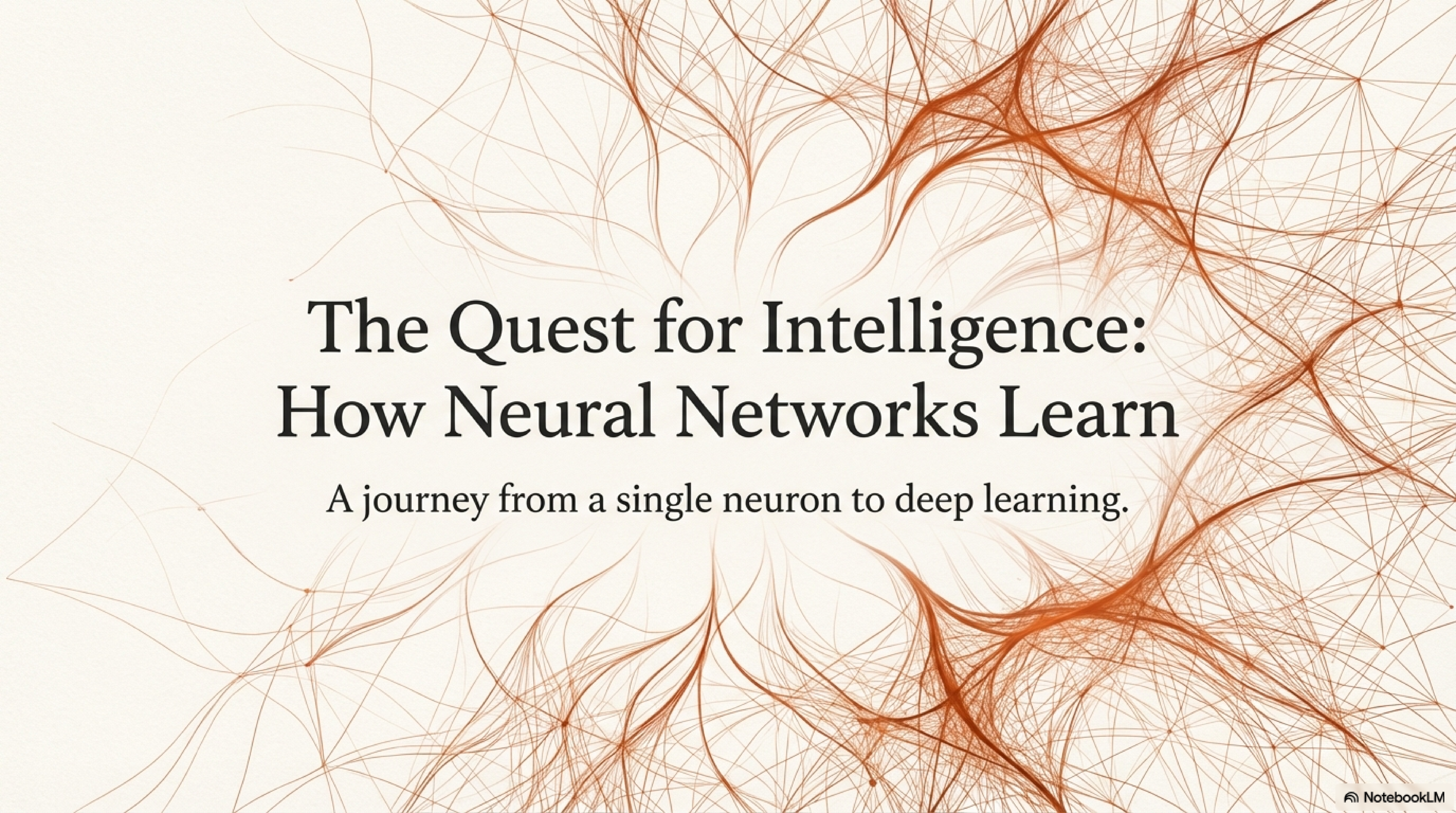


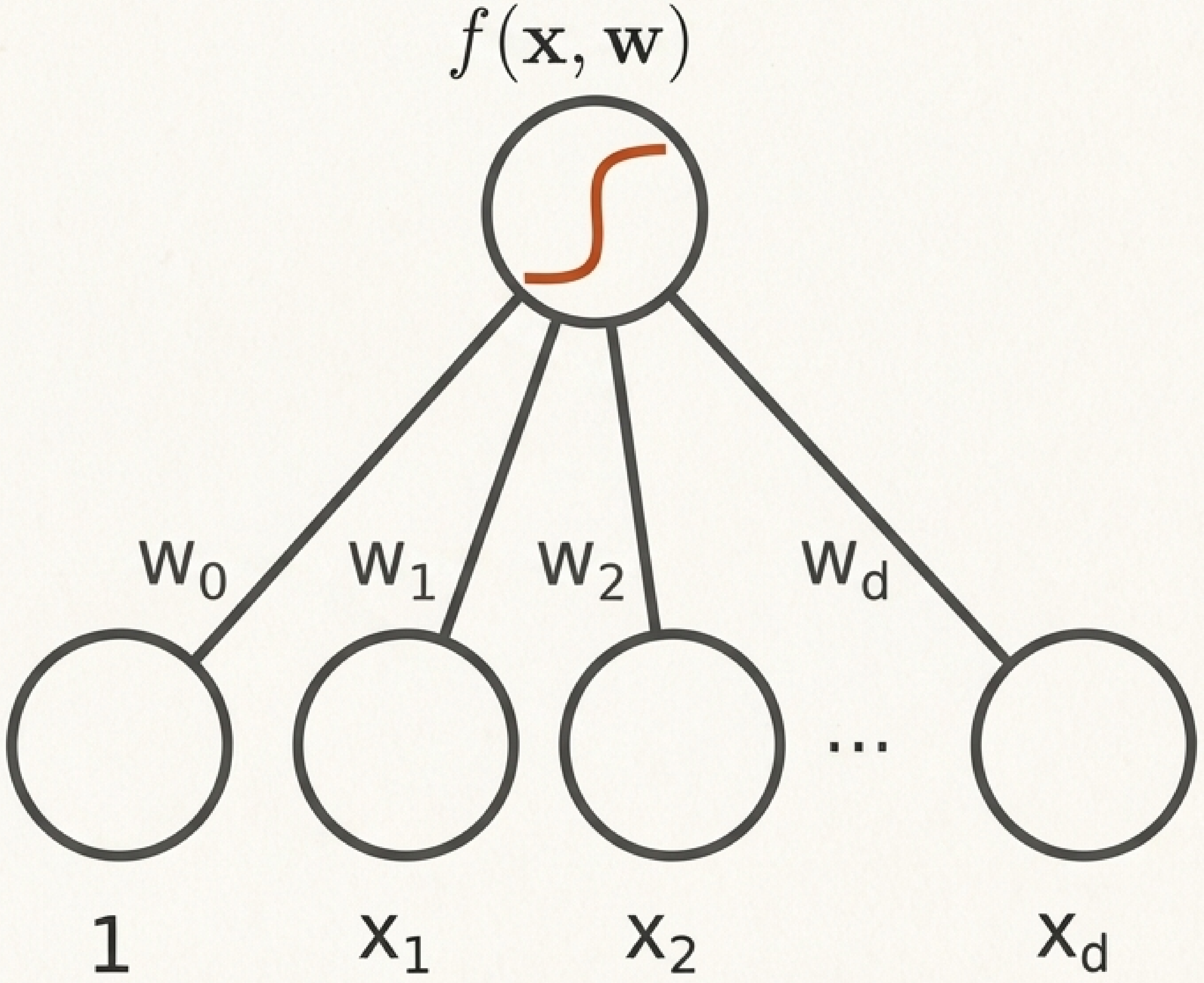
The Quest for Intelligence: How Neural Networks Learn

A journey from a single neuron to deep learning.

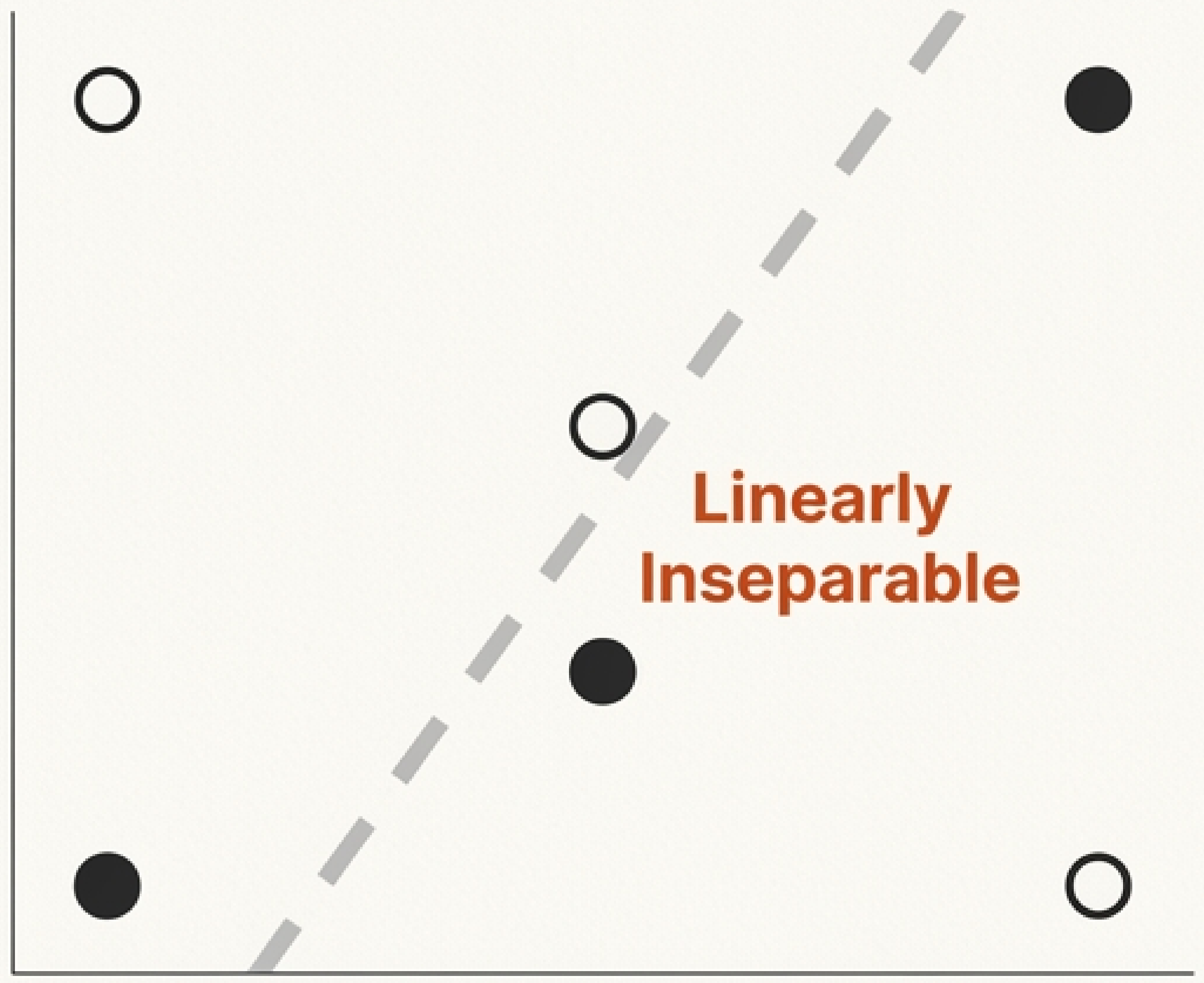
Our Journey Begins with a Single Neuron

We can think of Logistic Regression as a single neuron. It takes inputs, applies weights, and uses an activation function (`$\sigma$`) to make a decision.

$f(x, w) = \sigma(w^T * x)$

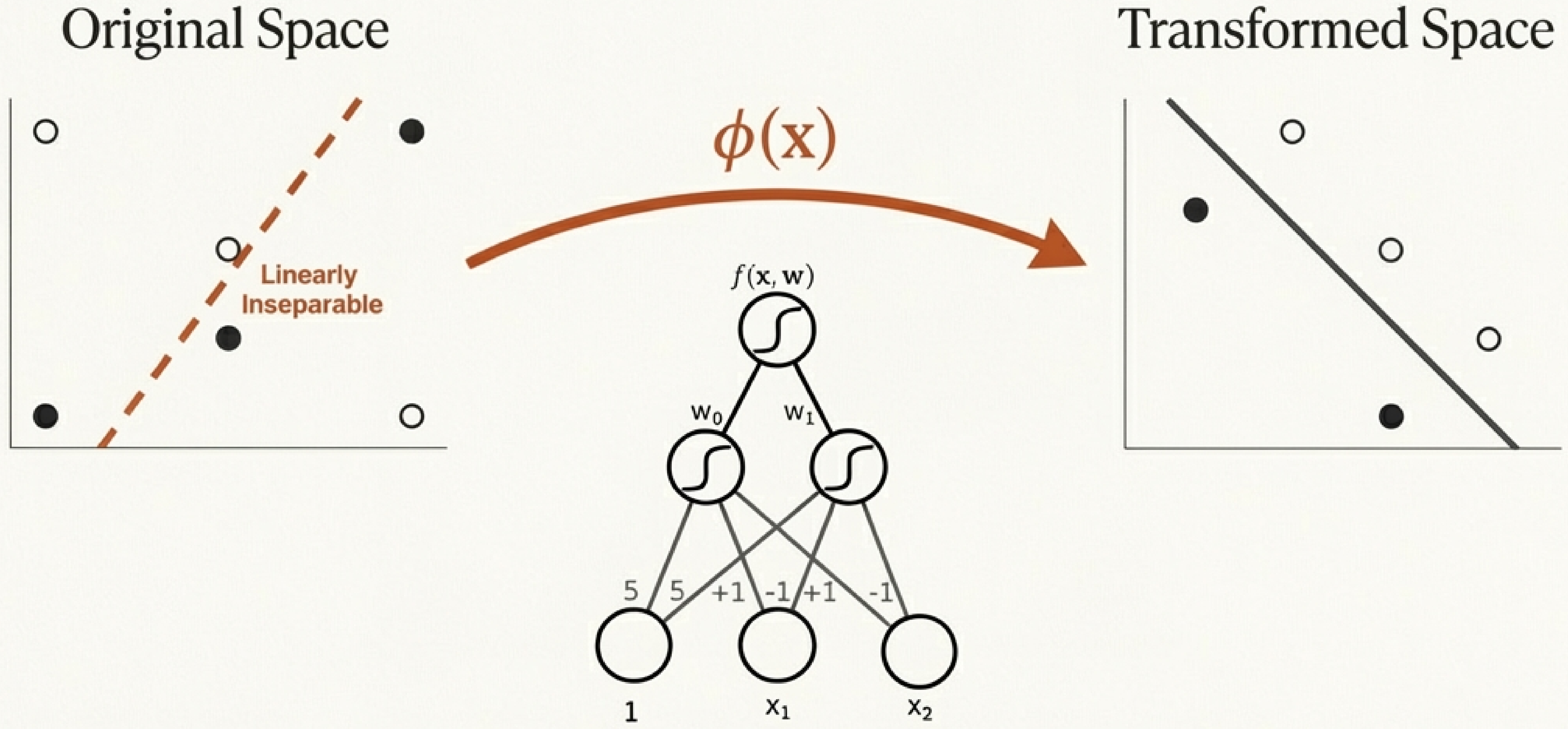


The Linear Barrier



Simple models are powerful, but they can only solve problems that are linearly separable. They fail when faced with a challenge like the XOR dataset.

A Clever Trick: Manually Bending Space

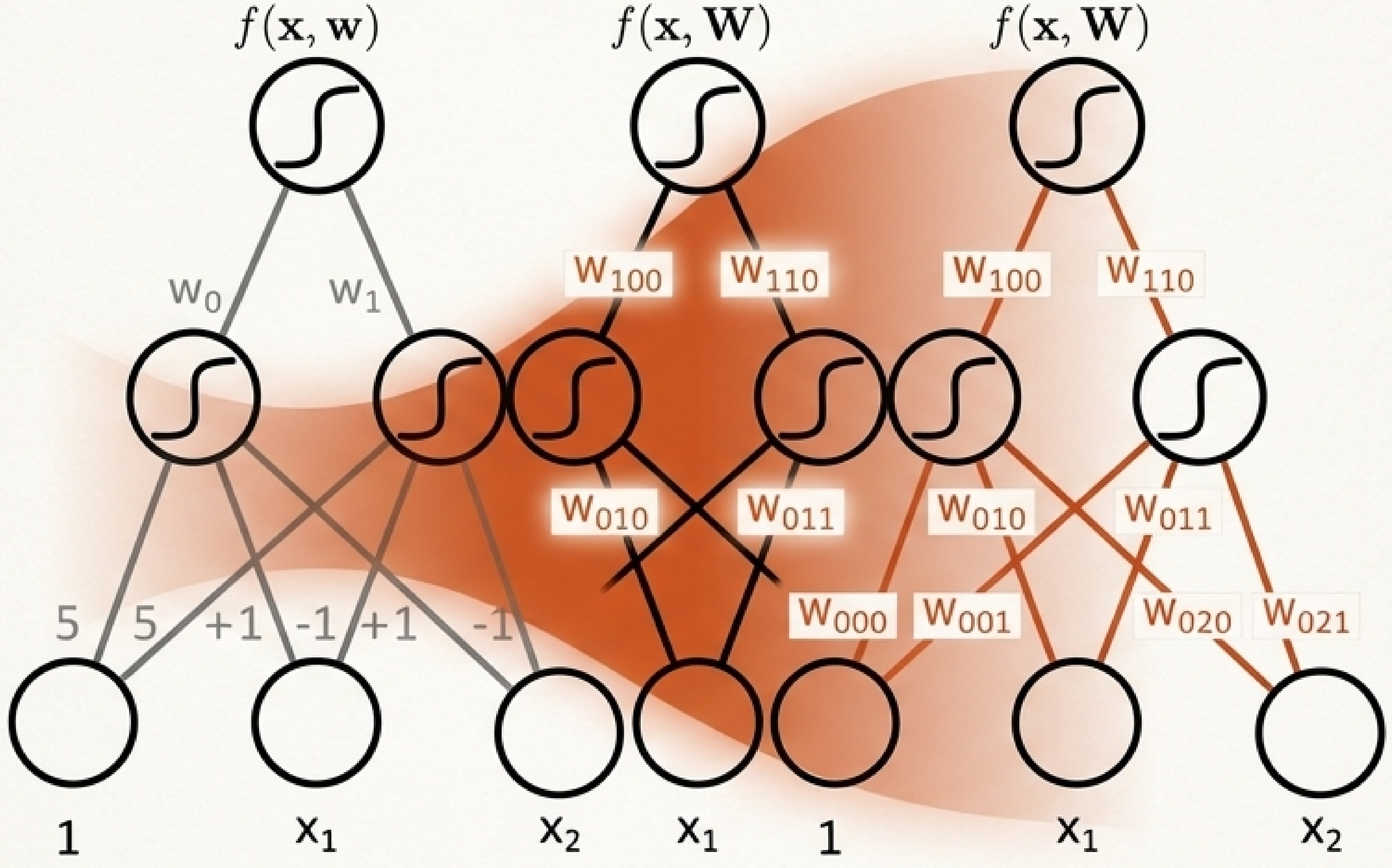


One solution is to apply a custom, non-linear transformation (ϕ(x)) to the data. This maps the problem into a new space where it *is* linearly separable. But this requires us to engineer the perfect ϕ by hand.

The Breakthrough: Learning the Transformation

The revolutionary idea: what if the network could learn the basis functions itself?

We can add a 'hidden layer' of neurons, whose sole job is to discover the best data transformation.



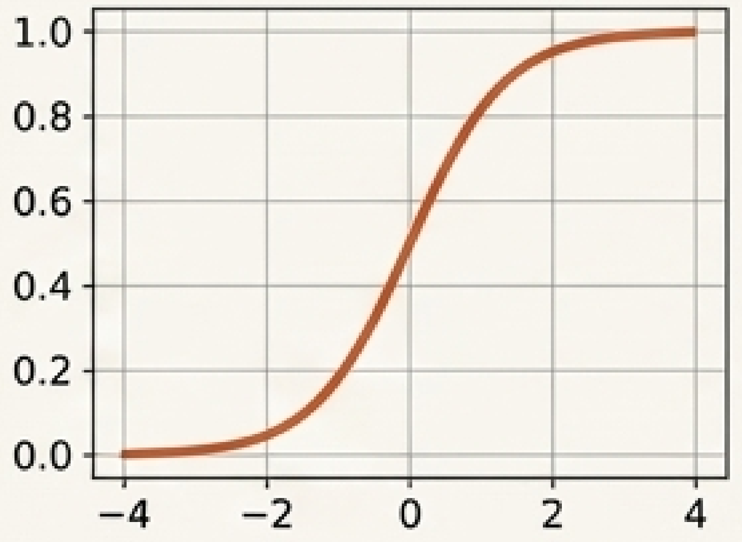
This is the core concept of end-to-end learning. We learn the features and the classifier simultaneously.

The Source of Power: Non-Linearity

What makes this possible? **Non-linear activation functions**. Without them, a multi-layer network collapses into a single linear transformation (` = `**W'x**`). Non-linearity allows us to learn truly complex functions.

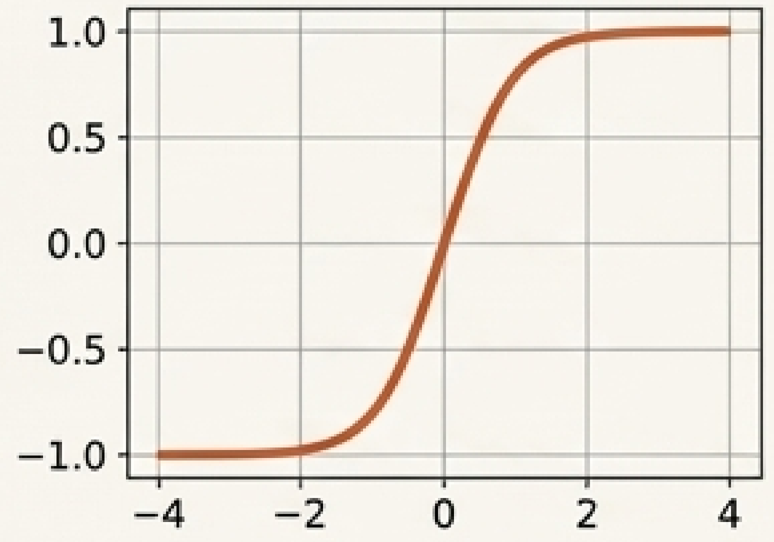
Sigmoid

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



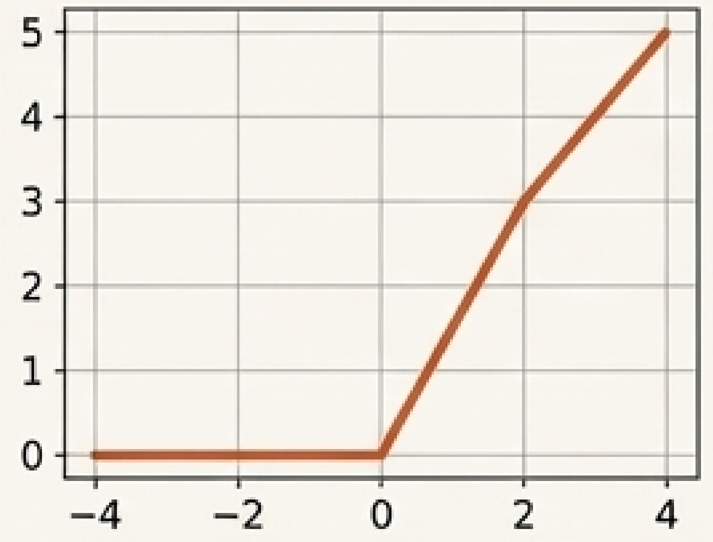
tanh

$$\tanh(x)$$



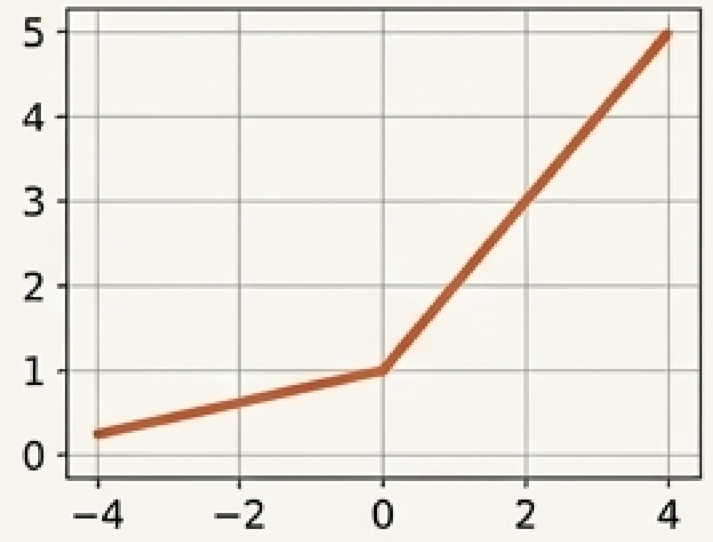
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$



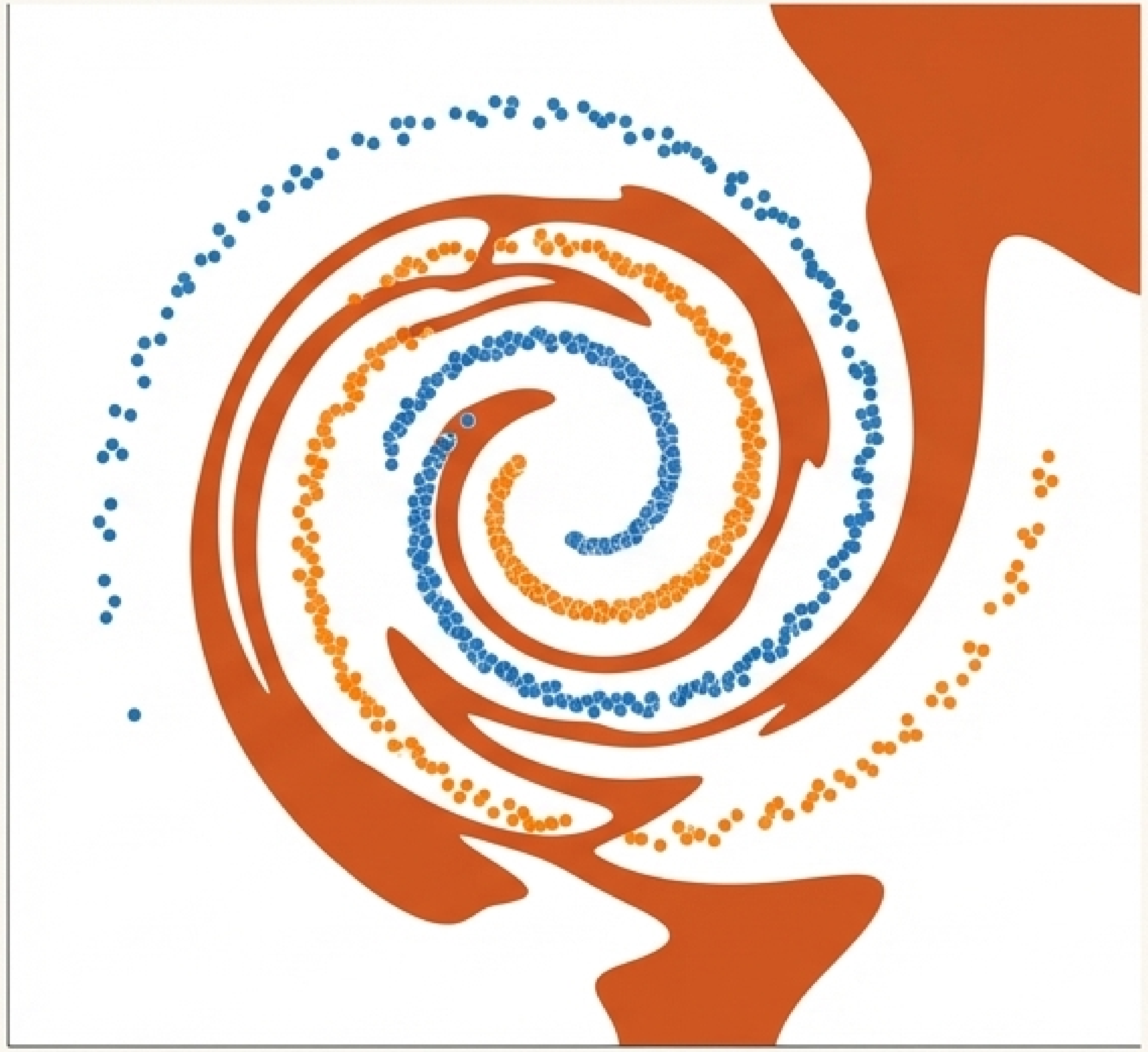
A Universal Approximator

The Good News

The Universal Approximation Theorem states that a network with just one hidden layer can approximate any continuous function arbitrarily well.

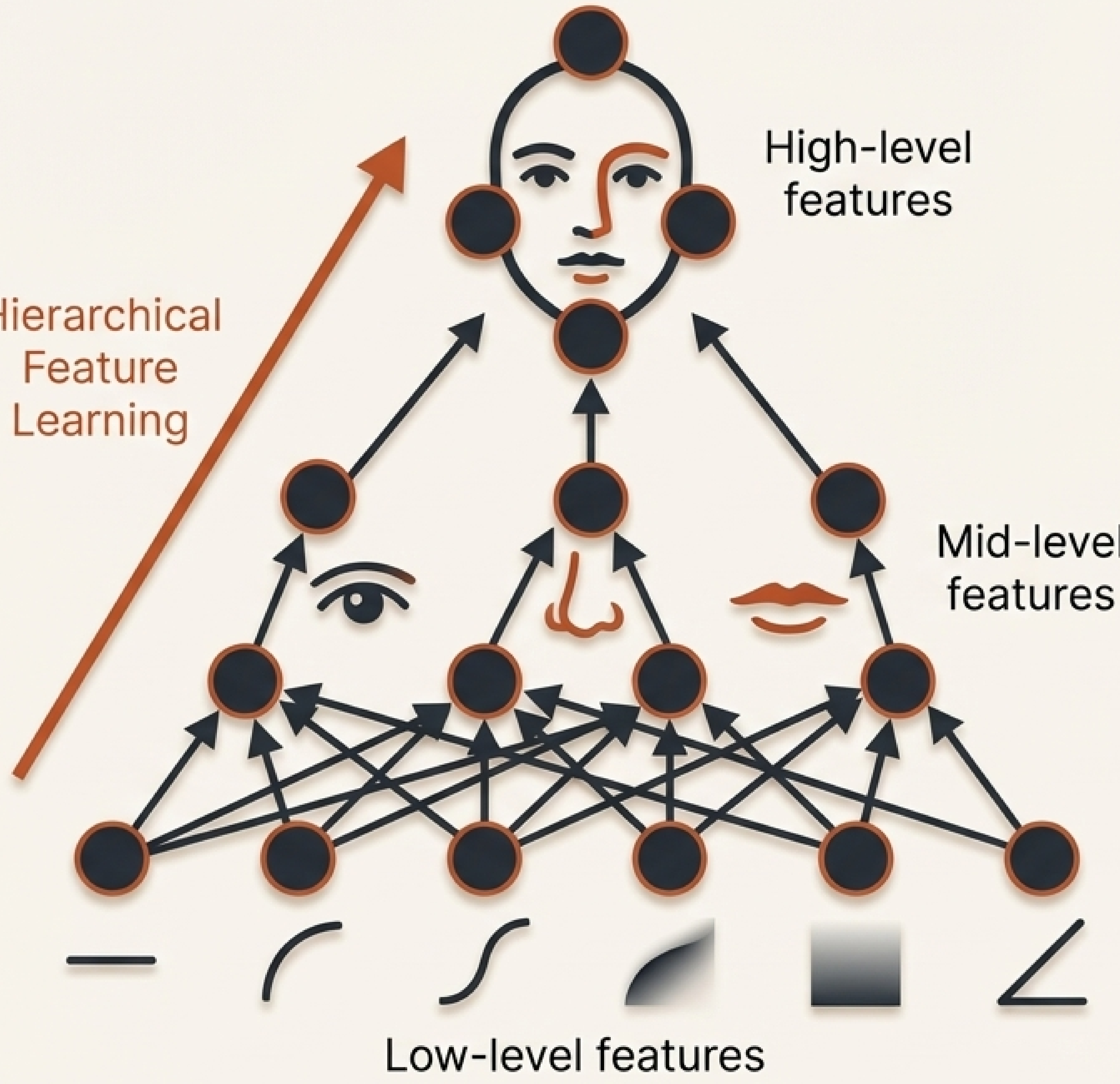
The Catch

The theorem doesn't tell us how to find the weights or how many hidden units we need. The potential is there, but unlocking it is the real challenge.

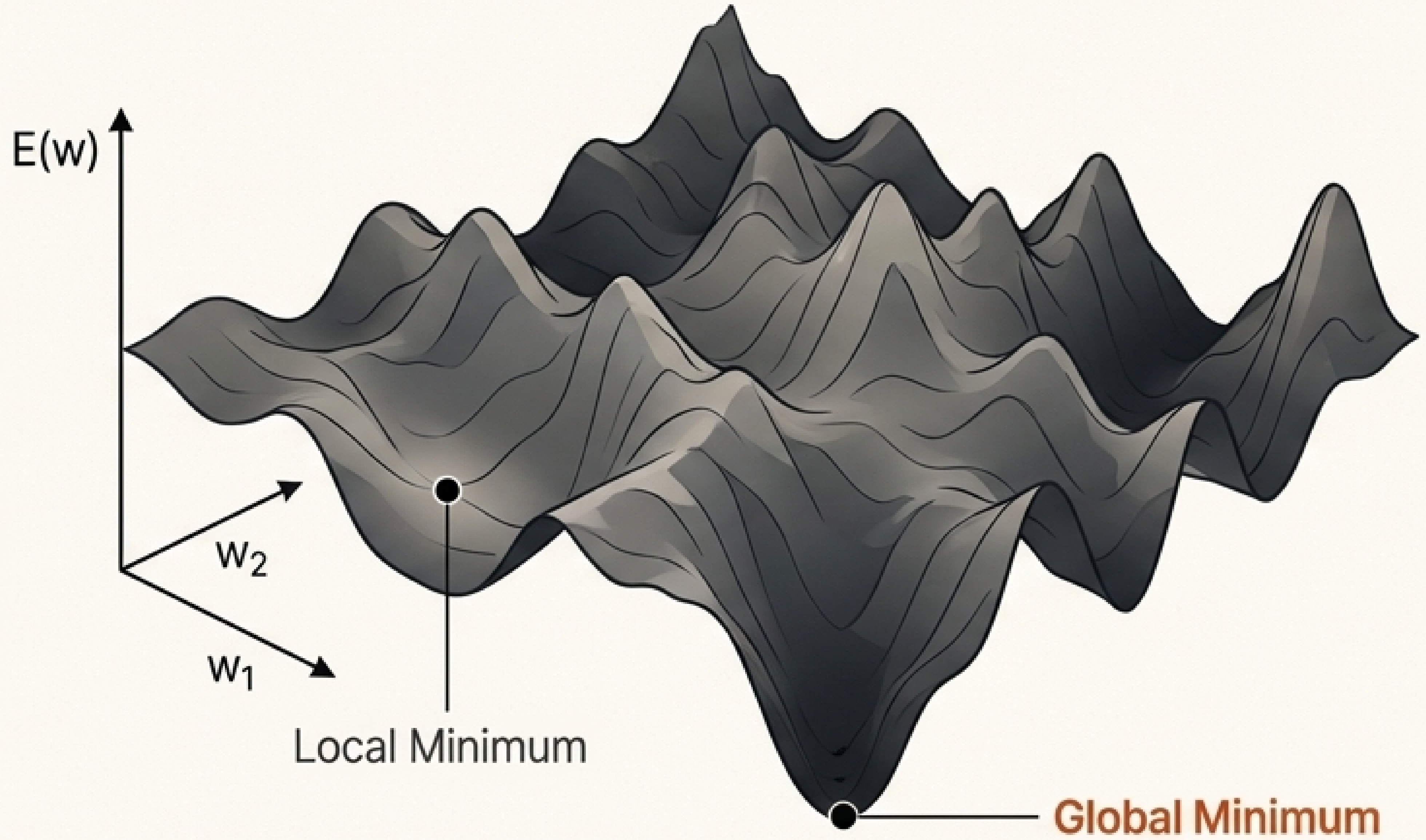


Why Go Deeper?

Deeper networks can be far more efficient. A function that requires an exponentially large single hidden layer might be represented compactly with multiple layers. This allows the network to learn a hierarchy of features—from simple edges to complex concepts.



The Ultimate Challenge: Finding the Optimal Weights

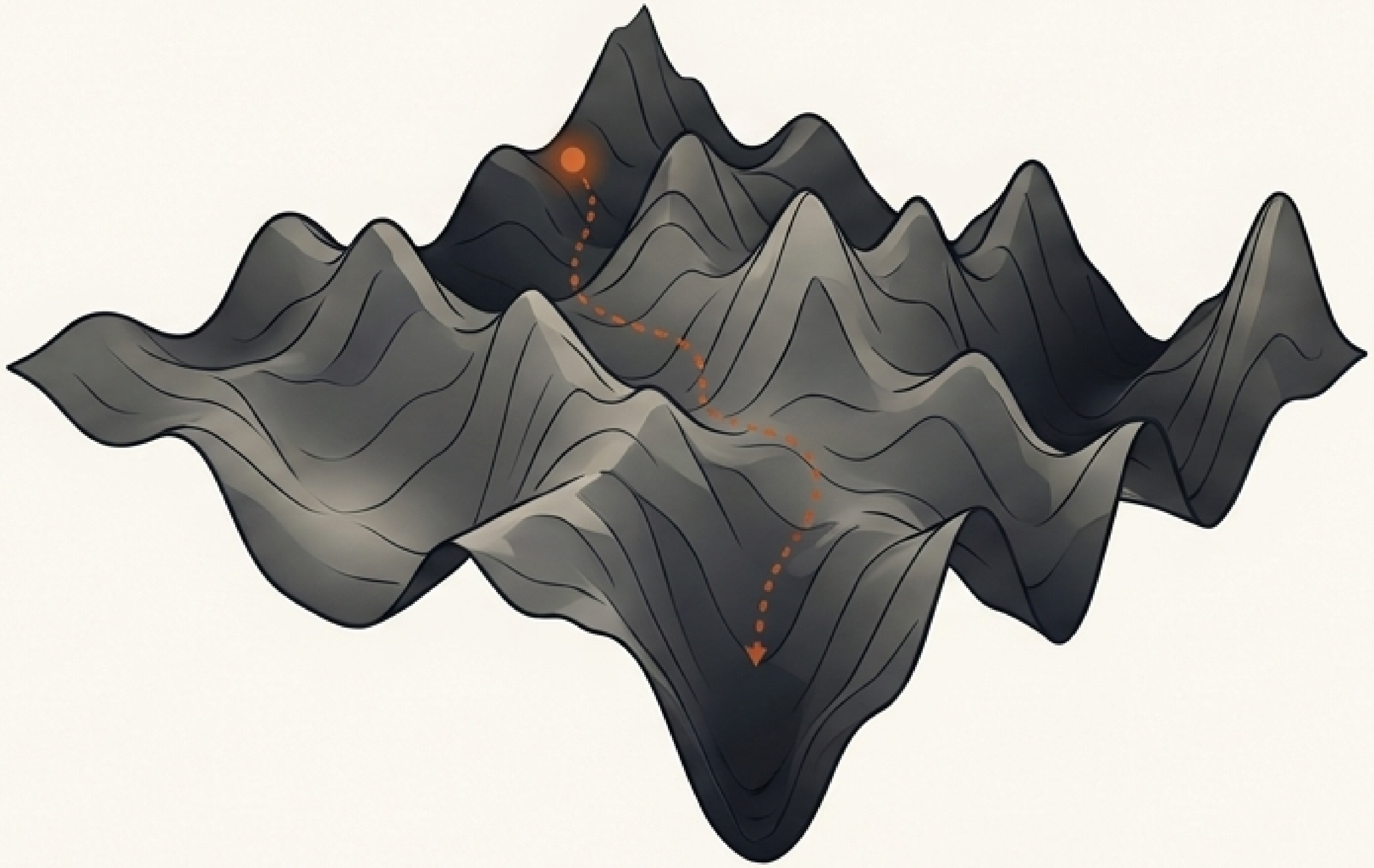


We train the network by defining a **loss function**, $E(W)$, that measures how wrong its predictions are. Our goal is to find the weights W that minimize this loss. The problem is that this function is often **highly non-convex**, a landscape of countless hills and valleys.

Navigating the Landscape with Gradient Descent

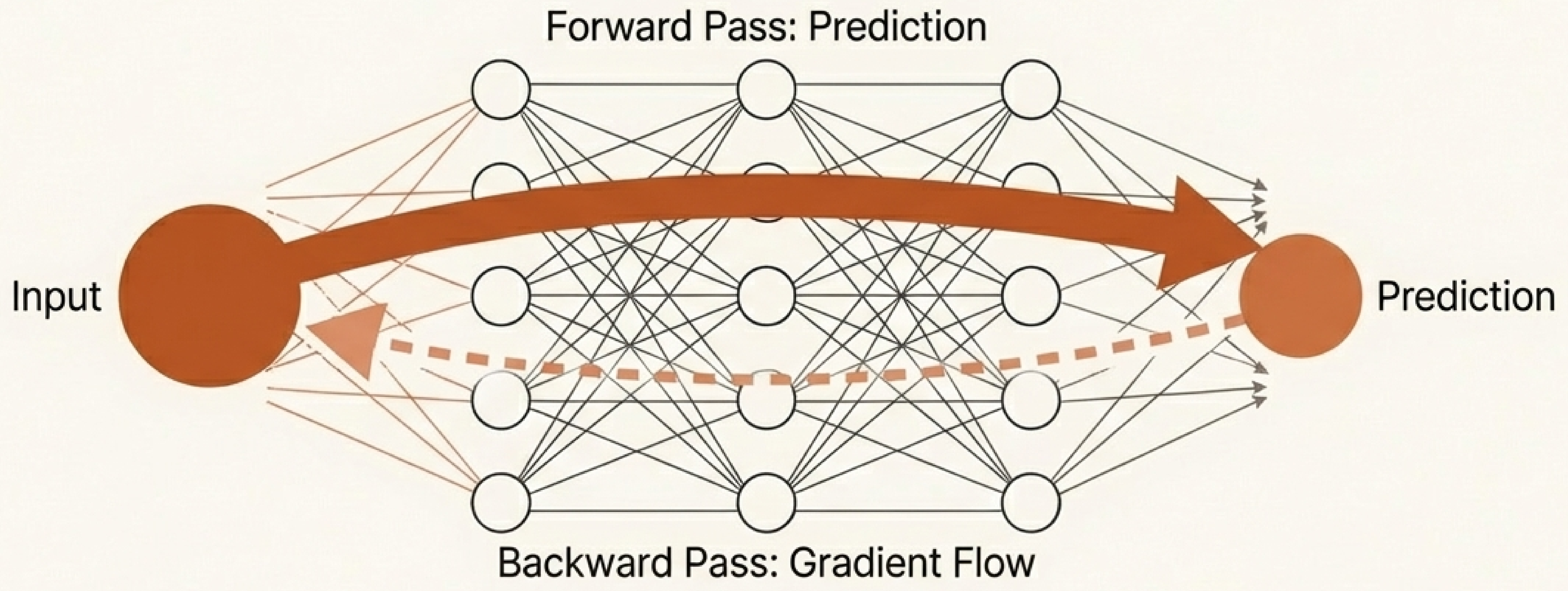
Our strategy is to “walk downhill.” We start with random weights and repeatedly take small steps in the direction of the steepest descent—the negative gradient.

$$W^{(new)} = W^{(old)} - \tau * \nabla WE(W^{(old)})$$



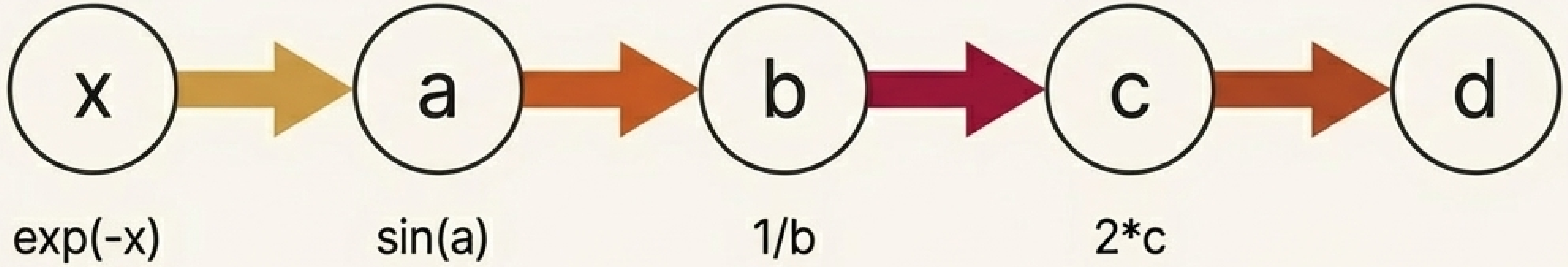
The Critical Question: How can we compute the gradient ∇WE for millions of parameters efficiently?

The Engine of Learning: Backpropagation



Numerical methods are too slow (O(|W|^2)), and symbolic methods are too complex. The answer is **Backpropagation**. It is an algorithm that computes the exact gradient for every parameter in the network with remarkable efficiency (O(|W|)).

How It Works: A Chain of Derivatives



Backpropagation is a clever application of the chain rule from calculus. The network is a giant composite function. To find the derivative with respect to an early parameter, we just multiply the local derivatives of each step along the path.

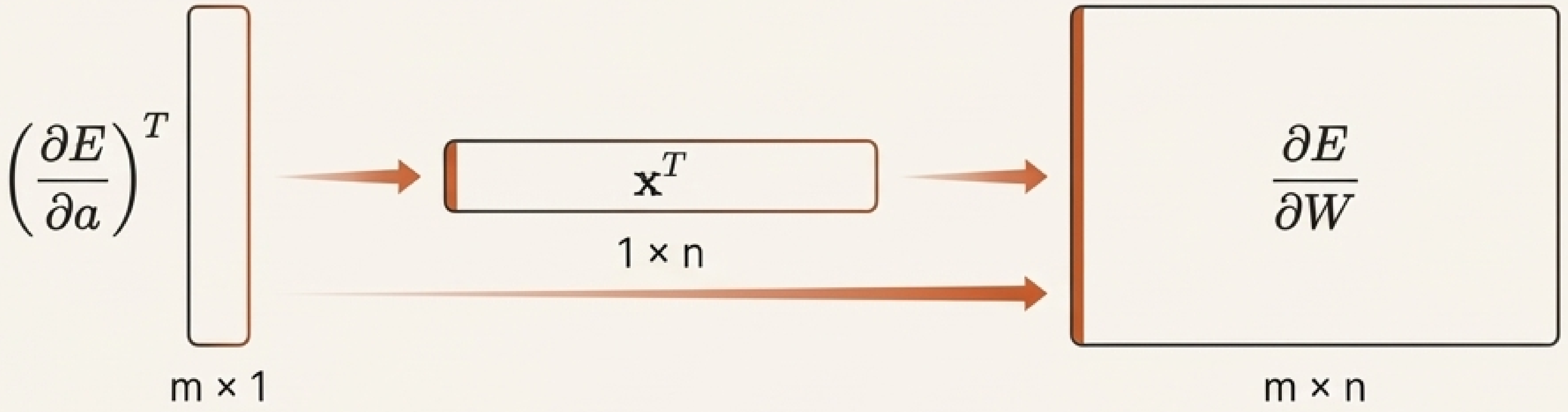
Global Derivative = Product of Local Derivatives:

$$\partial f/\partial x = (\partial d/\partial c) * (\partial c/\partial b) * (\partial b/\partial a) * (\partial a/\partial x)$$

NotebookLM

Backpropagation in Practice: Vectorized Gradients

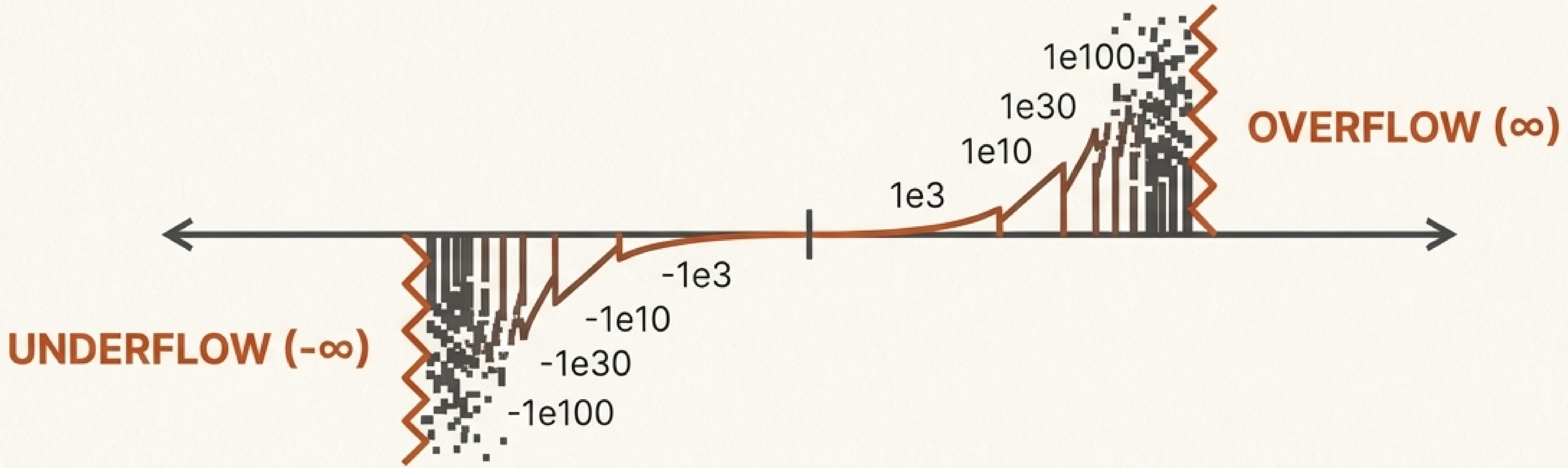
In practice, we don't multiply scalars one by one. Modern libraries implement the backward pass using highly optimized matrix and vector operations. For an affine layer ($a = W\mathbf{x} + b$), the entire gradient matrix $\frac{\partial E}{\partial W}$ can be computed with a single **outer product.**


$\left(\frac{\partial E}{\partial a}\right)^T$
m × 1
$\mathbf{x}^T$
1 × n
$\frac{\partial E}{\partial W}$
m × n

$$\frac{\partial E}{\partial W} = \left(\frac{\partial E}{\partial a}\right)^T * \mathbf{x}^T$$

This vectorization is why GPUs are so effective for training deep neural networks.

The Final Polish: Surviving in the Wild

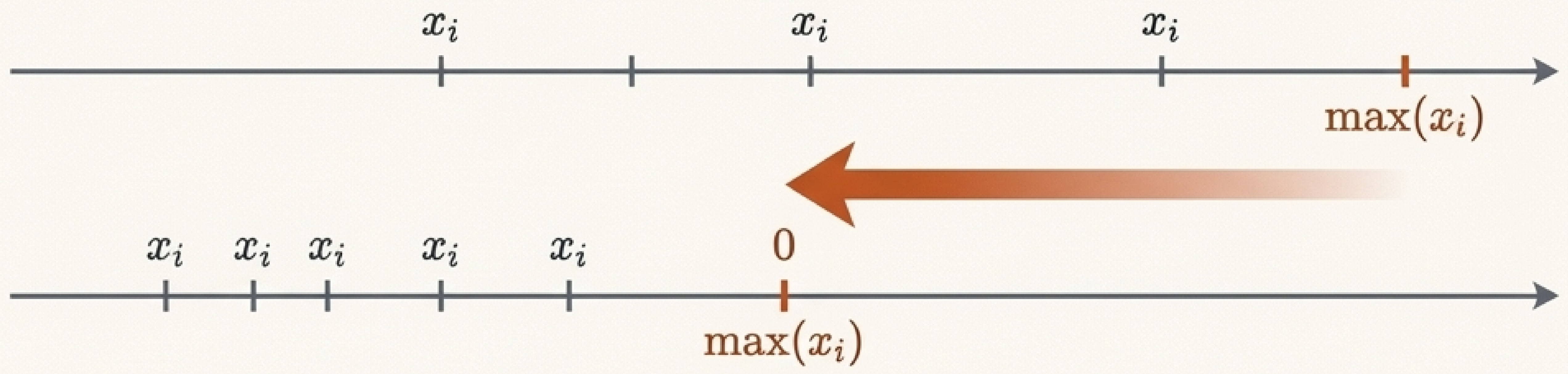


An elegant algorithm in theory can fail in practice. Computers have finite precision, which can lead to numerical overflow (∞) or underflow (-∞) when calculating quantities like the softmax function. This can bring training to a halt.

The Log-Sum-Exp Trick: A Lesson in Stability

To prevent these errors, we use numerical stabilization techniques. By shifting the values inside the exponential ($e^{x_i - a}$), we can ensure the largest value is 0, avoiding overflow without changing the final result. This is the log-sum-exp trick.

$$\log \sum e^{x_i} = a + \log \sum e^{x_i - a}, \text{ where } a = \max(x_i)$$



True mastery of deep learning lies in understanding both the grand theoretical concepts and the crucial practical details that make them work.