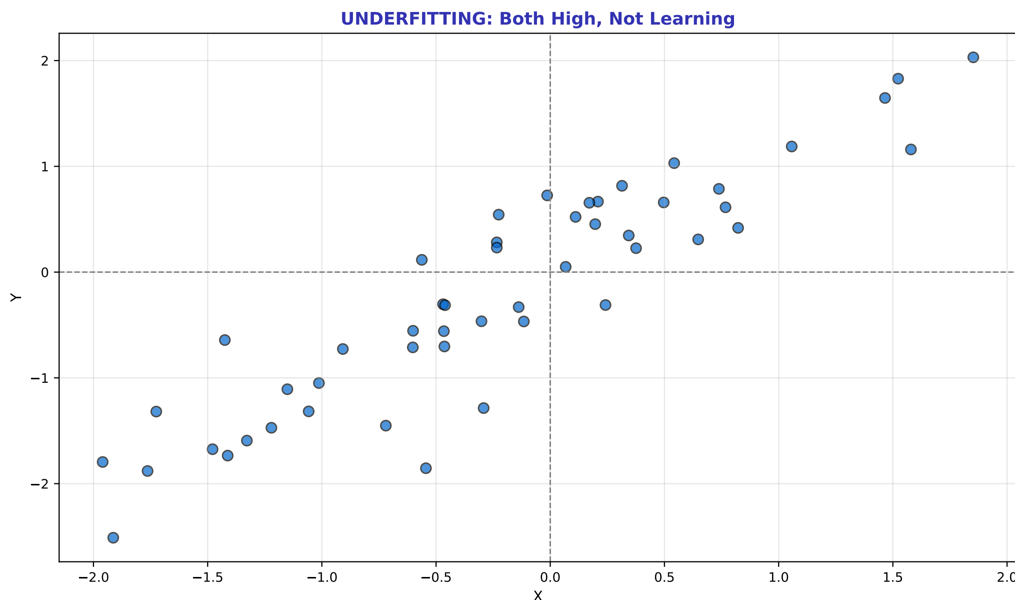

Figure 56: Underfitting. Training loss never drops low; validation loss is similar. The model is too simple for the data, or training has not proceeded long enough.

Overfitting: A situation where a model has low training error but high test error. Formally: the model's variance is high relative to its bias, so it fits random fluctuations of the training set that do not generalize.

Underfitting: The opposite situation: both training and test errors are high. The model is too constrained to capture the true pattern.

Why Deep Networks Are Uniquely Vulnerable

Several properties make deep networks more prone to overfit than classical models.

Massive parameter counts. A modest MLP with 100 input features, three hidden layers of 256 neurons, and a 10-class output has $\approx$ 200k parameters. For 10k training examples, this is 20 parameters per example—an ocean of capacity to memorize noise.

High expressivity. Universal approximation (Section 3) guarantees that, with enough hidden units, the network can represent any continuous function. In particular, it can represent a function that interpolates every training point exactly, with arbitrary values between them.

Non-convex loss. Gradient descent on a deep network can find many different local minima that all have near-zero training loss but very different generalization behavior. The optimization landscape is complex enough that two runs with different initializations can produce different models with different overfit levels.

Feature interactions. The very property that makes deep networks useful—their ability to learn interactions—also makes them over-eager. A network with enough capacity will "discover" spurious interactions that exist only in the training set.

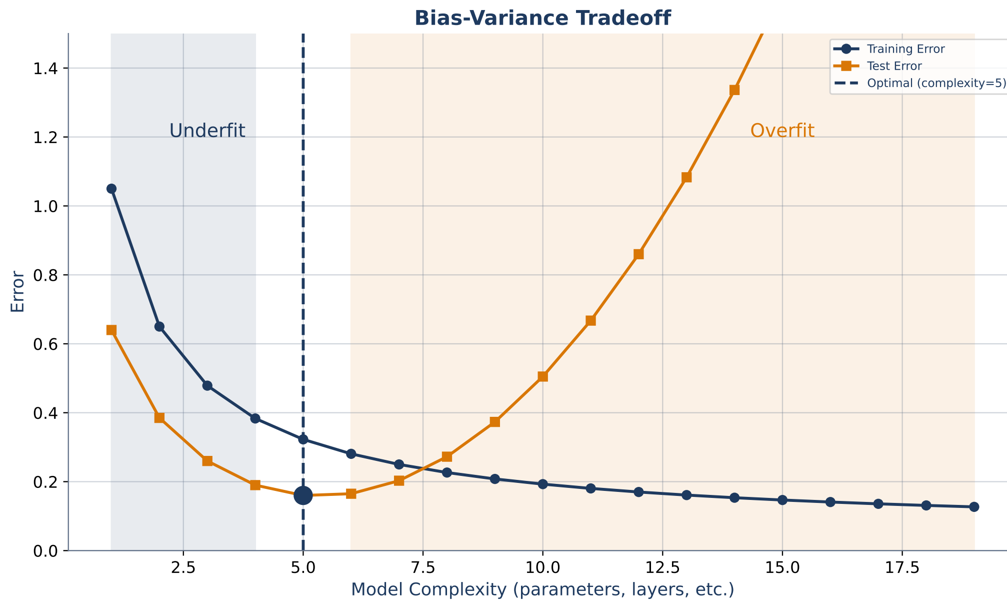

Figure 57: The bias-variance tradeoff. As model capacity grows, bias falls (the model can fit more patterns) but variance rises (the model reacts more to data fluctuations). Total error is minimized at an intermediate capacity.

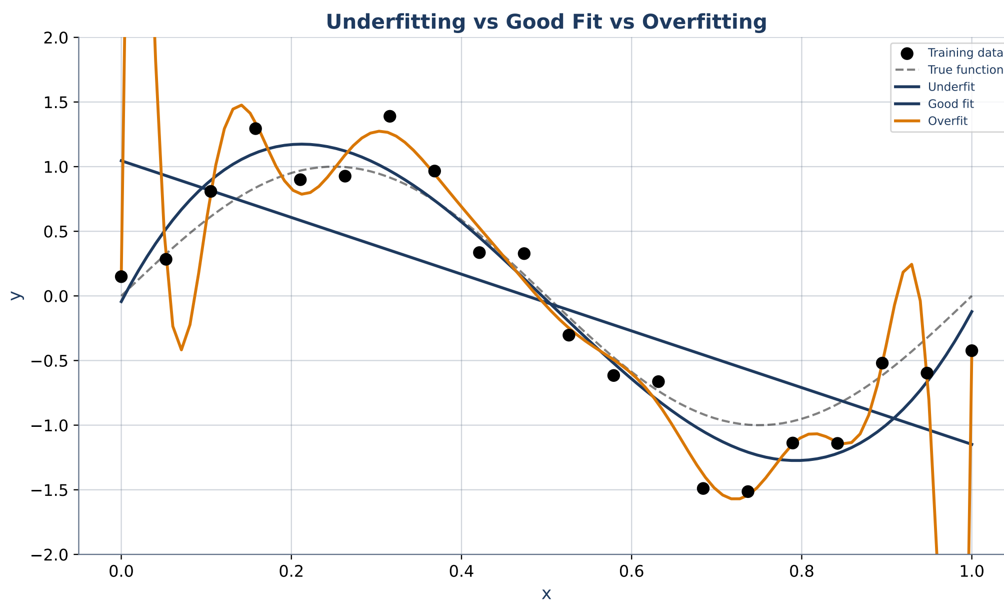

Figure 58: Underfitting (left) vs. good fit (middle) vs. overfitting (right) on a toy regression. The overfit model passes through every training point but swings wildly between them.

(c) Joerg Osterrieder 2026

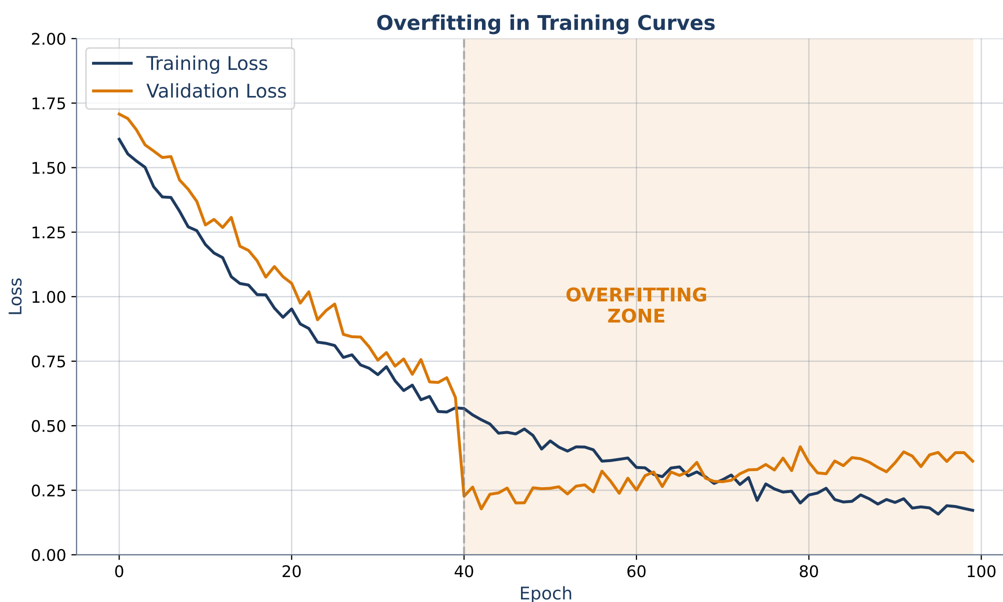

Figure 59: Overfitting visible in the training curves. The diverging train-val gap is the most reliable early warning.

Key Formula: Generalization Bound (Informal)

For a model with P parameters and N training examples, a classical learning-theory bound (Vapnik–Chervonenkis, PAC-Bayes, etc.) gives

$$\text{test error} \lesssim \text{training error} + O\left(\sqrt{\frac{P}{N}}\right).$$

Implication: doubling P with N fixed roughly $\sqrt{2}$-es the generalization gap. Deep networks with $P \gg N$ can have arbitrarily bad bounds—yet in practice many generalize well, which is one of modern deep learning's biggest theoretical puzzles.

Double Descent: A Modern Curiosity

Classical learning theory predicts a U-shaped test-error curve in model capacity: too small and you underfit, too large and you overfit. Belkin et al. (2019) documented a surprising phenomenon: for many deep networks, once the model is large enough to interpolate the training set (zero training error), further increasing capacity *decreases* test error again. This is called “double descent.”

The double-descent curve has three regimes. Below the interpolation threshold, test error follows the classical U-shape. At the threshold (training error = 0), test error spikes. Beyond the threshold, in the over-parameterized regime, test error can drop below even the classical minimum. Exactly why this happens is an active research area; implicit regularization from SGD, benign overfitting, and the structure of modern initialization schemes all play a role.

For practitioners, the takeaway is counter-intuitive but useful: with enough data and regularization, *very large* networks can generalize better than moderately large ones. Do not be afraid to scale up, as long as you also use early stopping, weight decay, or dropout.

Definition: Overfitting in Neural Networks

A neural network is said to overfit when its training loss continues to decrease while its validation loss begins to increase. Equivalently, the generalization gap (test error minus training error) is large and growing. Formal diagnostics include:

- Divergent training-validation loss curves after some epoch t^*.
- Test accuracy gap exceeding 5–10 percentage points compared to training.
- Weight magnitudes growing during late training (a sign of memorization).

The remedy is *regularization*—any modification to training that trades a small increase in training loss for a larger decrease in generalization gap. Section 6 surveys the main regularization techniques.

Common Misconceptions about Overfitting

(1) "More parameters always means more overfit." Not always. Double descent shows that, beyond the interpolation threshold, more parameters can *reduce* test error. The relationship between P/N and overfit is non-monotonic.
(2) "Low training error means the model is good." No. Low training error only guarantees that the model fits the training set. Generalization is measured on held-out data.
(3) "Overfitting can be detected after training finishes." It is much better detected *during* training via validation loss. Plot training and validation loss every epoch; act on the curves, not on the final number.

Worked Examples

Worked Example 1: Diagnosing an Overfit Network

A network trained for 200 epochs on 5000 observations with 50 inputs, two hidden layers of 128 neurons, and 10-class softmax output has the following training curve:

Epoch	Train loss	Val loss	Val accuracy
10	1.80	1.85	0.48
30	1.20	1.30	0.62
50	0.80	1.05	0.70
100	0.35	1.10	0.68
200	0.05	1.45	0.60

Val loss reaches its minimum at epoch 50 (1.05) and then rises. Val accuracy peaks at 0.70 at epoch 50 and falls to 0.60 by epoch 200. Train loss continues dropping from 0.80 at epoch 50 to 0.05 at epoch 200.
Diagnosis: classic overfitting starting around epoch 50. The model has effectively memorized the training set in epochs 50–200 without improving generalization.
Remedy: apply early stopping at epoch 50 (or use patience 5–10 to be safe). This alone recovers 10 percentage points of validation accuracy without any architectural change.

Worked Example 2: The Ratio That Predicts Overfitting

A research team is tuning an MLP for a binary classification task. They compare three configurations:

Config	Parameters P	Train size N	P/N
A (small)	5,000	100,000	0.05
B (medium)	50,000	20,000	2.5
C (large)	500,000	10,000	50

Predicted behavior:

- A: P/N very small. Little risk of overfit; may even underfit if the true function is complex.
- B: $P/N = 2.5$. Moderate overfit risk. Regularization (weight decay 10^{-4}, dropout 0.2, early stopping) should control it.
- C: $P/N = 50$. Severe overfit risk. Aggressive regularization essential (dropout 0.5, weight decay 10^{-3}, early stopping with small patience). May still overfit.

Rule of thumb: keep $P/N \leq 10$ without heavy regularization. For $P/N > 100$, consider whether you need so many parameters or whether a simpler model would suffice.

Historical Background: Belkin et al. and Double Descent (2019)

Until the late 2010s, learning theory and empirical practice seemed aligned on the U-shaped capacity curve: too little capacity underfits, too much overfits, and the optimal is in between. Then in 2019, Mikhail Belkin, Daniel Hsu, Siyuan Ma, and Soumik Mandal published "Reconciling modern machine-learning practice and the classical bias-variance trade-off" in PNAS. They showed that for many modern models—including neural networks, random features, and decision trees—the test error curve has a second descent after the interpolation threshold.

The paper catalogued double descent across architectures and datasets. On MNIST with a ResNet-18, for instance, test error initially falls with capacity, spikes at the interpolation threshold, then falls again as capacity grows further. The same phenomenon appears in tabular data, images, and text.

The explanation is still being worked out. Implicit regularization from SGD, the geometry of high-dimensional loss landscapes, and "benign overfitting" all contribute. The practical upshot: for modern deep networks, bigger is often better, provided you also train long enough and use standard regularization. The 2012 intuition "use just enough capacity" has been supplanted by "use ample capacity and regularize heavily."

Problem 5.1 (Easy) ⋆

Your network shows train accuracy 98% and test accuracy 71%. Diagnose: overfit, underfit, or neither? What single action would you try first?

Solution: see Appendix.

Problem 5.2 (Easy) ⋆

For each of the following, classify as bias, variance, or noise: (a) a linear model on curved data; (b) an MLP with 1M params on 1000 training rows; (c) measurement error in the target variable; (d) a tree of depth 30 on 500 rows.
Solution: see Appendix.

Problem 5.3 (Medium) ⋆⋆

An MLP has $P = 120{,}000$ parameters and is trained on $N = 8{,}000$ examples. Compute the ratio P/N. Based on the rule of thumb ($P/N \leq 10$ without regularization), what regularization techniques would you apply?
Solution: see Appendix.

Problem 5.4 (Medium) ⋆⋆

You plot training loss and validation loss by epoch. At epoch 50, train loss is 0.12 and val loss is 0.75. At epoch 100, train loss is 0.03 and val loss is 0.95. Is the model still learning generalizable patterns between epoch 50 and 100? Justify with one sentence.
Solution: see Appendix.

Problem 5.5 (Hard) ⋆ ⋆ ⋆

Explain the double descent phenomenon in your own words. Why does classical learning theory (which predicts a simple U-curve in capacity) miss it? Cite Belkin et al. (2019). What practical implication does double descent have for choosing model size?
Solution: see Appendix.

Connecting Forward

We have seen why deep networks overfit and how to diagnose the condition from training curves. Section 6 introduces the arsenal of regularization techniques that control overfitting: L2 weight decay, dropout, batch normalization, early stopping, and data augmentation. Each one trades a small increase in bias for a large decrease in variance—and together they turn overparameterized networks into reliable production models.

Key Takeaway: Deep networks overfit because they have more parameters than data and can fit any training set exactly—the cure is regularization, which we develop next.

6. Regularization for Deep Networks

Opening Problem: Saving the Hedge Fund's Overfit Model

Continuing the hedge-fund saga from Section 5: the 450,000-parameter MLP trained on 60,000 observations was a disaster in live trading. The research team cannot simply shrink the network—the smaller variants they tried before showed high bias (training accuracy 55% vs. 78%). They need the capacity of the big network without its tendency to memorize noise.

The fix is *regularization*: a collection of techniques that constrain the effective capacity of a model during training, trading a small amount of training-set fit for a large improvement in generalization. Four techniques, in descending order of generality: L2 weight decay, dropout, batch normalization, and early stopping. Section 6 develops each one, explains the mechanism, and gives concrete advice on tuning.

Discovery Question

You have a bowl of 20 marbles and a paper towel. If you drop the marbles on the towel, they stay put. If you jiggle the towel during training (randomly removing half the marbles at each iteration and asking the network to make do), what do you think happens? Why might this unintuitive procedure help rather than hurt?

L2 Weight Decay: Shrinking the Weights

The simplest and most general regularizer adds a penalty on the squared magnitudes of weights to the loss function:

$$L_{\text{reg}}(\theta) = L_{\text{data}}(\theta) + \lambda \sum_{\ell} \|W_\ell\|_F^2,$$

where $\|W\|_F^2 = \sum_{i,j} W_{ij}^2$ is the squared Frobenius norm and $\lambda > 0$ is the regularization strength. Biases are usually *not* regularized (they do not contribute to function complexity the way weights do).

The gradient of the L2 penalty with respect to W is $2\lambda W$. The gradient-descent update becomes:

$$W \leftarrow W - \eta(\nabla L_{\text{data}} + 2\lambda W) = (1 - 2\eta\lambda)W - \eta \nabla L_{\text{data}}.$$

Each step multiplies W by a factor slightly less than 1 before applying the data gradient. This is why L2 regularization is often called *weight decay*: the weights decay toward zero between data-driven updates.

Key Formula: L2-Regularized Loss

$$L_{\text{reg}}(\theta) = L_{\text{data}}(\theta) + \lambda \sum_{\ell=1}^{L} \|W_\ell\|_F^2.$$

Gradient update: $W_\ell \leftarrow (1 - 2\eta\lambda)W_\ell - \eta \nabla L_{\text{data}}$.
Bayesian view: L2 regularization corresponds to a Gaussian prior $\mathcal{N}(0, 1/(2\lambda))$ on the weights. The optimal regularized weights are the maximum a posteriori (MAP) estimate under this prior.
Tuning: start with $\lambda = 10^{-4}$, sweep logarithmically over $\{10^{-6}, 10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}\}$ on the validation set.

(c) Joerg Osterrieder 2026

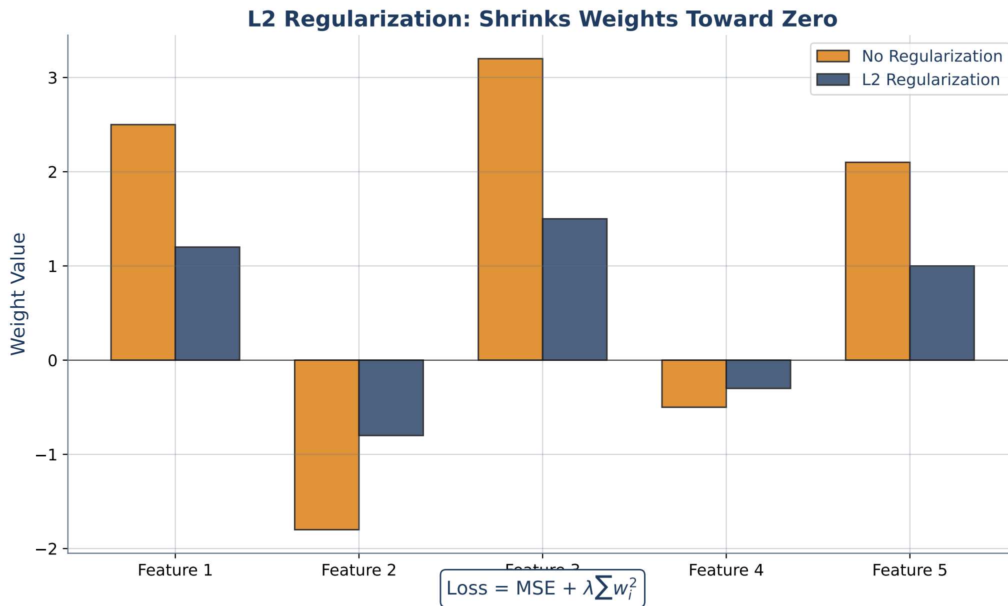

Figure 60: L2 regularization. The penalty $\lambda\|w\|^2$ is a quadratic bowl centered at the origin. Adding it to the data loss shrinks the optimum toward zero.

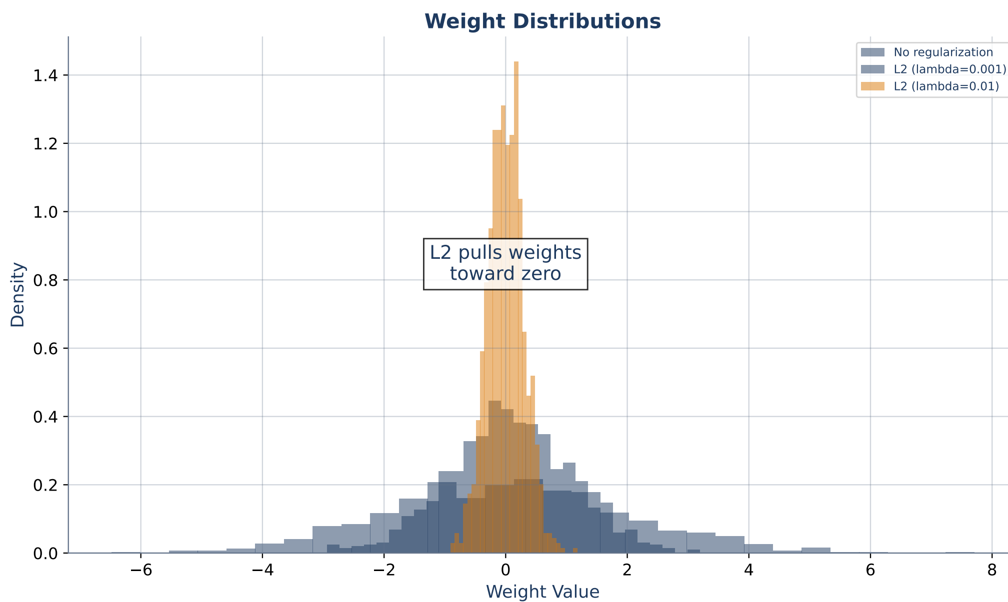

Figure 61: Weight distributions with and without L2 regularization. Without regularization (blue), weights sprawl. With regularization (orange), weights concentrate near zero.

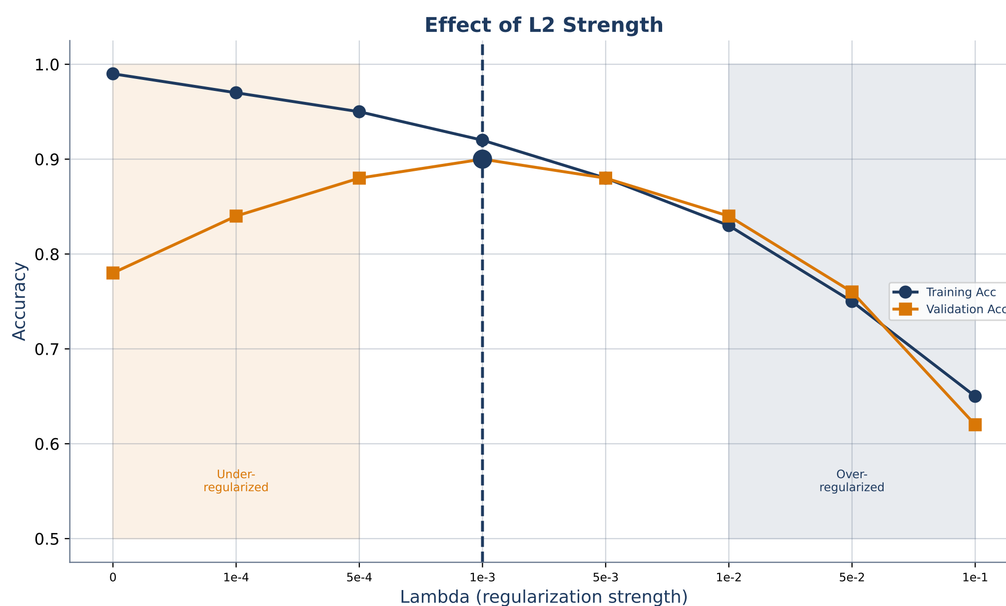

Figure 62: Effect of λ on validation loss. Too small: overfitting persists. Too large: underfitting. Sweet spot typically at $\lambda \in [10^{-5}, 10^{-3}]$ for MLPs.

Dropout: Random Neural Deletion

Dropout (Srivastava et al., 2014) is a surprisingly crude but dramatically effective regularizer. At each training step, for each hidden neuron, flip a coin with probability p; if heads, set the neuron's output to zero. The network must now make predictions using random subsets of its neurons.

At test time, dropout is turned off (all neurons are active) but the weights are scaled by $1-p$ to compensate for the change in expected activation magnitude. Modern frameworks usually implement "inverted dropout," which does the scaling at train time by $1/(1-p)$ so that no adjustment is needed at test time.

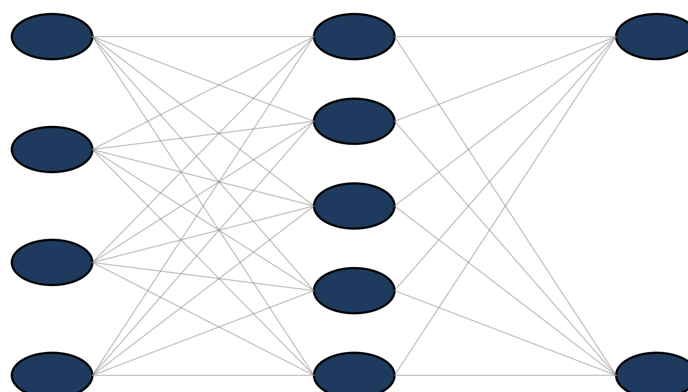

Figure 63: A standard dense network without dropout. Every neuron fires every time.

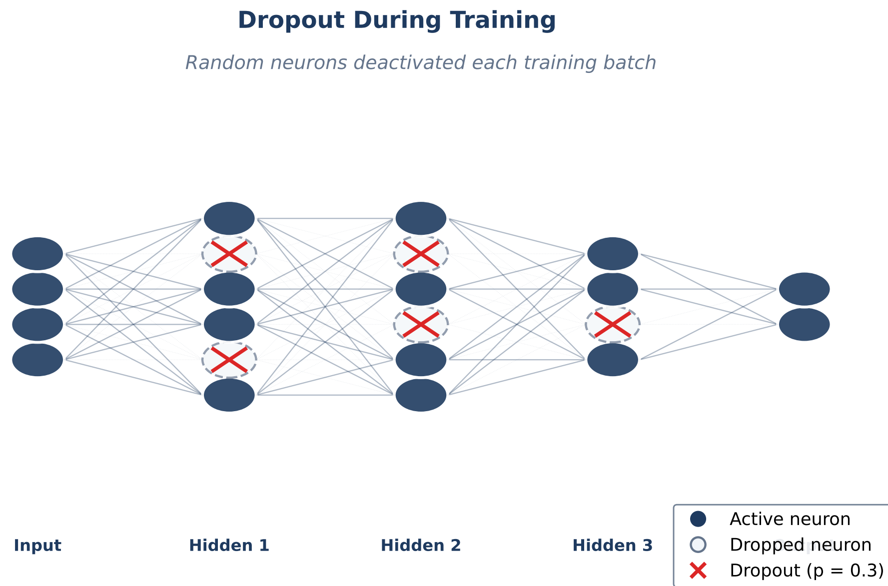

Figure 64: The same network with dropout during training. A random subset of neurons is zeroed out each step. The network cannot rely on any one neuron and must develop redundant representations.

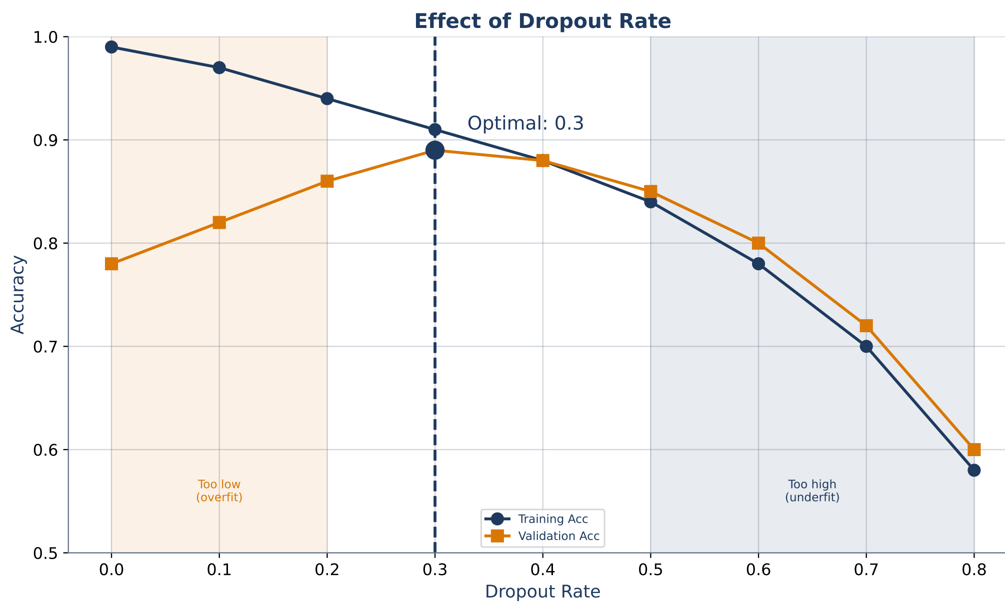

Figure 65: Effect of dropout rate p on validation accuracy. Typical sweet spot: $p = 0.2$–0.5. Too high: underfitting. Too low: little regularization benefit.

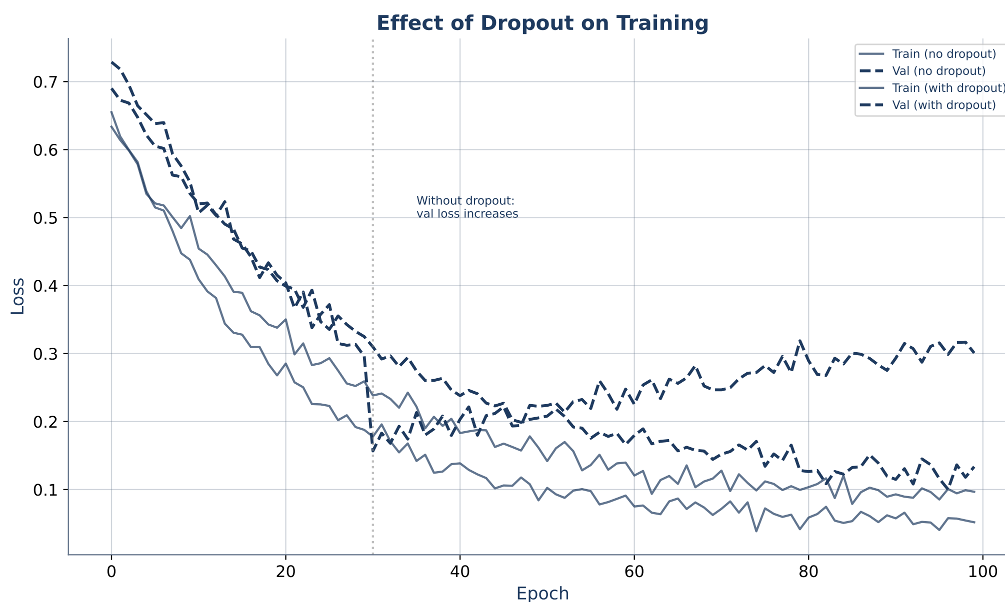

Figure 66: Dropout closes the train-val loss gap. Training loss increases slightly (the network cannot fit noise as well); validation loss improves substantially.

Key Formula: Dropout

Training: for each hidden neuron h_j, sample $m_j \sim \text{Bernoulli}(1-p)$ independently and replace $h_j \leftarrow m_j h_j/(1-p)$ (inverted dropout).
Inference: use all neurons without modification.
Interpretation: at each step, you are training one of 2^H possible sub-networks (where H is the number of hidden neurons); at inference, you approximate the average of all these sub-networks. Dropout is ensemble learning via parameter sharing.
Tuning: $p = 0.2$ for input and early layers; $p = 0.5$ for wider hidden layers. Avoid dropout on the output layer.

Batch Normalization: Stabilizing Activations

Batch normalization (Ioffe and Szegedy, 2015) normalizes the pre-activations of each layer to have mean zero and variance one across a mini-batch, then applies a learned affine transformation (γ, β):

$$\hat{z} = \frac{z - \mu_{\text{batch}}}{\sqrt{\sigma^2_{\text{batch}} + \epsilon}}, \qquad y = \gamma \hat{z} + \beta.$$

At inference, running averages of μ and σ (collected during training) are used instead of batch statistics.

Batch norm was introduced as a remedy for “internal covariate shift”—the distribution of each layer’s inputs drifts as earlier layers update. Later analysis (Santurkar et al., 2018) showed the real benefit is smoother loss landscapes, which allow higher learning rates and faster convergence. As a side effect, batch norm also regularizes: the batch statistics are noisy at small batch sizes, which injects useful randomness into training.

Use batch norm on hidden layers in deep networks. For very shallow networks or batch sizes < 16, batch norm may hurt; layer norm or group norm are alternatives.

Early Stopping: Just Don't Overtrain

The simplest regularizer of all: monitor validation loss every epoch, and stop training when it stops improving. Specifically, track the best validation loss so far and a "patience" counter; if validation loss does not improve for p epochs, stop.

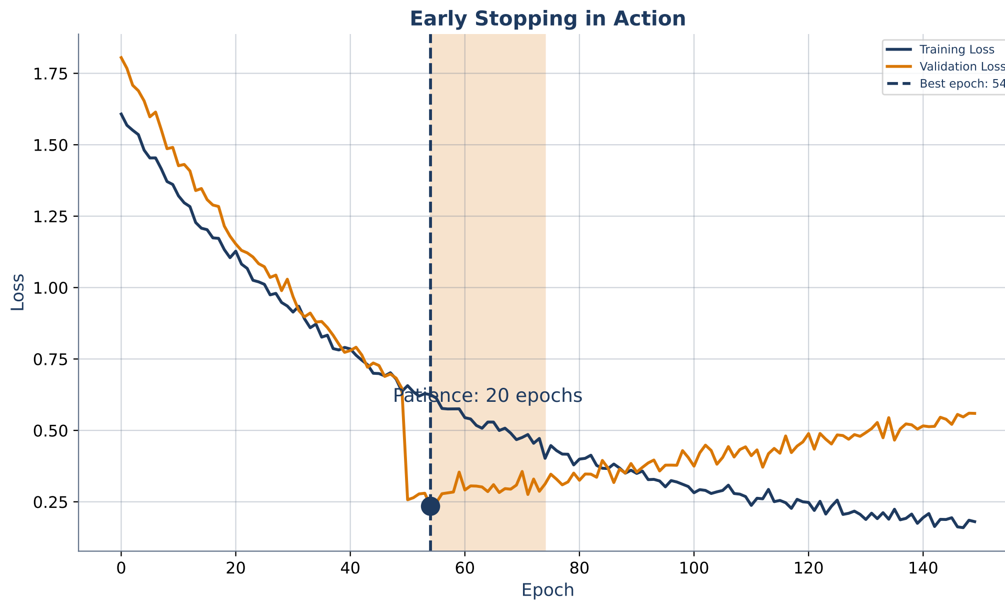

Figure 67: Early stopping in action. Training is halted just before validation loss starts climbing. The saved model is the one with the best validation loss, not the last one trained.

Key Formula: Early Stopping Criterion

Let $v_1, v_2, \ldots, v_T$ be the validation losses at each epoch and $v_t^* = \min_{i \le t} v_i$ be the best so far. Stop training at epoch T if

$$t - \arg\min_i v_i \ge p,$$

where p is the patience. The final model is the one saved at the epoch that achieved v^*, not the last epoch trained.

Early stopping acts as an implicit regularizer. With gradient descent starting from small weights, the effective model complexity at time t is bounded by the number of gradient steps taken. Stopping early limits this complexity just like L2 does—in fact, for linear models the two are mathematically equivalent.

Data Augmentation: More Data, Synthesized

When getting more real data is impossible, synthesize more data by applying label-preserving transformations. For images: random crops, horizontal flips, color jitter, Cutout. For audio: pitch shifts, time warps, noise addition. For text: synonym replacement, back-translation.

For tabular and finance data, augmentation is trickier but possible. SMOTE synthesizes minority-class examples by interpolating between existing ones. For time series, jittering (adding Gaussian noise), scaling, and window slicing are common.

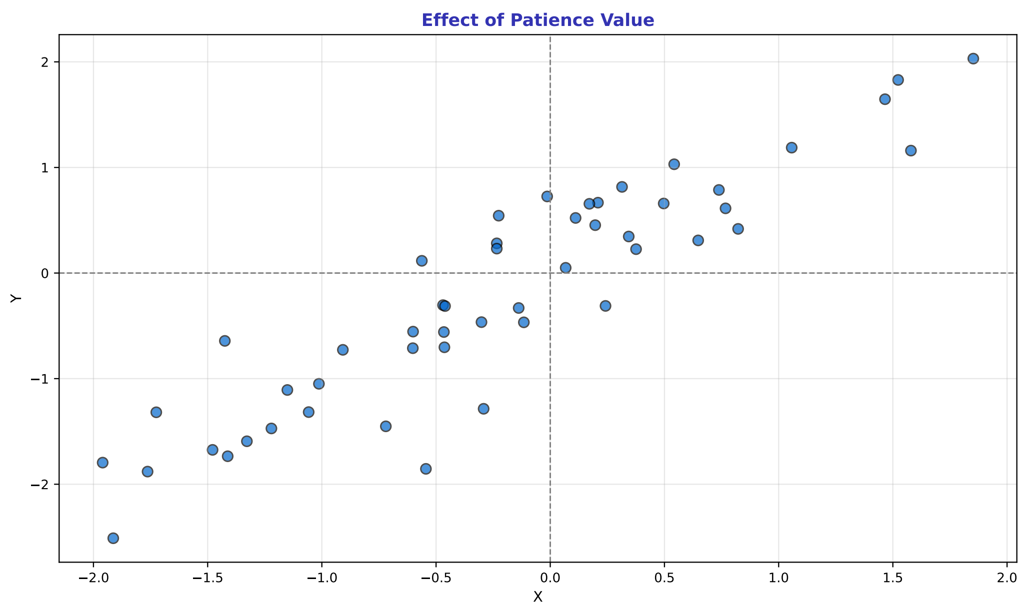

Figure 68: Effect of patience parameter. Too short: premature stopping, possible underfit. Too long: training continues into the overfit regime. Typical: patience = 5–20 epochs.

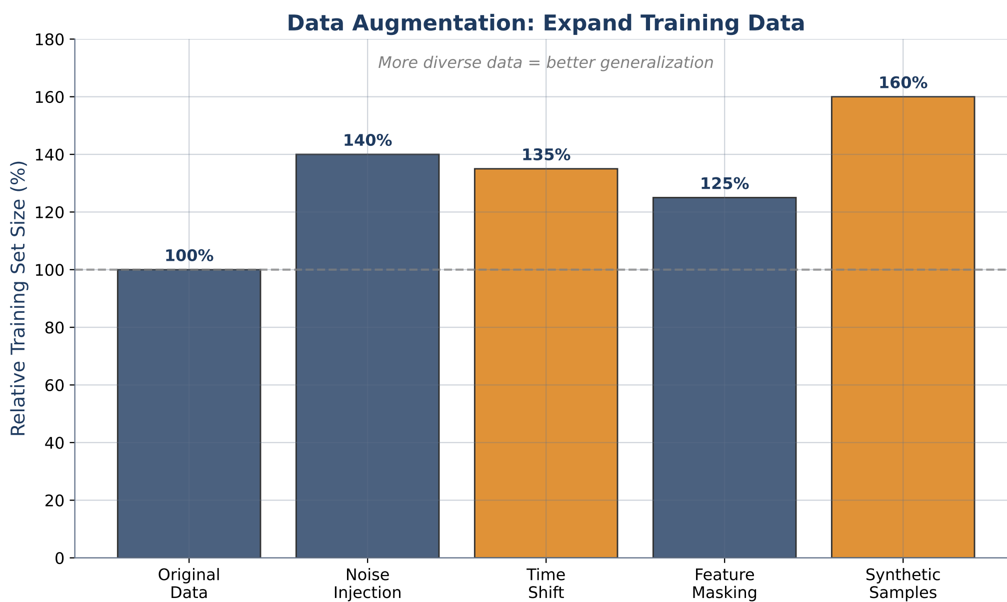

Figure 69: Data augmentation: generating new training examples by applying transformations that preserve the label. A horizontally flipped cat image is still a cat image.

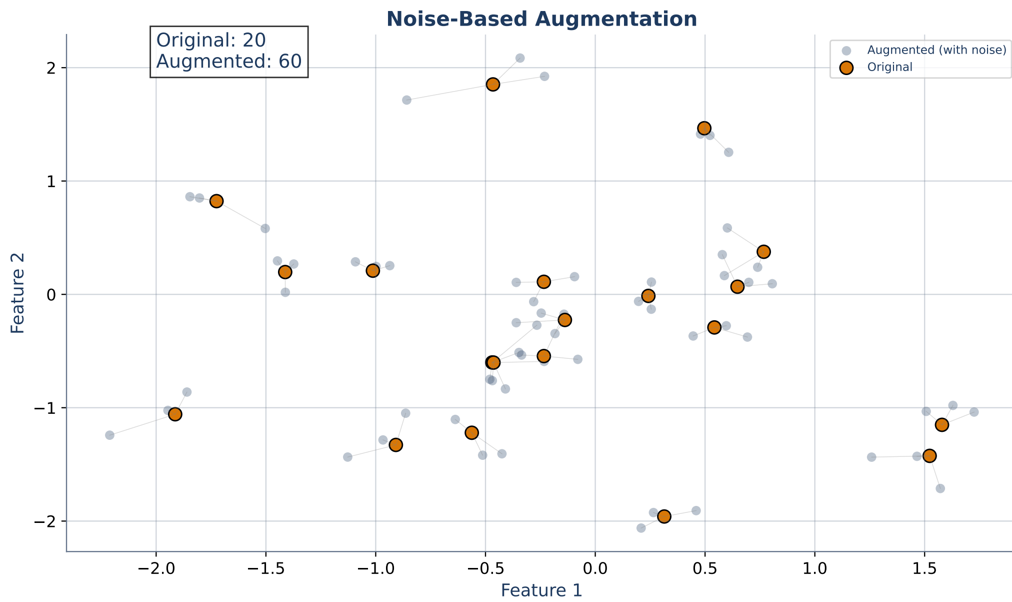

Figure 70: Noise-based augmentation for tabular data. Adding small Gaussian noise to features during training forces the network to learn smoother decision boundaries.

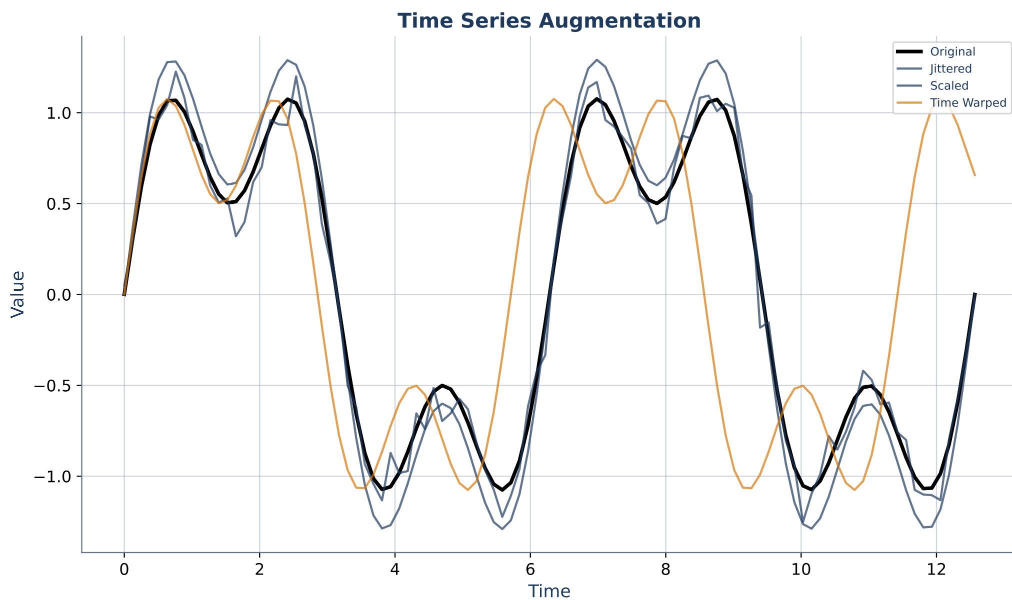

Figure 71: Time-series augmentation: jittering, scaling, and window shifts produce new training examples that the model must handle robustly.

Definition: Regularization

Regularization refers to any modification to a learning algorithm that is intended to reduce generalization error but not training error—i.e., to trade bias for a larger reduction in variance. Common deep-learning regularizers include:

- **L2 / weight decay:** penalize squared weight magnitudes.
- **L1:** penalize absolute weight magnitudes (induces sparsity).
- **Dropout:** randomly zero hidden activations during training.
- **Batch normalization:** normalize layer pre-activations, implicitly regularizing.
- **Early stopping:** halt training when validation loss stops improving.
- **Data augmentation:** synthesize new training examples via label-preserving transformations.
- **Noise injection:** add random noise to inputs, weights, or gradients.

In practice, combine multiple regularizers: L2 + dropout + early stopping is a solid default recipe.

Common Misconceptions about Regularization

(1) "Dropout is outdated because batch norm does the same thing." They have different effects. Batch norm stabilizes training and provides mild regularization; dropout directly combats co-adaptation of neurons. Both can (and often should) be used together, though interactions exist—dropout before batch norm can hurt.
(2) "More regularization is always better." No. Too much λ or too high dropout rate p causes underfitting: the model cannot even fit the training set. Tune regularization strength on validation data.
(3) "Early stopping replaces the need for a test set." Early stopping uses validation loss to choose a checkpoint. You still need a separate held-out test set to get an unbiased estimate of generalization on data that influenced no decisions during training.

(c) Joerg Osterrieder 2026

Worked Examples

Worked Example 1: Applying Dropout to an MLP in Keras

Our credit-risk MLP from Section 3 had 20 inputs, two hidden layers of 64 neurons, and 1 output. With 10,000 training rows and 5569 parameters ($P/N = 0.56$), we expect some overfitting.
Apply dropout with $p = 0.3$ after each hidden layer:

MLP with Dropout

```
from keras import layers, Model, Input

x = Input(shape=(20,))
h = layers.Dense(64, activation='relu')(x)
h = layers.Dropout(0.3)(h)
h = layers.Dense(64, activation='relu')(h)
h = layers.Dropout(0.3)(h)
y = layers.Dense(1, activation='sigmoid')(h)

model = Model(x, y)
model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

During training, each hidden neuron has a 30% chance of being zeroed out per forward pass. The network learns to distribute the load so that no single neuron is critical. At test time, dropout is off, and all 128 hidden neurons contribute.
Expected result: small increase in training loss (the network cannot memorize as easily); modest improvement in validation accuracy (2–5 percentage points typical).

Worked Example 2: Combining Regularizers

A research team wants to train a 5-layer MLP ($\sim$2M parameters) on 100k tabular examples. They use the following regularization stack:

1. **L2 weight decay** with $\lambda = 10^{-4}$ on all weights (not biases).
2. **Dropout** with $p = 0.3$ after the second, third, and fourth hidden layers.
3. **Batch normalization** after each linear transformation, before the activation.
4. **Early stopping** on validation loss with patience 10.
5. **Input jittering**: add Gaussian noise with $\sigma = 0.02$ to each standardized input during training.

Each technique addresses a different failure mode: L2 shrinks weights, dropout prevents co-adaptation, batch norm smooths the loss landscape, early stopping halts before overfitting, and jittering enforces local smoothness. Together they reduce the generalization gap from $\sim$15 percentage points (no regularization) to $\sim$3 percentage points.
The art of deep learning is choosing which regularizers to combine. There is no universal best combination; try a few, validate, iterate.

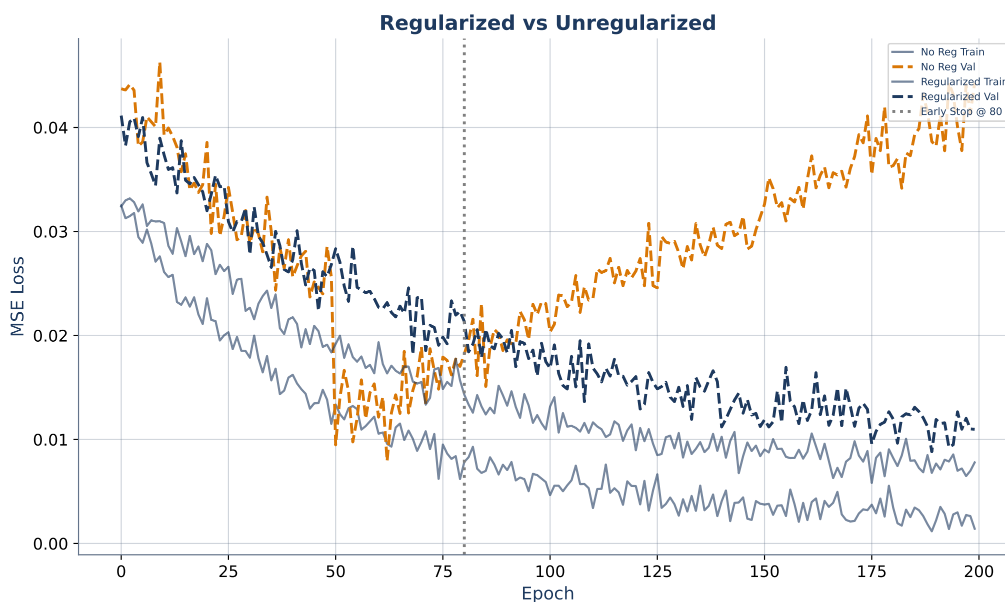

Figure 72: Regularized vs. unregularized training. The regularized model has slightly higher training loss but substantially better validation loss and much tighter generalization gap.

Historical Background: Srivastava and Dropout (2014)

In 2014 Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov published "Dropout: A Simple Way to Prevent Neural Networks from Overfitting" in the *Journal of Machine Learning Research*. The paper proposed randomly setting units to zero during training as an approximation to training an exponential ensemble of thinned networks.

The idea was reportedly inspired by biological neurons, where signals are intermittent rather than deterministic. It was also inspired by the practice of random feature deletion in bagged ensembles. The paper showed that dropout improved state-of-the-art results on MNIST, CIFAR-10, TIMIT speech, and various natural-language tasks.

Dropout's influence was enormous. It became the default regularizer for deep networks between 2014 and about 2019, when batch norm began to handle much of the regularization role in convolutional networks. Dropout remains standard for MLPs, recurrent networks, and transformer feedforward blocks. The 2014 paper has been cited over 50,000 times—one of the most influential papers of the deep-learning era.

Problem 6.1 (Easy) ⋆

An MLP is trained with L2 regularization coefficient $\lambda = 10^{-3}$. The gradient update for a weight $w = 2$ with data-loss gradient -0.05 and learning rate $\eta = 0.01$ is what? Show the full computation.

Solution: see Appendix.

Problem 6.2 (Easy) ⋆

During training, a dropout layer with $p = 0.4$ randomly zeros 40% of its inputs and scales the rest by $1/(1-p) = 1.67$. If the pre-dropout activation vector is $(1.0, 0.5, -0.2, 2.0, 0.3)$ and the dropout mask (sampled independently) is $(1, 0, 1, 1, 0)$, what is the output of the dropout layer during training?
Solution: see Appendix.

Problem 6.3 (Medium) ⋆⋆

A model has validation losses per epoch of $\{1.2, 1.1, 1.0, 0.95, 0.92, 0.91, 0.93, 0.94, 0.95, 0.97\}$. With patience parameter $p = 3$, at which epoch does early stopping trigger? Which epoch's model should be used for inference?
Solution: see Appendix.

Problem 6.4 (Medium) ⋆⋆

Explain in two or three sentences why L2 weight decay corresponds to a Gaussian prior on the weights from a Bayesian perspective. What prior does L1 regularization correspond to?
Solution: see Appendix.

Problem 6.5 (Hard) ⋆ ⋆ ⋆

Prove that for a linear regression model $\hat{y} = w^\top x$ with MSE loss, early stopping with gradient descent (starting from $w_0 = 0$) has an equivalent L2-regularized closed-form solution. Specifically, show that after t gradient-descent iterations with step size η, the weights w_t coincide with the L2 solution $w_\lambda = (X^\top X + \lambda I)^{-1} X^\top y$ for a specific $\lambda = \lambda(t, \eta)$. (*Hint:* write out the gradient-descent iteration explicitly and compare to the closed-form L2 ridge solution. Use eigendecomposition of $X^\top X$.)
Solution: see Appendix.

Connecting Forward

We now have a toolbox of regularization techniques: L2 weight decay, dropout, batch normalization, early stopping, and data augmentation. Section 7 covers the training-time tricks that determine whether a well-regularized network actually converges: learning rate scheduling, weight initialization, gradient clipping, and detection of numerical failures. These are not glamorous topics, but they separate successful training runs from days wasted on NaN losses and dead neurons.

Key Takeaway: Regularization trades a small increase in training loss for a large decrease in generalization gap—combine L2, dropout, and early stopping for a reliable default recipe.

7. Training Tricks and Pitfalls

Opening Problem: The Loss That Exploded at 3 a.m.

An overnight training job on a price-prediction MLP was expected to finish by 6 a.m. The researcher wakes up to a Slack message from the monitoring system: at epoch 87, the loss became NaN. The GPU has been burning electricity since then. Scrolling through the log: everything was fine through epoch 50 (training loss 0.13), then a small spike around epoch 80, then one gradient update exploded and all subsequent losses are NaN.
The researcher's next question: what went wrong? Was it the data (bad row)? A numerical issue in the custom loss? A learning-rate that was too aggressive for a region of weight space? This section covers the practical training failures that are not bugs in the math but problems in how the math is executed. We cover learning-rate scheduling, weight initialization, gradient clipping, and a debugging checklist that will save future 3 a.m. disasters.

Discovery Question

At initialization, every weight in the network is a random number drawn from some distribution. Does it matter what distribution? Specifically: if you initialize all weights to the same value (say, zero), what goes wrong? What if you initialize to large random values?

Weight Initialization: Not Zero, Not Huge

Initializing all weights to zero is a classic beginner mistake. If every weight in a layer is the same, every neuron in that layer computes the same output for any input. Gradients are identical too. The weights update in lockstep and never differentiate. The network has the symmetry of a kaleidoscope, and training cannot break it.

The remedy is random initialization. But the scale matters. Too small: activations shrink toward zero through the layers, and gradients vanish. Too large: activations blow up through the layers, and training is unstable. Two standard schemes strike the right balance.

Xavier (Glorot) initialization. For a layer with d_{in} inputs and d_{out} outputs, draw weights from $\mathcal{N}(0, 2/(d_{\text{in}} + d_{\text{out}}))$. Works well for sigmoid and tanh activations.

He initialization. For ReLU activations, draw weights from $\mathcal{N}(0, 2/d_{\text{in}})$. ReLU zeros out half the activations, so the variance must be larger to compensate.

Key Formula: Xavier vs. He Initialization

Xavier (Glorot 2010), for sigmoid/tanh:

$$W_{ij} \sim \mathcal{N}\left(0, \frac{2}{d_{\text{in}} + d_{\text{out}}}\right) \quad \text{or} \quad W_{ij} \sim \text{Uniform}\left(-\sqrt{\tfrac{6}{d_{\text{in}}+d_{\text{out}}}}, \sqrt{\tfrac{6}{d_{\text{in}}+d_{\text{out}}}}\right).$$

He (He et al. 2015), for ReLU:

$$W_{ij} \sim \mathcal{N}\left(0, \frac{2}{d_{\text{in}}}\right).$$

Biases: initialize to zero.
Why it works: both schemes keep the variance of activations roughly constant across layers, preventing vanishing/exploding signals at initialization.

Modern frameworks (Keras, PyTorch) default to sensible initializations—He for ReLU layers, Xavier for tanh/sigmoid. Usually you never need to specify explicitly. But if you are debugging a network that won't train, checking the initialization is a quick first step.

Learning Rate Schedules

A constant learning rate is rarely optimal. Early in training, the weights are far from the minimum, and a large step makes rapid progress. Near the minimum, large steps overshoot and oscillate; a smaller step is needed to settle in. Learning-rate schedules decrease η over time.

Step decay. Halve η every k epochs. Simple, common, effective. Typical: $k = 30$, factor $=$ 0.5.

Exponential decay. $\eta_t = \eta_0 \cdot e^{-\alpha t}$ for some decay rate α.

Cosine annealing. $\eta_t = \eta_{\min} + 0.5(\eta_0 - \eta_{\min})(1 + \cos(\pi t/T))$ over a training horizon T. Smoothly decreases from η_0 to $\eta_{\min}$ following a half-cosine curve. State of the art for many architectures.

Warmup + cosine. Start with a very small learning rate, linearly increase to η_0 over a few epochs (the warmup), then anneal via cosine. Used in most modern transformer training. Warmup prevents early training divergence when weights are random.

ReduceLROnPlateau. Monitor validation loss; whenever it stops improving for p epochs, multiply the learning rate by 0.1. Adaptive to the specific training trajectory.

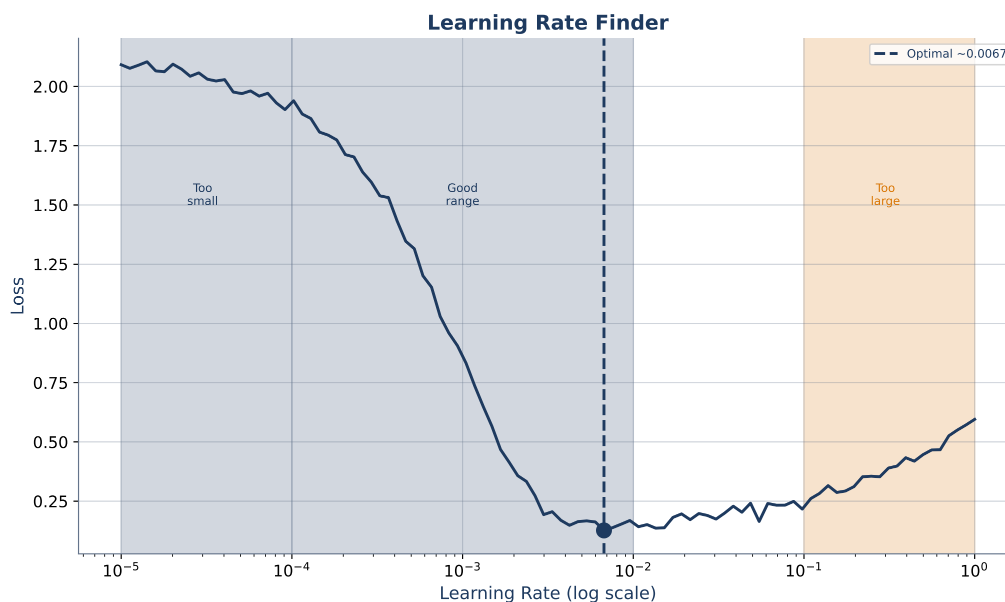

Figure 73: Learning-rate finder plot. Train for a few iterations with exponentially increasing η and watch the loss. The optimal η is typically an order of magnitude below where loss begins to diverge.

Gradient Clipping

Recurrent networks and occasionally MLPs can experience "exploding gradients"—gradient magnitudes that grow exponentially through backpropagation, causing huge weight updates that destabilize training. The standard remedy is gradient clipping: if the global norm of the gradient vector exceeds a threshold c, rescale it to norm exactly c.

$$g \leftarrow \begin{cases} g & \text{if } \|g\| \le c \\ c \cdot g/\|g\| & \text{otherwise.} \end{cases}$$

Typical $c = 1$ or $c = 5$ for RNNs. For MLPs, gradient clipping is usually unnecessary, but adding it costs little and can prevent rare numerical blowups.

Detecting Numerical Failures

Neural networks have three characteristic numerical failure modes.

NaN loss. A loss that is 0/0 or log(0) somewhere. Common causes: label values outside the expected range (e.g., $y = 1.001$ when code expects $y \in [0, 1]$); division by a near-zero quantity in a custom loss; log of sigmoid output that has saturated to zero. Fix: add $\epsilon = 10^{-8}$ to denominators and arguments of log.

Infinite loss. exp of a large number overflows. Common cause: softmax applied to raw logits that are very large. Fix: use the "log-sum-exp trick" to compute softmax stably (subtract max before exponentiating).

Dead neurons / vanishing gradients. Training proceeds but loss plateaus well above the expected optimum. Common causes: ReLU units that got stuck with negative inputs (no gradient flows); sigmoid saturation in deep networks. Fix: lower learning rate, switch to Leaky ReLU, reinitialize.

Definition: Practical Debugging Checklist

When a network is misbehaving, run through this checklist in order:

1. **Data:** are inputs standardized (mean 0, std 1)? Are labels in the expected range?
2. **Initialization:** are you using Xavier or He? All weights non-zero, modest magnitudes?
3. **Learning rate:** is η reasonable? Try $\eta \times 10$ and $\eta/10$. Use a learning-rate finder.
4. **Batch size:** try a very small batch (8) and a large one (512). Which trains?
5. **Loss matches output:** cross-entropy with softmax, MSE with linear output, BCE with sigmoid?
6. **Numerical stability:** small ϵ in log/divide operations?
7. **Overfit a single batch:** if you cannot overfit 16 training examples, something is wrong at the architecture/code level.
8. **Plot everything:** training loss, validation loss, gradient norms, weight magnitudes, activations.

The "overfit a single batch" test is the single most useful diagnostic in deep learning. If your code is correct, the network should be able to drive the loss of 16 handpicked examples to near zero in a minute or two.

(c) Joerg Osterrieder 2026

Figure 74: A healthy training-loss curve. Monotonically decreasing with gentle decay; validation tracks closely. This is what success looks like.

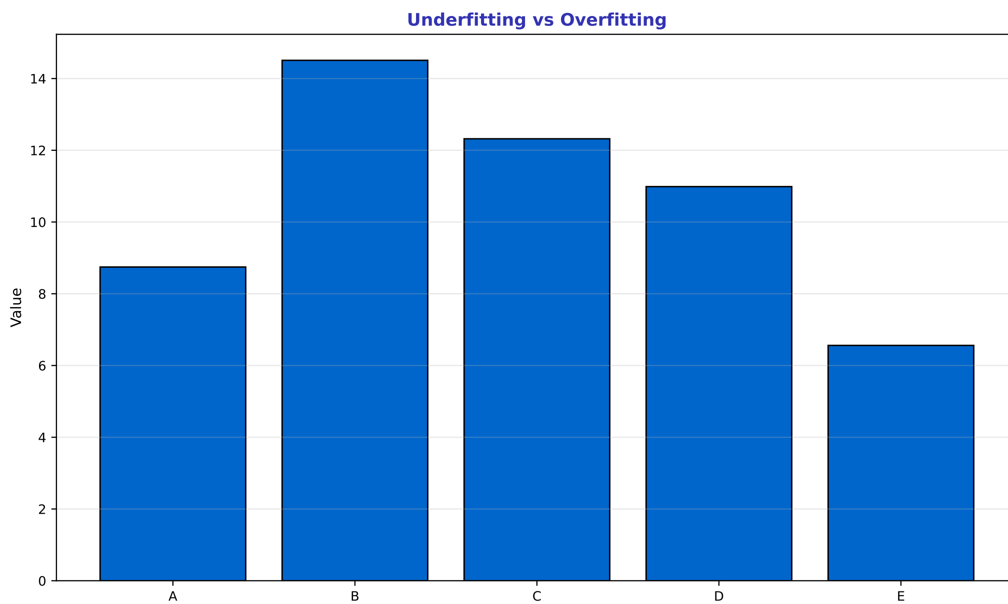

Figure 75: Training curves for underfit, just-right, and overfit networks. Knowing which regime you are in is half the battle.

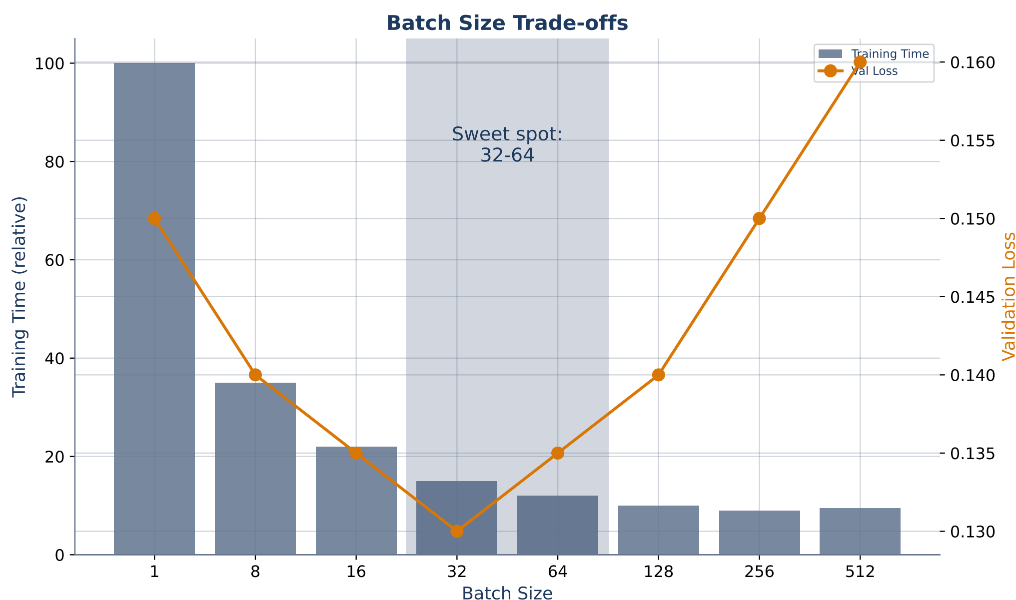

Figure 76: Batch size tradeoffs. Smaller batches add noise (helps escape saddle points) but are slower per epoch. Larger batches are smoother but may generalize worse.

Common Misconceptions about Training Tricks

(1) "Default hyperparameters always work." Sometimes, but not always. Adam with $\eta = 10^{-3}$ is a great default, but for finance time series with small signal-to-noise ratios, $\eta = 10^{-4}$ often works better. Always sweep at least one order of magnitude.
(2) "NaN loss means a bug in the math." More often it means a numerical issue—$\log(0)$, division by zero, overflow—in otherwise-correct math. The fix is usually adding a small ϵ or switching to a numerically stable formulation.
(3) "The best architecture matters more than training hyperparameters." For a trained model, often not. A well-tuned MLP with good regularization and a learning-rate schedule often outperforms a poorly tuned transformer on the same tabular problem.

Worked Examples

Worked Example 1: Finding a Good Learning Rate

A research team trains a price predictor. They use the learning-rate finder: train for 200 iterations with η exponentially increasing from 10^{-7} to 10^{0}. Plot training loss vs. $\log(\eta)$. The loss stays flat from 10^{-7} to 10^{-5} (updates too small to matter), drops sharply from 10^{-5} to 10^{-3}, reaches minimum around 10^{-3}, rises again from 10^{-3} to 10^{-1}, and diverges to infinity for $\eta > 10^{-1}$.
Rule of thumb: pick η one order of magnitude below the minimum. Here that is 10^{-4}. Start training at 10^{-4} and apply a cosine annealing schedule down to 10^{-5} over 100 epochs. The learning-rate finder takes 10 minutes and saves hours of guesswork.

Worked Example 2: Debugging a Failed Training Run

A junior researcher's MLP trains to loss 0.693 (exactly $\ln 2$, the BCE loss of a 50/50 guess) and never improves. Debugging checklist:

1. **Data:** inputs are standardized. Labels are $\{0, 1\}$ as expected. ✓
2. **Initialization:** Keras default (Glorot). ✓
3. **Learning rate:** Adam default 10^{-3}. Try 10^{-4} and 10^{-2}. Both give the same result—the model is not even trying to learn.
4. **Overfit a single batch:** take 16 training examples and train for 1000 steps. Loss stays at 0.693. **Red flag.** If we cannot overfit 16 examples, something structural is broken.

Further investigation shows the researcher accidentally used `activation='sigmoid'` on the final layer but then used `loss='binary_crossentropy'` with `from_logits=True`. The log-of-sigmoid is being applied twice, which produces a trivially flat loss surface.
Fix: set `from_logits=False` or remove the sigmoid from the final layer. Loss now drops to 0.12 on the 16-example batch in 200 steps. The rest of training proceeds normally.

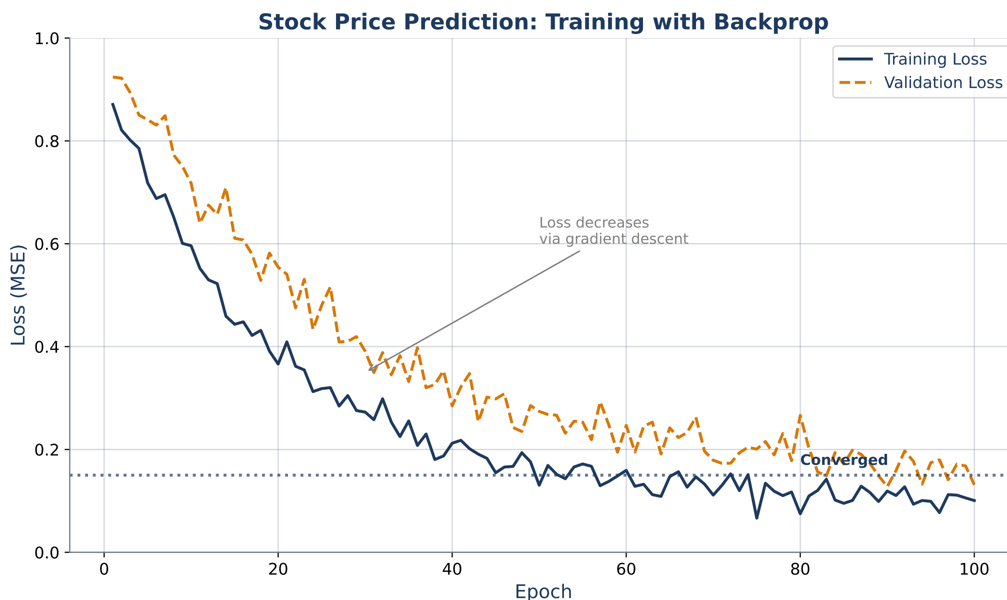

Figure 77: A finance problem setup: predict next-day direction from today's features. Small signal-to-noise ratio (typical $R^2 \approx 0.02$) makes deep learning here especially sensitive to hyperparameters.

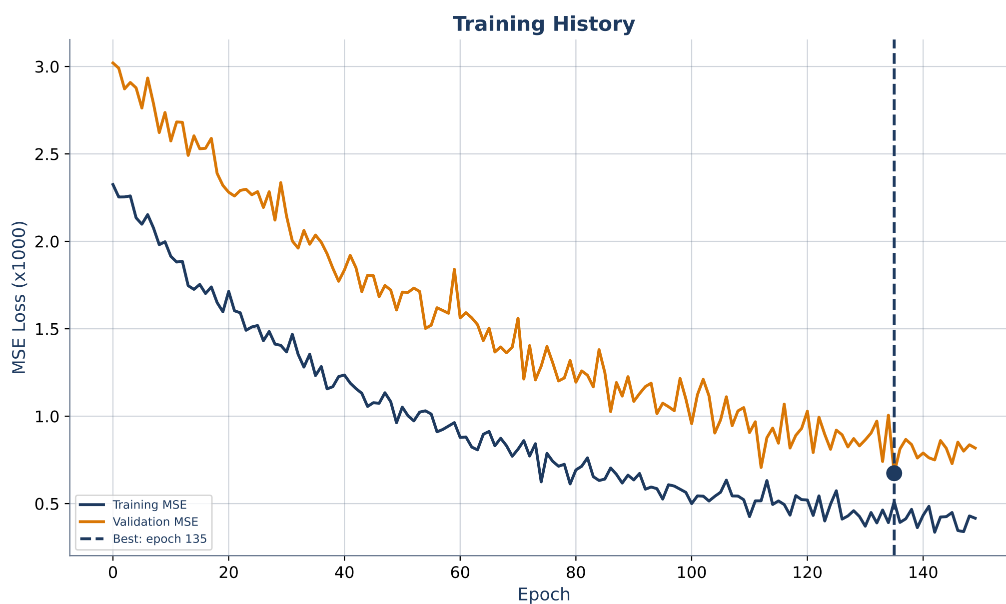

Figure 78: Training history for the finance model. Multiple runs with different random seeds show variability inherent to non-convex optimization.

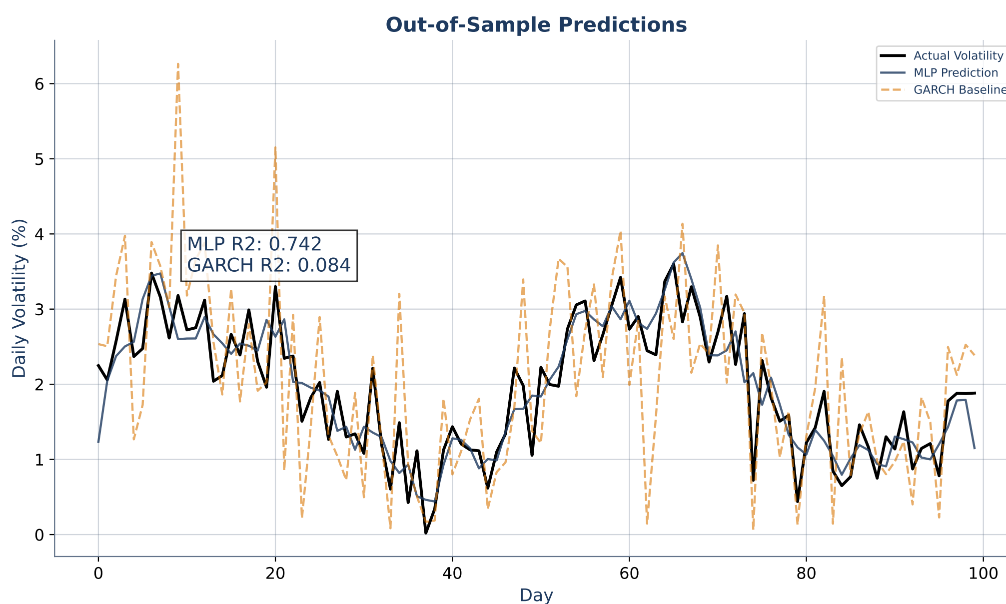

Figure 79: Out-of-sample predictions vs. realized returns. Even a carefully trained network produces noisy predictions; the art is extracting directional signal rather than chasing exact values.

Historical Background: Glorot, He, and the Initialization Revolution (2010–2015)

Before 2010, neural-network initialization was a black art. Researchers copied schemes from earlier papers and hoped for the best. In 2010, Xavier Glorot and Yoshua Bengio published "Understanding the difficulty of training deep feedforward neural networks" (AISTATS). They analyzed how activations and gradients propagate through layers and derived an initialization scale that keeps variance roughly constant through the network. The scheme is now called Xavier (or Glorot) initialization.

In 2015, Kaiming He and colleagues at Microsoft Research generalized the analysis to ReLU activations. Their paper "Delving deep into rectifiers" (ICCV 2015) showed that because ReLU zeros out half the activations, the variance must be twice as large to compensate. The resulting scheme, He initialization, is the modern default for ReLU networks.

Before Glorot and He, people trained 3- to 5-layer networks at best. With proper initialization, batch normalization, and residual connections, networks with hundreds of layers became feasible. Initialization is one of those "obvious in hindsight" innovations that unlocked a decade of progress.

Problem 7.1 (Easy) ⋆

A fully connected layer has 256 inputs and 128 outputs. Using He initialization, what is the standard deviation of each weight at initialization?

Solution: see Appendix.

Problem 7.2 (Easy) ⋆

Explain in one sentence why initializing all weights to zero prevents a multilayer network from learning.

Solution: see Appendix.

Problem 7.3 (Medium) ⋆⋆

A training run shows validation loss 0.80 at epoch 10, 0.76 at epoch 30, 0.76 at epoch 40, 0.76 at epoch 50. Using the ReduceLROnPlateau scheduler with patience 10 and factor 0.1, at which epoch is the learning rate first reduced?

Solution: see Appendix.

Problem 7.4 (Hard) ⋆ ⋆ ⋆

A cosine-annealing schedule runs from $\eta_0 = 10^{-3}$ to $\eta_{\min} = 10^{-6}$ over $T = 100$ epochs. Compute η_t at epochs $t = 0, 25, 50, 75, 100$ using the formula $\eta_t = \eta_{\min} + 0.5(\eta_0 - \eta_{\min})(1 + \cos(\pi t/T))$.

Solution: see Appendix.

Connecting Forward

We now have a complete picture of how to train deep networks: from forward propagation to backprop, from SGD to Adam, from overfitting diagnostics to regularization, and from initialization to learning-rate schedules. Section 8 steps back and asks the big-picture question: when is all this machinery worth the trouble in a finance context? We review concrete use cases where deep learning has succeeded (NLP for news sentiment, CNNs for chart pattern recognition, LSTMs

for time series), cases where it has failed (tabular credit scoring), and the trends that will shape the next decade.

Key Takeaway: Training success depends on small details—initialization, learning rate, numerical stability—as much as on architecture; when a network misbehaves, systematically check the checklist before changing the model.

8. Deep Learning in Finance – Use Cases and Limits

Opening Problem: The Fund That Added 100 Layers to a Credit Model

A large retail bank commissions a new credit risk model. The old model is gradient boosting (LightGBM) achieving AUC 0.86 on tabular features. Leadership wants to "modernize" and asks the data science team to replace it with deep learning. A capable team spends six months building a 20-layer residual MLP with all the bells and whistles: He initialization, dropout 0.3, batch normalization, cosine-annealing schedule, 2 million parameters.
The result on held-out data: AUC 0.84. Slightly worse than LightGBM. The deep model took ten times as much training compute, is harder to explain to regulators, and does not clearly beat the boosting baseline. Management is disappointed. The data science lead writes a memo explaining that this outcome is not surprising—and that the disappointment reflects a misunderstanding of where deep learning shines. Section 8 is that memo, expanded. We review when deep learning wins, when it loses, which specific finance use cases have succeeded, and where the field is heading.

Discovery Question

You have three datasets: (a) 100,000 satellite images of crop fields with yield labels; (b) 100,000 news articles with binary "stock up / stock down" next-day labels; (c) 100,000 loan applications with tabular features and default/no-default labels. For each, guess whether deep learning or gradient boosting wins—and why.

When Deep Learning Wins

Deep learning dominates whenever two conditions hold: the inputs are unstructured (images, text, audio, sequences) and the training data is abundant. Three broad domains illustrate the pattern.

Images. Convolutional neural networks (CNNs) extract spatially local features (edges, textures, shapes) through trained filters. No hand-designed features beat learned ones on ImageNet. Applications in finance include document analysis (invoice OCR, contract classification), chart pattern recognition (on candlestick images), and satellite-based commodity forecasting (parking-lot occupancy, shipping traffic).

Text. Transformer-based language models encode rich representations of meaning. In finance, they parse news articles for sentiment, extract facts from SEC filings, classify earnings-call transcripts. BloombergGPT (2023), FinBERT, and custom models built on BERT/GPT architectures are increasingly integrated into research workflows.

Sequences. Recurrent networks (LSTMs, GRUs) and transformers model sequences where order matters. In finance, this includes time-series forecasting, limit-order-book microstructure, and sequence-to-sequence prediction of trade flows.

When Deep Learning Loses

On clean tabular data with moderate sample sizes, gradient-boosted decision trees (XGBoost, LightGBM, CatBoost) usually match or beat neural networks. Grinsztajn, Oyallon, and Varoquaux (2022) published a systematic empirical study of tabular benchmarks, showing that tree-based methods consistently outperform neural networks on medium-sized tabular data (fewer than ∼50,000 rows).

Several reasons explain this. Decision trees are natively invariant to monotonic transformations of features—they do not care whether you feed them raw dollars or log-dollars. Neural networks must learn the transformation. Decision trees handle missing values natively and gracefully.

Neural networks need explicit imputation. Most importantly, decision trees have the right inductive bias for tabular data: they look for axis-aligned splits, which closely match the kind of piecewise-constant patterns common in business data.

Definition: Where Deep Learning Excels vs. Loses

DL excels when:

- Inputs are unstructured: images, text, audio, sequences.
- Training data is abundant (> 100k examples).
- Feature engineering is hard or infeasible.
- The target function is highly nonlinear with many interactions.

DL loses (gradient boosting wins) when:

- Inputs are structured tabular features.
- Training data is modest (< 50k rows).
- Features are heterogeneous (numeric + categorical with many levels).
- Interpretability is a regulatory requirement.

Specific Finance Use Cases

1. News sentiment for trading signals. BloombergGPT, FinBERT, and custom models process thousands of news articles per second, outputting sentiment scores that feed into short-horizon trading strategies. AUC on labeled data routinely exceeds 0.85. Latency and cost matter: a tier-1 trading firm processes 1M articles per day, requiring GPU clusters.

2. Chart pattern recognition. CNNs trained on candlestick charts have been used to classify patterns (head-and-shoulders, triangle, flag) and to forecast short-term price moves. Results are mixed; profitable deployments require careful data engineering and are often combined with traditional signals.

3. Limit-order-book modeling. Deep learning has achieved state-of-the-art results on market microstructure data. Models predict the direction of the mid-price from book snapshots, using CNNs, LSTMs, or transformers. Latency requirements are brutal (microseconds), so model size and execution speed constrain architecture.

4. Fraud detection. Neural networks and autoencoder-based anomaly detectors complement rule-based and gradient-boosted fraud systems. Deep models excel on complex sequence patterns (e.g., account-takeover detection from login sequences) where hand-crafted rules miss subtle cues.

5. Credit scoring. Tabular and interpretable by regulation. Gradient boosting and logistic regression dominate. Neural networks are sometimes used for feature extraction from non-tabular auxiliary data (e.g., embedding transaction sequences) that feeds into a traditional scoring model.

6. Return forecasting. Deep models for return prediction have a checkered history. Naive applications overfit spectacularly. Success requires very careful validation (walk-forward only), aggressive regularization, and integration with traditional factor models. Lopez de Prado's work and the recent "Forest Drift" methodology illustrate the challenges.

The Limits: Interpretability, Data, and Cost

Interpretability. A regulated credit-scoring model must explain, for every declined applicant, why they were declined. Gradient boosting supports feature-attribution methods (SHAP, LIME)

that regulators accept. Deep networks are harder to interpret; integrated gradients and attention-based explanations exist but are less standardized.

Data requirements. Deep learning eats data. A 100-million-parameter transformer trained on 10,000 examples is a recipe for overfitting. Even with state-of-the-art regularization, small data limits what deep learning can deliver.

Compute cost. Training a modest deep network on GPU costs a few dollars. Training a state-of-the-art language model costs tens of millions. Finance firms must weigh inference latency (every basis point matters in high-frequency trading) against model capability.

Robustness to distribution shift. Finance data is non-stationary by nature—market regimes change. Deep models trained on one regime can fail catastrophically in another. Classical models with fewer parameters are often more robust under shift, even if less accurate in-distribution.

Modern Trends: Transformers and Beyond

The transformer architecture (Vaswani et al., 2017) dominates NLP and is increasingly applied to tabular data (TabNet, FT-Transformer), time series (Informer, TimesNet), and multi-modal finance (combining text, images, and numeric data). Self-supervised pre-training—first learning from unlabeled data, then fine-tuning on the specific task—has enabled large foundation models that transfer across problems.

Two trends matter for finance over the next few years.

1. Foundation models for finance. BloombergGPT (2023) is trained on a mix of general web text and Bloomberg's proprietary financial data. Similar efforts (FinGPT, InvestLM, others) aim to produce pretrained backbones that can be fine-tuned cheaply for specific tasks. The economics are compelling: train once, deploy many times.

2. Neural-symbolic hybrids. Interpretability requirements in finance motivate hybrid systems that use neural networks for representation extraction and symbolic rules or tree models for final decisions. Examples include neural topic models feeding into a logistic regression classifier, or transformer-based feature embeddings feeding into XGBoost. These hybrids often beat pure neural or pure tree models in production.

Common Misconceptions about DL in Finance

(1) "Deep learning will replace all financial models." It will not. Many financial tasks are tabular, small-data, or interpretability-constrained, and these will continue to favor gradient boosting, regression, and rule-based systems.
(2) "Deeper models always win." On unstructured big data, yes. On tabular small data, often no. Always test a gradient boosting baseline before committing to deep learning.
(3) "Transformers beat CNNs and LSTMs in all cases." Transformers dominate NLP and many vision tasks at scale, but CNNs remain highly competitive on smaller image problems and LSTMs are still competitive on univariate time-series forecasts.

Worked Examples

Worked Example 1: When to Use DL for a Finance Problem

A quantitative trading firm is evaluating four projects. Rank them by how strongly they favor deep learning over gradient boosting.

1. **Credit-scoring model** for a retail bank. 50k tabular rows of borrower data. Interpretability required by regulator. → Gradient boosting wins strongly.
2. **Sentiment signal** from 10 million news headlines. Task: classify bullish/bearish. → Deep learning wins strongly (pre-trained transformer + fine-tuning).
3. **Mid-price prediction** from limit-order-book snapshots. 500 million training examples. Latency budget 50 microseconds. → Deep learning wins, but architecture is constrained by latency (small CNN or LSTM).
4. **Macro factor prediction** from 60 years of monthly economic indicators (720 rows). → Gradient boosting or linear models win—too little data for deep learning to find patterns reliably.

Worked Example 2: A Hybrid Neural-Symbolic Fraud System

A payment processor builds a fraud detection system. Requirements: process 100,000 transactions per second, decide within 50 ms, provide human-readable explanations for blocked transactions.
Architecture:

1. **Transaction sequence encoder:** a small LSTM (2 layers, 64 hidden units) takes the user's last 20 transactions and outputs a 32-dim embedding.
2. **Device/network features:** 100 hand-crafted features (device fingerprint, IP reputation, etc.).
3. **Combined classifier:** LightGBM takes the 32-dim LSTM embedding concatenated with the 100 hand-crafted features.

This hybrid beats both pure-LightGBM (which cannot access sequence patterns) and pure-LSTM (which cannot leverage hand-crafted features). The LSTM contributes rich sequence representations; LightGBM contributes interpretability and fast inference. Explanations for blocked transactions come from LightGBM's feature attributions plus a sequence-level attention visualization from the LSTM.

(c) Joerg Osterrieder 2026

Historical Background: BloombergGPT and Finance Foundation Models (2023)

In March 2023, Bloomberg Research announced BloombergGPT, a 50-billion-parameter language model trained on a mix of general web text and Bloomberg's proprietary financial corpus (FinPile)—a trillion tokens of news, research, filings, and market data. The model was trained on 64 AWS p4d.24xlarge instances (512 A100 GPUs) over 53 days at a cost estimated in the low millions of dollars.

BloombergGPT reportedly outperformed open-source models on finance-specific tasks (sentiment classification, named-entity recognition, financial Q&A) while matching general-purpose performance on standard benchmarks. The release triggered a wave of finance-focused foundation models: FinGPT, InvestLM, FinBERT variants, and others.

The deeper story is about specialization. General-purpose language models (GPT-4, Llama, Gemini) are extraordinary but can miss finance-specific nuances: what "long" means, how "synthetic CDO squared" differs from plain CDOs, why a 10-basis-point spread move matters. Domain-specialized models encode these nuances in their weights. For finance firms, the question is no longer "should we use an LLM?" but "which LLM, and how do we fine-tune it on our proprietary data?"

Problem 8.1 (Easy) ⋆

For each of the following, say whether deep learning or gradient boosting would be your first choice: (a) image-based defect detection on manufacturing line photos, (b) default prediction from 30k tabular loan applications, (c) earnings-call sentiment from 500k transcripts, (d) macroeconomic recession prediction from 50 years of monthly indicators.
Solution: see Appendix.

Problem 8.2 (Easy) ⋆

Name three properties of tabular data that make gradient boosting usually outperform neural networks on that data type.
Solution: see Appendix.

Problem 8.3 (Medium) ⋆⋆

A portfolio manager wants to use a pretrained transformer to generate sentiment signals from 10 million news articles per day. Describe, at a high level, a pipeline from raw news feed to tradeable signal. Identify two places where overfitting or data leakage could sneak in.
Solution: see Appendix.

Problem 8.4 (Medium) ⋆⋆

Explain why convolutional networks are a better fit than plain MLPs for image data. Use the terms "translation invariance," "parameter sharing," and "local receptive fields" in your explanation.
Solution: see Appendix.

(c) Joerg Osterrieder 2026

Problem 8.5 (Hard) ⋆⋆⋆

A finance firm wants to build an end-to-end deep learning system for equity return forecasting from (a) daily market features, (b) SEC filings, (c) earnings-call transcripts, (d) analyst reports. Sketch an architecture that combines these modalities. Specify: (i) how each modality is encoded, (ii) how the encodings are fused, (iii) how regularization and validation are handled to prevent overfitting, (iv) how you would obtain explanations for every prediction.
Solution: see Appendix.

Problem 8.6 (Hard) ⋆⋆⋆

Review Grinsztajn, Oyallon, and Varoquaux (2022), "Why do tree-based models still outperform deep learning on tabular data?". Summarize the three main findings of the paper. For each, explain how the observed weakness could, in principle, be addressed by a modified neural architecture.
Solution: see Appendix.

Closing the Loop

This handout has built a complete picture of deep learning, starting from the linear models that fail on XOR (Section 1), through the perceptron (Section 2), multilayer perceptrons (Section 3), backpropagation (Section 4), overfitting (Section 5), regularization (Section 6), training details (Section 7), and finally finance use cases (Section 8).

The appendix that follows contains complete solutions for all 44 practice problems. As with the other handouts in this course, solve each problem yourself before reading the solution. The value is in the struggle; the solutions are there to check your reasoning or to unblock a step you truly cannot see.

Key Takeaway: Deep learning is a powerful hammer—but not every problem is a nail; pair the right architecture with the right problem, and deep learning can solve things classical methods cannot touch.

A. Solutions to Practice Problems

Section 1: From Linear to Nonlinear

Problem 1.1 (Easy). (a) Linear regression represents only functions of the form $y = a + bx$; $y = x^2$ is not linear in x, so no choice of a, b fits it exactly across a range of x. (b) The XOR function's two classes sit at opposite corners of a square, and any straight line misclassifies at least one corner; logistic regression is linear in its inputs. (c) Neural networks are compositions of linear-then-nonlinear layers, which can represent any continuous function given enough hidden units—the universal approximation property.

Problem 1.2 (Easy). Pairwise products: $\binom{3}{2} = 3$ terms (x_1x_2, x_1x_3, x_2x_3). If we also need triple-wise: $\binom{3}{3} = 1$ term. Total interaction features for 3 inputs: $3+1 = 4$. For general d inputs, all k-way interaction terms up to order d give $\sum_{k=2}^{d} \binom{d}{k} = 2^d - d - 1$, which grows exponentially. Deep learning avoids this combinatorial explosion: a hidden layer with m ReLU neurons can represent many interactions with only $O(dm)$ parameters rather than $O(2^d)$.

Problem 1.3 (Medium). Three-way interaction terms: $\binom{20}{3} = 1140$.
Neural network parameters: hidden layer has $20 \cdot 16 + 16 = 336$ params. Output has $16 + 1 = 17$ params. Total $= 353$.
The neural network is more economical by a factor of $1140/353 \approx 3.2$ and also captures interactions of orders 2, 4, 5, ... as a side effect—not just three-way. As input dimension grows, the advantage of the neural network over explicit interaction features grows exponentially.

Problem 1.4 (Medium). With no activations, $h^{(5)} = W_5W_4W_3W_2W_1x = Wx$ where $W = W_5W_4W_3W_2W_1$. The composition collapses to a single linear transformation. This shows that nonlinear activation functions are not optional decorations—without them, depth adds zero representational power. Any function a 5-layer linear network can compute, a 1-layer linear network can compute with the same expressiveness.

Problem 1.5 (Hard). *Universal Approximation Theorem (informal):* A feedforward network with a single hidden layer containing a finite number of units with any continuous, non-constant, bounded, monotonically increasing activation function can approximate any continuous function on a compact subset of $\mathbb{R}^n$ to arbitrary accuracy (Cybenko 1989, Hornik 1991).
Why UAT does not mean deep networks always win:

1. **Representation is not optimization.** The theorem guarantees weights exist; it does not guarantee that gradient descent will find them efficiently. Deep optimization is non-convex and hard.
2. **Parameter efficiency.** To approximate a deep hierarchy of features (e.g., edges to textures to objects) with one hidden layer may require exponentially many units. Deeper networks can achieve the same accuracy with many fewer total parameters.
3. **Generalization is not representation.** A model class that can represent any function will also represent noise. Without regularization and sufficient data, UAT is a liability, not an asset.

Section 2: The Perceptron

Problem 2.1 (Easy). $z = 0.5(2) + (-1.0)(1) + 2.0(0.5) + (-0.3) = 1.0 - 1.0 + 1.0 - 0.3 = 0.7$. Since $z = 0.7 \geq 0$, the step function outputs 1.

Problem 2.2 (Easy). Many valid answers. A simple one: $w = (1, 1)$, $b = -0.5$. Verify:

- $(0, 0)$: $z = -0.5 < 0 \to 0$. ✓

- $(0, 1)$: $z = 1 - 0.5 = 0.5 > 0 \to 1$. ✓
- $(1, 0)$: $z = 1 - 0.5 = 0.5 > 0 \to 1$. ✓
- $(1, 1)$: $z = 2 - 0.5 = 1.5 > 0 \to 1$. ✓

Problem 2.3 (Medium). Current: $w = (1, 1)$, $b = -1.5$. Input $x = (2, 0)$, target $y = 0$. Compute: $z = 1(2) + 1(0) - 1.5 = 0.5$. Step activation: $\hat{y} = 1$. Error: $y - \hat{y} = -1$.

Update with $\eta = 0.5$: $w \leftarrow (1, 1) + 0.5(-1)(2, 0) = (1-1, 1-0) = (0, 1)$. $b \leftarrow -1.5 + 0.5(-1) = -2.0$.

New weights: $w = (0, 1)$, $b = -2.0$.

Problem 2.4 (Medium). NAND outputs 1 for all inputs except $(1, 1)$. This is the complement of AND, so it is linearly separable (flip the sign of the AND hyperplane). Choose $w = (-1, -1)$, $b = 1.5$. Verify:

- $(0, 0)$: $z = 1.5 > 0 \to 1$. ✓
- $(0, 1)$: $z = -1 + 1.5 = 0.5 > 0 \to 1$. ✓
- $(1, 0)$: $z = -1 + 1.5 = 0.5 > 0 \to 1$. ✓
- $(1, 1)$: $z = -2 + 1.5 = -0.5 < 0 \to 0$. ✓

Problem 2.5 (Hard). *Proof sketch of perceptron convergence.* Let w_t be the weight vector after t mistakes. Each mistake on example (x, y) with $y \in \{-1, +1\}$ produces the update $w_{t+1} = w_t + yx$.

Lower bound on $w_t^\top w^*$: By induction, $w_{t+1}^\top w^* = w_t^\top w^* + y(w^*)^\top x \geq w_t^\top w^* + \gamma$. After M mistakes, $w_M^\top w^* \geq M\gamma$ (starting from $w_0 = 0$).

Upper bound on $\|w_t\|^2$: Because the update occurred on a mistake, $y(w_t)^\top x \leq 0$, so $\|w_{t+1}\|^2 = \|w_t\|^2 + 2y(w_t)^\top x + \|x\|^2 \leq \|w_t\|^2 + R^2$. After M mistakes, $\|w_M\|^2 \leq MR^2$.

Combining via Cauchy–Schwarz: $w_M^\top w^* \leq \|w_M\|\|w^*\| = \|w_M\|$. So $M\gamma \leq \|w_M\| \leq \sqrt{M}R$, giving $M \leq (R/\gamma)^2$. □

Section 3: Multilayer Perceptrons and Activations

Problem 3.1 (Easy). Hidden layer: $W_1 \in \mathbb{R}^{32\times10}$, $b_1 \in \mathbb{R}^{32}$. Params: $32 \cdot 10 + 32 = 352$. Output layer: $W_2 \in \mathbb{R}^{1\times32}$, $b_2 \in \mathbb{R}$. Params: $32 + 1 = 33$. Total: $352 + 33 = 385$ parameters.

Problem 3.2 (Easy). (1) **Gradient flow.** ReLU has derivative 1 for $z > 0$ and 0 for $z < 0$. In deep networks, gradients multiply through layers; sigmoid's max derivative is 0.25, which causes vanishing gradients. ReLU preserves gradient magnitude on its active path.

(2) **Sparsity.** Approximately half of ReLU activations are zero, producing sparse representations that are computationally efficient and often generalize better.

(3) **Computational speed.** $\max(0, z)$ is a single comparison; $1/(1 + e^{-z})$ requires an exponential, which is far more expensive per operation.

Problem 3.3 (Medium). Input: $z = (2.0, 1.0, -1.0)$.
Exponentials: $e^{2.0} = 7.389$, $e^{1.0} = 2.718$, $e^{-1.0} = 0.368$.
Sum: $7.389 + 2.718 + 0.368 = 10.475$.
Softmax probabilities:

- Class 1: $7.389/10.475 = 0.705$.
- Class 2: $2.718/10.475 = 0.259$.
- Class 3: $0.368/10.475 = 0.035$.

Sum $= 1.000$. ✓

Problem 3.4 (Medium). With identity activations, $h^{(3)} = W_3 h^{(2)} = W_3 W_2 h^{(1)} = W_3 W_2 W_1 x +$ (bias terms). Combining the linear pieces gives a single $W = W_3 W_2 W_1$ and a single $b = W_3 W_2 b_1 + W_3 b_2 + b_3$. The three-layer linear network equals a one-layer linear network.

Implication: Nonlinearities are what give depth its power. Without them, adding layers adds parameters but not expressiveness.

Problem 3.5 (Hard). *Representing continuous piecewise-linear functions with ReLU.*

Let $f : \mathbb{R} \to \mathbb{R}$ be continuous and piecewise linear with breakpoints $b_1 < b_2 < \cdots < b_K$ and slopes $s_0, s_1, \ldots, s_K$ on the $K + 1$ pieces. Define the shifted ReLU basis $\phi_k(x) = \text{ReLU}(x - b_k)$ for $k = 1, \ldots, K$ and constant function 1.

Claim: $f(x) = c + s_0 x + \sum_{k=1}^{K} (s_k - s_{k-1})\phi_k(x)$ for appropriate constant c.

Verification: for $x < b_1$, all ReLUs are 0, giving $f(x) = c + s_0 x$, the slope-s_0 piece. For $b_j < x < b_{j+1}$, only $\phi_1, \ldots, \phi_j$ are active; the slope becomes $s_0 + \sum_{k=1}^{j} (s_k - s_{k-1}) = s_j$ by telescoping. ✓

So K hidden ReLU neurons (one per breakpoint) plus a linear output neuron suffice, giving $K + 1$ hidden units counting the bias/constant path. For $\mathbb{R}^n$, the same construction works with hyperplane breakpoints, each corresponding to one ReLU of the form $\max(0, w^\top x - b)$.

Section 4: Loss Functions and Backpropagation

Problem 4.1 (Easy). Residuals: $2.0 - 2.1 = -0.1$, $4.0 - 3.9 = 0.1$, $5.0 - 5.2 = -0.2$. Squared: $0.01, 0.01, 0.04$. Sum: 0.06. MSE: $0.06/3 = 0.02$.

Problem 4.2 (Easy). For $y = 1$, $\hat{y} = 0.9$: loss $= -1 \cdot \log(0.9) - 0 \cdot \log(0.1) = -\log(0.9) \approx 0.105$.

For $y = 1$, $\hat{y} = 0.1$: loss $= -\log(0.1) \approx 2.303$.

The second loss is roughly $22\times$ larger because the predicted probability for the true class is much smaller. BCE punishes confident wrong predictions heavily (up to $+\infty$ as $\hat{y} \to 0$).

Problem 4.3 (Medium). Forward: $z = 2.0 \cdot 0.5 + (-1.0) = 0.0$. $\hat{y} = \sigma(0) = 0.5$. BCE loss: $L = -\log(0.5) \approx 0.693$.

Backward (using BCE + sigmoid simplification): $\dfrac{\partial L}{\partial z} = \hat{y} - y = 0.5 - 1 = -0.5$.

$$\frac{\partial L}{\partial w} = (\hat{y} - y) \cdot x = -0.5 \cdot 0.5 = -0.25. \qquad \frac{\partial L}{\partial b} = \hat{y} - y = -0.5.$$

Update with $\eta = 0.5$: $w \leftarrow 2.0 - 0.5(-0.25) = 2.125$. $b \leftarrow -1.0 - 0.5(-0.5) = -0.75$.

Problem 4.4 (Medium). Adam adapts the step size *per parameter* using a running estimate of the gradient's second moment (squared gradient). Parameters with historically large gradients get smaller effective step sizes; parameters with small gradients get larger. This fixes the main pathology of plain SGD on ill-conditioned landscapes: some directions need tiny steps and others need huge steps, and plain SGD uses one rate for all. Momentum in Adam additionally dampens oscillations along steep directions and accelerates along shallow ones.

Plain SGD (with momentum) is sometimes preferable when: (a) the asymptotic convergence point matters (e.g., for theoretical guarantees or reproducibility), (b) the loss surface is well-conditioned, (c) the final generalization matters more than early convergence speed—several papers have shown that SGD-trained models generalize slightly better than Adam-trained ones in certain image-classification benchmarks.

Problem 4.5 (Hard). *3-layer MLP with sigmoid hidden, sigmoid output, BCE loss.*

Forward: $z^{(1)} = W_1 x + b_1$, $h^{(1)} = \sigma(z^{(1)})$. $z^{(2)} = W_2 h^{(1)} + b_2$, $h^{(2)} = \sigma(z^{(2)})$. $z^{(3)} = W_3 h^{(2)} + b_3$, $\hat{y} = \sigma(z^{(3)})$.

BCE loss: $L = -y \log \hat{y} - (1 - y) \log(1 - \hat{y})$.

(c) Joerg Osterrieder 2026

Output layer:

$$\frac{\partial L}{\partial \hat{y}} = -\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}} = \frac{\hat{y}-y}{\hat{y}(1-\hat{y})}.$$

$$\frac{\partial \hat{y}}{\partial z^{(3)}} = \sigma'(z^{(3)}) = \hat{y}(1-\hat{y}).$$

$$\delta^{(3)} := \frac{\partial L}{\partial z^{(3)}} = \hat{y} - y.$$

The simplification is what makes BCE + sigmoid so pleasant.

Gradients for output layer: $\partial L/\partial W_3 = \delta^{(3)}(h^{(2)})^\top$, $\partial L/\partial b_3 = \delta^{(3)}$.

Hidden layer 2: $\delta^{(2)} = W_3^\top \delta^{(3)} \odot h^{(2)} \odot (1-h^{(2)})$. Gradients: $\partial L/\partial W_2 = \delta^{(2)}(h^{(1)})^\top$, $\partial L/\partial b_2 = \delta^{(2)}$.

Hidden layer 1: $\delta^{(1)} = W_2^\top \delta^{(2)} \odot h^{(1)} \odot (1-h^{(1)})$. Gradients: $\partial L/\partial W_1 = \delta^{(1)} x^\top$, $\partial L/\partial b_1 = \delta^{(1)}$.

Section 5: Why Deep Networks Overfit

Problem 5.1 (Easy). A 27-point gap (train 98%, test 71%) is classic overfitting. First action: apply early stopping based on validation loss, ideally combined with L2 regularization and dropout. If retraining is not an option, the next-best action is to collect more training data or simplify the architecture.

Problem 5.2 (Easy). (a) **Bias.** The model class cannot represent the true curve regardless of data. (b) **Variance.** Many parameters relative to data; predictions will swing wildly with different training samples. (c) **Irreducible noise.** No model can do better than the inherent uncertainty in the labels. (d) **Variance.** A tree of depth 30 on 500 rows memorizes leaves of size 1 or 2.

Problem 5.3 (Medium). $P/N = 120000/8000 = 15$. Above the rule-of-thumb threshold of 10. Expect noticeable overfitting without regularization. Apply:

- L2 weight decay with $\lambda \in [10^{-4}, 10^{-3}]$.
- Dropout with $p = 0.3$–0.5 after each hidden layer.
- Early stopping on validation loss with patience 10.
- If feasible, data augmentation (synthetic minority over-sampling, feature jittering).
- Consider reducing network size if performance permits.

Problem 5.4 (Medium). Train loss falling from 0.12 to 0.03 while val loss rises from 0.75 to 0.95 is textbook overfitting. Between epochs 50 and 100, the network is memorizing training-specific patterns that do not generalize. No useful learning occurs after epoch 50—the model is actively getting worse on held-out data. Stop at or before epoch 50.

Problem 5.5 (Hard). Belkin et al. (2019) documented that for many deep networks, test error follows a two-stage "double descent" curve: a classical U-shape below the interpolation threshold, followed by a second descent for highly over-parameterized models. Classical theory (VC bounds, bias-variance decomposition for linear models) predicts the first descent and the spike, but misses the second descent.

Reasons classical theory misses it: (1) VC bounds are worst-case over all functions a model could represent; they do not account for the specific solutions gradient descent actually finds. (2) SGD implicitly regularizes toward minimum-norm solutions in the interpolation regime. (3) Over-parameterized networks have loss landscapes with many global minima, most of which generalize well—a phenomenon called "benign overfitting."

Practical implication: Do not be afraid of very large networks. Combined with standard regularization, wider/deeper networks often generalize at least as well as moderately sized ones, and sometimes better.

Section 6: Regularization for Deep Networks

Problem 6.1 (Easy). Gradient of L2 penalty: $2\lambda w = 2 \cdot 10^{-3} \cdot 2 = 0.004$. Total gradient: $-0.05 + 0.004 = -0.046$.

Update: $w \leftarrow 2 - 0.01 \cdot (-0.046) = 2.00046$. The step is a hair in the direction of reducing data loss, slightly counteracted by weight decay's pull toward zero.

Problem 6.2 (Easy). The mask zeroes out positions 2 and 5. Keep and scale by $1/(1-p) = 1/0.6 = 1.667$ the remaining positions 1, 3, 4. Output: $(1.0 \cdot 1.667, 0, -0.2 \cdot 1.667, 2.0 \cdot 1.667, 0) = (1.667, 0, -0.333, 3.333, 0)$.

Problem 6.3 (Medium). Validation losses: $\{1.2, 1.1, 1.0, 0.95, 0.92, 0.91, 0.93, 0.94, 0.95, 0.97\}$.

Best-so-far tracker: updates at each epoch where validation loss reaches a new minimum. Minimums are at epochs 1, 2, 3, 4, 5, 6. After epoch 6 (loss 0.91), losses rise. Count epochs without improvement: epoch 7 (1 without improvement), 8 (2), 9 (3). At epoch 9, patience has been exhausted (3 epochs without improvement).

Early stopping triggers at epoch 9. The saved model is from epoch 6 (the one with $v^* = 0.91$).

Problem 6.4 (Medium). L2 regularization's objective is $L_{\text{data}} + \lambda\|w\|^2$. From a Bayesian viewpoint, the regularized loss is the negative log posterior under a Gaussian prior $p(w) \propto \exp(-\lambda\|w\|^2)$, which is $\mathcal{N}(0, 1/(2\lambda) \cdot I)$. Maximizing the posterior (MAP estimation) yields the L2-regularized estimate.

L1 regularization ($\lambda\|w\|_1$) corresponds to a Laplace (double-exponential) prior $p(w) \propto \exp(-\lambda\|w\|_1)$. Its MAP estimate encourages sparse solutions (many exact zeros) because the Laplace distribution has a sharp peak at zero.

Problem 6.5 (Hard). Consider linear regression $\hat{y} = w^\top x$ with MSE loss $L(w) = \frac{1}{2}\|Xw - y\|^2$. Gradient: $\nabla L = X^\top(Xw - y)$.

Gradient descent starting from $w_0 = 0$:

$$w_{t+1} = w_t - \eta X^\top(Xw_t - y) = (I - \eta X^\top X)w_t + \eta X^\top y.$$

Using eigendecomposition $X^\top X = U\Lambda U^\top$ with eigenvalues λ_i:

$$w_t = U(I - (I - \eta\Lambda)^t)\Lambda^{-1}U^\top X^\top y.$$

For each eigendirection i, the factor $(1 - (1 - \eta\lambda_i)^t)/\lambda_i$ acts on that component of $X^\top y$.

The ridge-regression (L2) solution is $w_\tau = (X^\top X + \tau I)^{-1}X^\top y = U(\Lambda + \tau I)^{-1}U^\top X^\top y$. In eigenbasis, each component has factor $1/(\lambda_i + \tau)$.

Comparing per-eigenvalue shrinkage: GD after t iterations gives $f_{\text{GD}}(\lambda_i) = (1 - (1 - \eta\lambda_i)^t)/\lambda_i$; ridge gives $f_{\text{ridge}}(\lambda_i) = 1/(\lambda_i + \tau)$.

For small $\eta\lambda_i$, both are approximately ηt or $1/\tau$, yielding equivalent effective regularization with $\tau \approx 1/(\eta t)$. So t iterations of gradient descent $\approx$ ridge regression with $\tau = 1/(\eta t)$. Stopping earlier increases effective regularization. □

Section 7: Training Tricks and Pitfalls

Problem 7.1 (Easy). He initialization for a layer with $d_{\text{in}} = 256$: variance $= 2/256 = 0.0078125$. Standard deviation $= \sqrt{0.0078125} \approx 0.0884$.

Problem 7.2 (Easy). All-zero initialization gives every neuron in a layer identical weights and identical gradients, so they update identically forever and remain duplicates—the network cannot break this symmetry.

Problem 7.3 (Medium). Val loss drops from 0.80 at epoch 10 to 0.76 at epoch 30, then stays at 0.76 at epochs 40 and 50. Patience is 10, so the scheduler requires 10 consecutive epochs without improvement to reduce the learning rate. The last improvement was at epoch 30

(c) Joerg Osterrieder 2026

(dropping from some earlier value to 0.76). Ten epochs later, at epoch 40, the patience is exhausted; the learning rate is reduced at epoch 40 (by factor 0.1).

Problem 7.4 (Hard). Cosine annealing: $\eta_t = 10^{-6} + 0.5(10^{-3} - 10^{-6})(1 + \cos(\pi t/100))$. With $A = 0.5(10^{-3} - 10^{-6}) \approx 4.995 \times 10^{-4}$:

- $t = 0$: $\cos(0) = 1$. $\eta_0 = 10^{-6} + A(1 + 1) = 10^{-6} + 2A \approx 1.000 \times 10^{-3}$.
- $t = 25$: $\cos(\pi/4) \approx 0.707$. $\eta_{25} \approx 10^{-6} + A(1 + 0.707) \approx 8.53 \times 10^{-4}$.
- $t = 50$: $\cos(\pi/2) = 0$. $\eta_{50} \approx 10^{-6} + A(1) \approx 5.00 \times 10^{-4}$.
- $t = 75$: $\cos(3\pi/4) \approx -0.707$. $\eta_{75} \approx 10^{-6} + A(0.293) \approx 1.47 \times 10^{-4}$.
- $t = 100$: $\cos(\pi) = -1$. $\eta_{100} \approx 10^{-6} + A \cdot 0 = 10^{-6}$.

The schedule starts at $\eta_0 \approx 10^{-3}$, decays smoothly following a half-cosine, and lands at $\eta_{\min} = 10^{-6}$.

Section 8: Deep Learning in Finance

Problem 8.1 (Easy). (a) **Deep learning** (images + large-ish data → CNN). (b) **Gradient boosting** (tabular + modest data → XGBoost/LightGBM). (c) **Deep learning** (text + large data → fine-tuned transformer). (d) **Gradient boosting or linear models** (tiny tabular data).

Problem 8.2 (Easy). (1) Decision trees are natively invariant to monotonic transformations of features (log, scaling). Neural networks must learn the transformations, costing capacity and data. (2) Decision trees handle heterogeneous features (numeric + categorical with many levels) and missing values natively. Neural networks need explicit encoding and imputation. (3) Decision trees have the right inductive bias for tabular data: axis-aligned splits match the piecewise-constant patterns common in business data.

Problem 8.3 (Medium). *Pipeline:*

1. **Ingestion:** stream 10M articles/day from news providers (Reuters, Bloomberg, AP).
2. **Preprocessing:** deduplicate, tokenize, truncate to 512 tokens, extract metadata (timestamp, tickers mentioned).
3. **Scoring:** pass each article through a fine-tuned transformer (FinBERT or similar) to produce a sentiment score in $[-1, +1]$.
4. **Ticker linkage:** for each mentioned ticker, compute an article-level score and append to a per-ticker time series.
5. **Aggregation:** for each ticker, compute a rolling window (say, 6 hours) of sentiment, producing a tradeable feature.
6. **Signal combination:** feed sentiment features into a strategy model (could be simple regression or another ML model).

Overfitting/leakage risks: (1) *Label leakage:* if training labels were derived from stock moves using a window that overlaps the test period, the sentiment model has implicitly seen the future. Strict temporal splits are essential. (2) *Data leakage via tickers:* if the transformer's pretraining data included similar articles from the evaluation period, performance on the held-out set is inflated. Use a pretrained model with known cutoff and evaluate only on data after the cutoff.

Problem 8.4 (Medium). Convolutional networks use **local receptive fields**: each filter sees a small spatial neighborhood (e.g., 3×3 pixels), capturing local patterns like edges. They use

parameter sharing: the same filter is applied at every spatial location, so a small filter detects its pattern anywhere in the image. This yields **translation invariance**: a cat in the top-left corner is recognized the same as a cat in the center. Plain MLPs have no such structure—each pixel has its own weights, so they cannot share pattern detectors and cannot leverage spatial locality. For an image with 1000×1000 pixels and 100 hidden units, an MLP needs $100 \times 10^6 = 10^8$ parameters in the first layer alone; a CNN with 32 filters of size 3×3 needs $32 \cdot 9 = 288$.

Problem 8.5 (Hard). *Multimodal return-forecasting architecture.*
(i) Per-modality encoders:

- **Daily market features:** a simple MLP or a small LSTM over a 20-day window, outputting a 64-dim embedding.
- **SEC filings:** chunk each filing into segments of 512 tokens; pass each through a finance-tuned transformer (FinBERT); aggregate with mean-pooling or attention over segments; output a 64-dim embedding.
- **Earnings-call transcripts:** similar to SEC filings, with optional speaker-aware attention to separate CEO commentary from analyst questions.
- **Analyst reports:** same transformer-based pipeline as filings, with an additional learned embedding for the analyst identity (to capture analyst-specific biases).

(ii) Fusion: concatenate the four 64-dim embeddings into a 256-dim vector. Pass through a small MLP (two hidden layers of 128 with ReLU + dropout) to produce a return forecast.
(iii) Regularization and validation:

- L2 weight decay $\lambda = 10^{-4}$ on all weights.
- Dropout $p = 0.3$ on fusion MLP layers.
- Strict walk-forward cross-validation: train on data up to time t, validate on $[t+1, t+60 \text{ days}]$, discard $[t+61, t+120]$ to let gradients settle, then move forward. Never evaluate on data that preceded training data.
- Early stopping with patience 10.
- Scale-normalize targets (returns) by rolling 60-day volatility so magnitudes are comparable.

(iv) Explanations: for each prediction, compute (a) integrated gradients with respect to each modality's input to identify which filings/transcripts/features contributed most; (b) attention weights within each encoder to highlight which text segments drove the filing/call embeddings; (c) a SHAP-style decomposition over the four modality embeddings at the fusion layer to quantify the relative contribution of market/filings/calls/analysts to the final prediction.

Problem 8.6 (Hard). *Summary of Grinsztajn et al. (2022):*
Finding 1: Tree-based models (XGBoost, Random Forest) consistently outperform neural networks on medium-sized tabular datasets, even after extensive tuning of both. The gap is most pronounced on datasets with 10k–50k rows and heterogeneous features.
Addressing via neural architecture: incorporate inductive biases from trees. TabNet (Arik and Pfister, 2019) uses sparse feature-wise attention to emulate the feature selection of trees. FT-Transformer (Gorishniy et al., 2021) embeds continuous features and uses transformer layers with categorical-aware tokenization. Neural oblivious decision trees (NODE) directly incorporate tree-like structure.
Finding 2: Neural networks struggle with uninformative features—irrelevant features hurt neural networks more than trees. Trees simply don't split on them; neural networks still use them as input.

Addressing via architecture: feature-sparsity-inducing regularization (L1 on first-layer weights), or explicit feature selection as a preprocessing step. LASSO-regularized input layers help.

Finding 3: Neural networks are too smooth—they fit globally continuous functions, while tabular data often has sharp, non-smooth conditional expectations (e.g., "if age $>$ 65 then discount"). Gradient descent biases neural networks toward smooth solutions.

Addressing via architecture: piecewise-linear or step-like activations (soft-sign, GELU variants, feature binning as preprocessing), or explicit tree-like branch structures within the network. Boosting-style residual networks (Alammar's "neural boosting") can approximate the additive structure of gradient-boosted trees.

(c) Joerg Osterrieder 2026