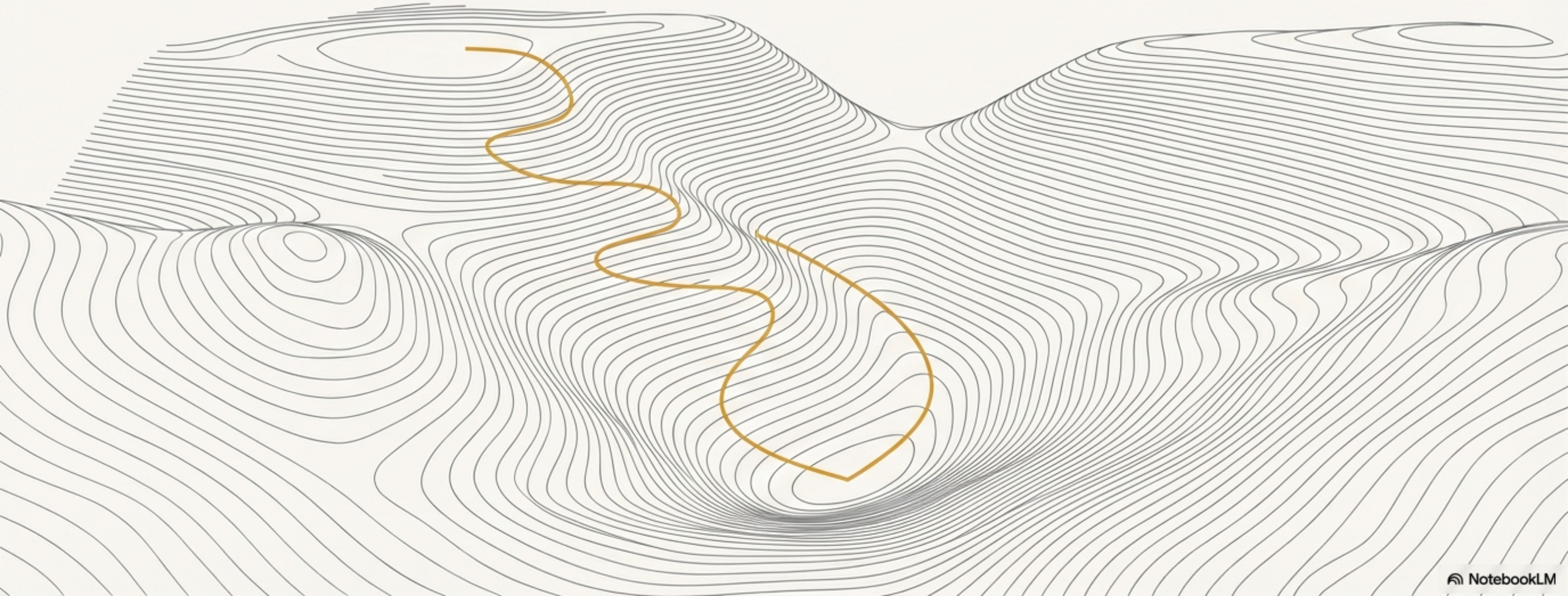
The Quest for Learning

How Neural Networks Learn from Mistakes with Backpropagation



Our Goal: Find the Best Parameters by Minimizing Error

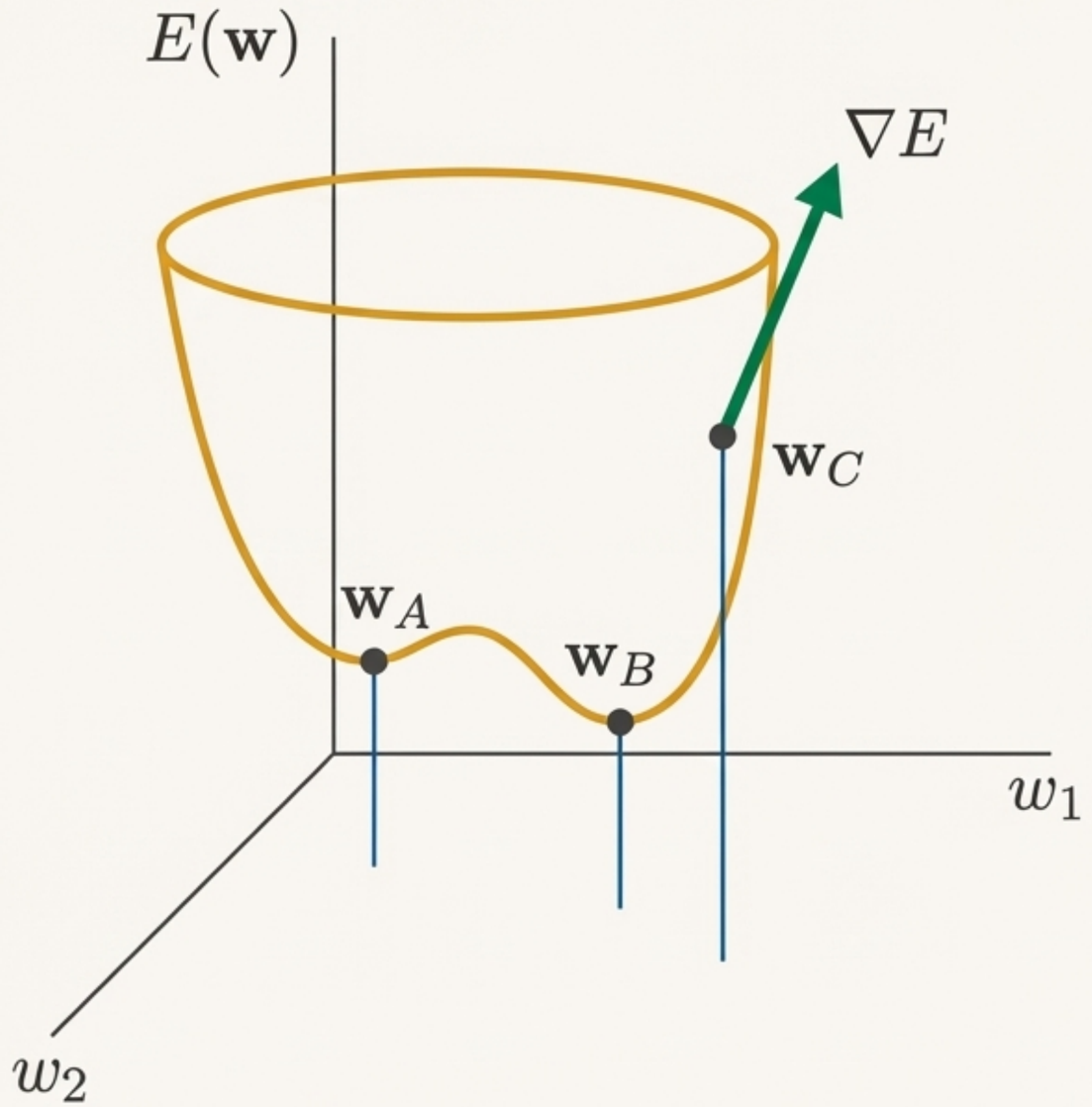
We have a neural network with parameters $\mathbf{W}$. We also have a loss function, $E(\mathbf{W})$, which tells us how "wrong" the network's predictions are. Our goal is to find the set of weights W that makes this error as small as possible.

The Strategy: Gradient Descent

We start with random parameters and take small steps "downhill" on the error surface until we find a local minimum. The update rule is simple:

$$\mathbf{W}(\text{new}) = \mathbf{W}(\text{old}) - \tau \nabla_W E(\mathbf{W}(\text{old}))$$

- τ is the learning rate (step size).
- $\nabla_W E(\mathbf{W})$ is the gradient—a vector that points in the direction of the steepest ascent. We move in the *opposite* direction to minimize the error.

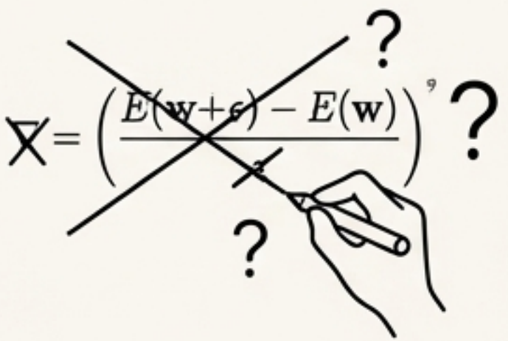


The Core Challenge: How Do We Compute the Gradient $\nabla \mathbf{W}\ E$?

For a deep neural network, **W** can contain millions of parameters. Calculating the gradient efficiently is critical. Let's look at the options:

1. Manual Calculation

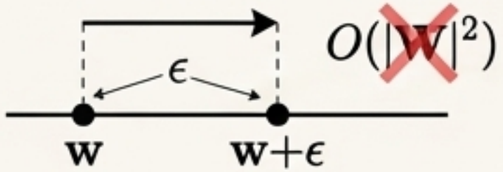
Manually deriving the gradient for every parameter is extremely complex, error-prone, and not scalable.



2. Numeric Differentiation

$$\frac{E(\mathbf{w}+\epsilon) - E(\mathbf{w})}{\epsilon}$$

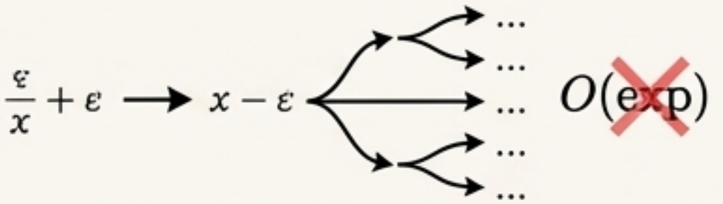
Problem: Very slow. Requires $O(|\mathbf{W}|^2)$ operations because we have to re-evaluate the entire network for each parameter.



3. Symbolic Differentiation

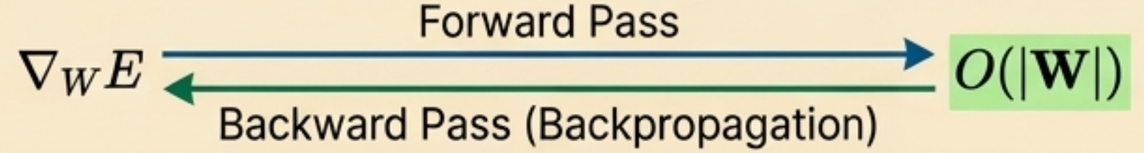
Let a program automatically derive the full mathematical expression for the gradient.

Problem: The resulting expression can be exponentially large and doesn't reuse shared intermediate calculations.



The Solution: Automatic Differentiation

We need a method that is both exact and efficient. This is where **Backpropagation** comes in. It computes the exact gradient for all parameters in just O(|**W**|) time.



The Big Idea: Any Function Can Be Seen as a Graph of Simple Steps

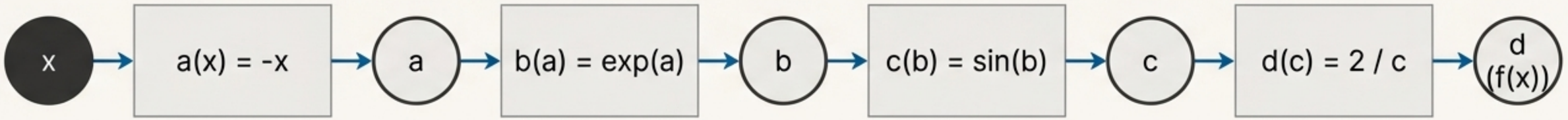
Backpropagation's magic starts by decomposing a complex function into a sequence of simple, modular operations. We can visualize this as a **computational graph**.

A Toy Example

Let's analyze the function $f(x) = 2 \,/\, \sin(\exp(-x))$. We can break it down into four simple modules:

- $a(x) = -x$
- $b(a) = \exp(a)$
- $c(b) = \sin(b)$
- $d(c) = 2 \,/\, c$

This sequence of operations forms a simple, linear graph where the output of one module becomes the input to the next.



The Engine of Backpropagation: The Chain Rule

To find how the final output f changes with respect to the initial input x (the global derivative), we use the chain rule. It tells us to simply multiply the derivatives of each module along the path (local derivatives).

Applying the Chain Rule

The derivative of our function $f(x)$ is the product of the local derivatives of each module in the graph:

$$\frac{\partial f}{\partial x} = \frac{\partial d}{\partial c} * \frac{\partial c}{\partial b} * \frac{\partial b}{\partial a} * \frac{\partial a}{\partial x}$$

- **Global Derivative** ($\partial f/\partial x$): The overall sensitivity of the output to the input.
- **Local Derivatives** ($\partial d/\partial c$, etc.): The sensitivity of each module's output to its immediate input.

The Local Derivatives (Symbolically)

We can easily work out the derivative for each simple module:

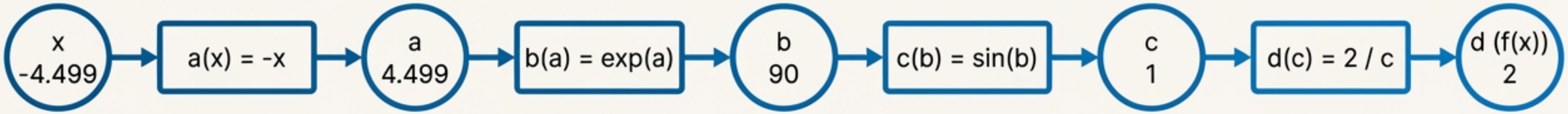
- $a(x) = -x \quad \rightarrow \quad \frac{\partial a}{\partial x} = -1$
- $b(a) = \exp(a) \quad \rightarrow \quad \frac{\partial b}{\partial a} = \exp(a)$
- $c(b) = \sin(b) \quad \rightarrow \quad \frac{\partial c}{\partial b} = \cos(b)$
- $d(c) = \frac{2}{c} \quad \rightarrow \quad \frac{\partial d}{\partial c} = -\frac{2}{c^2}$

Step 1: The Forward Pass – Calculate and Cache

First, we push an input value through the graph from start to finish to compute the final output. As we do this, we **cache (save) the intermediate values** at each step. We'll need them for the backward pass.

Example Calculation at $x = -4.499$

1. **Input:** $x = -4.499$
2. $a = -x \rightarrow a = 4.499$
3. $b = \exp(a) \rightarrow b = \exp(4.499) = 90$
4. $c = \sin(b) \rightarrow c = \sin(90) = 1$ (approx.)
5. $d = 2 \,/\, c \rightarrow d = 2 \,/\, 1 = 2$
6. **Final Output:** $f(-4.499) = 2$



Step 2: The Backward Pass – Compute and Multiply

Now, we move backward through the graph. We use our saved values from the forward pass to compute the numerical value of each local derivative.

Computing Local Derivatives

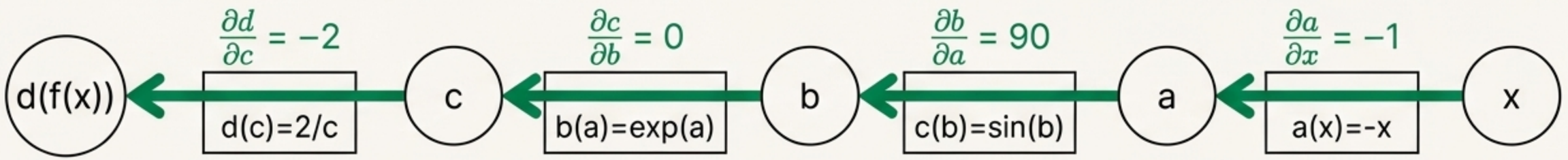
Using the cached values (a=4.499, b=90, c=1):

- $\partial d/\partial c = -2/c^2 \rightarrow -2/1^2 = -2$
- $\partial c/\partial b = \cos(b) \rightarrow \cos(90) = 0$ (approx.)
- $\partial b/\partial a = \exp(a) \rightarrow \exp(4.499) = 90$
- $\partial a/\partial x = -1$

Obtaining the Global Derivative

Finally, we multiply these local derivatives together as dictated by the chain rule:

$$\frac{\partial f}{\partial x} = \frac{\partial d}{\partial c} * \frac{\partial c}{\partial b} * \frac{\partial b}{\partial a} * \frac{\partial a}{\partial x} \quad \text{and} \quad \frac{\partial f}{\partial x} = (-2) * (0) * (90) * (-1) = 0$$



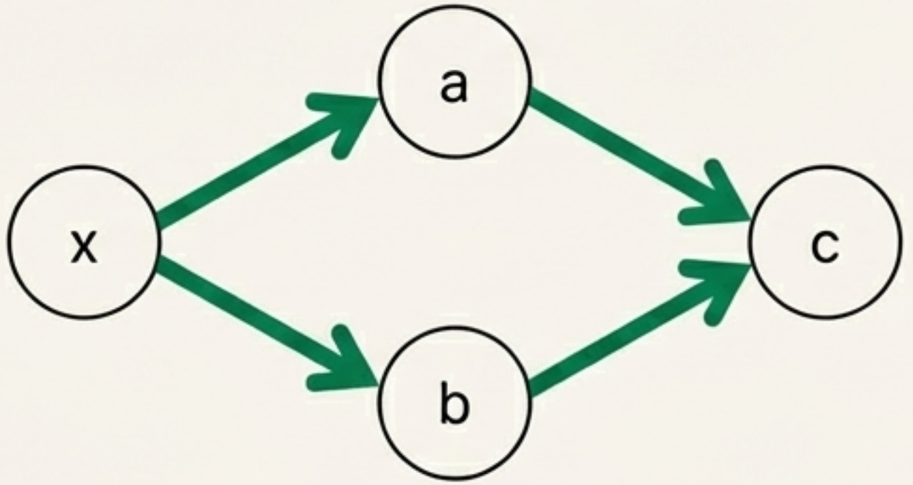
Generalizing the Chain Rule for Complex Graphs

Real neural networks are not simple chains. A single variable can influence the output through multiple paths.

The Multivariate Chain Rule

If a variable x affects an output c through multiple intermediate variables a_i, we must calculate the derivative along each path and **sum them up**.

$$\frac{\partial c}{\partial x} = \sum_{i=1}^{m} \frac{\partial c}{\partial a_i} \frac{\partial a_i}{\partial x}.$$



In Plain English

The total influence of x on c is the sum of its influences through all possible paths.

$$\frac{\partial c}{\partial x} = \frac{\partial c}{\partial a} \frac{\partial a}{\partial x} + \frac{\partial c}{\partial b} \frac{\partial b}{\partial x}.$$

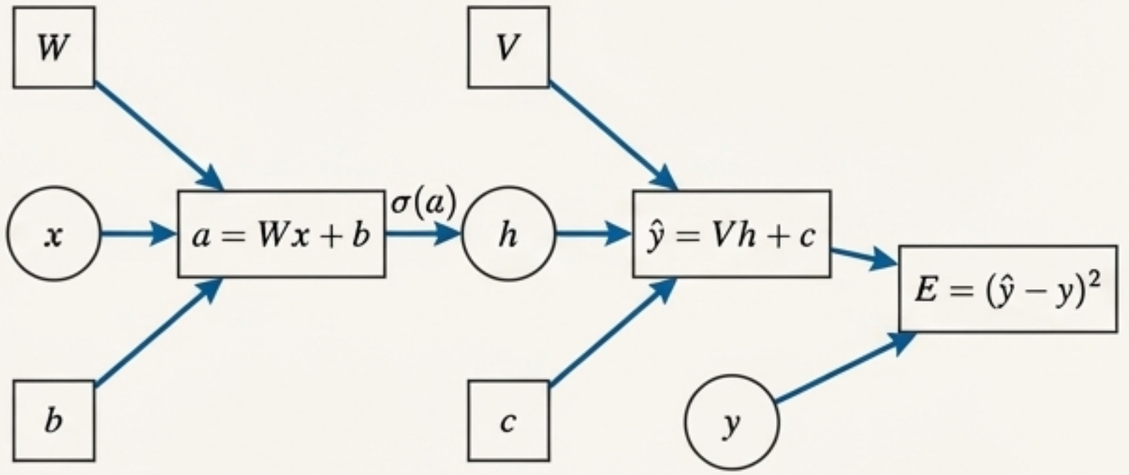
From a Simple Function to a Neural Network

The same principles apply to a neural network. Let's consider a simple network for regression with one hidden layer.

The Network as a Computational Graph

- Input: $x \in \mathbb{R}^D$
- Layer 1 (Affine): $a = Wx + b$
- Activation: $h = \sigma(a)$
- Layer 2 (Output): $\hat{y} = Vh + c$
- Loss: $E = (\hat{y} - y)^2$

Our goal is to find the gradient of the Error E with respect to the weights, e.g., $\frac{\partial E}{\partial W}$.



The New Challenge

The chain rule still applies: $\frac{\partial E}{\partial W} = \frac{\partial E}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial h} \frac{\partial h}{\partial a} \frac{\partial a}{\partial W}$. But what does it mean to take a derivative of a vector **a** with respect to a matrix **W**? We need to move from simple calculus to matrix calculus.

The Language of Vector Derivatives: The Jacobian

When dealing with vector functions, the derivative is generalized to a matrix of all possible partial derivatives, called the **Jacobian**.

For a function $a = f(x)$ where $x \in \mathbb{R}^n$ and $a \in \mathbb{R}^m$, the Jacobian $\frac{\partial a}{\partial x}$ is an $m \times n$ matrix:

$$\left[\frac{\partial a}{\partial x}\right]_{ij} = \frac{\partial a_i}{\partial x_j}$$

The Chain Rule in Matrix Form

The multivariate chain rule can be written compactly using Jacobians. If $c = g(a)$ and $a = f(x)$:

Jacobian form: $\frac{\partial c}{\partial x} = \frac{\partial c}{\partial a} * \frac{\partial a}{\partial x}$

(where the dimensions are (1 × n) = (1 × m) * (m × n))

Gradient form: $\nabla_x c = \left(\frac{\partial a}{\partial x}\right)^T * \nabla_a c$

(where the dimensions are (n × 1) = (n × m) * (m × 1))

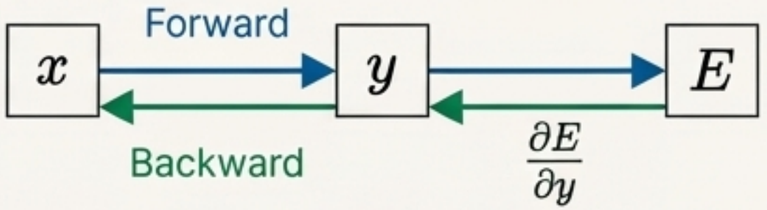
This gives us a formal way to handle the chain rule for entire layers of a neural network at once.

The Secret to Efficiency: The Jacobian-Vector Product

The Jacobians (e.g., ∂a/∂W) can be huge multidimensional tensors. Computing and storing them explicitly would be incredibly expensive.

The Key Insight

We never actually need the Jacobian matrix itself! We only need the product of the upstream gradient with the local Jacobian.



$$\frac{\partial E}{\partial x} = \frac{\partial E}{\partial y} * \frac{\partial y}{\partial x}$$

This Jacobian-vector product can often be calculated much more efficiently than first building the full Jacobian *∂y/∂x* and then multiplying.

The forward/backward API

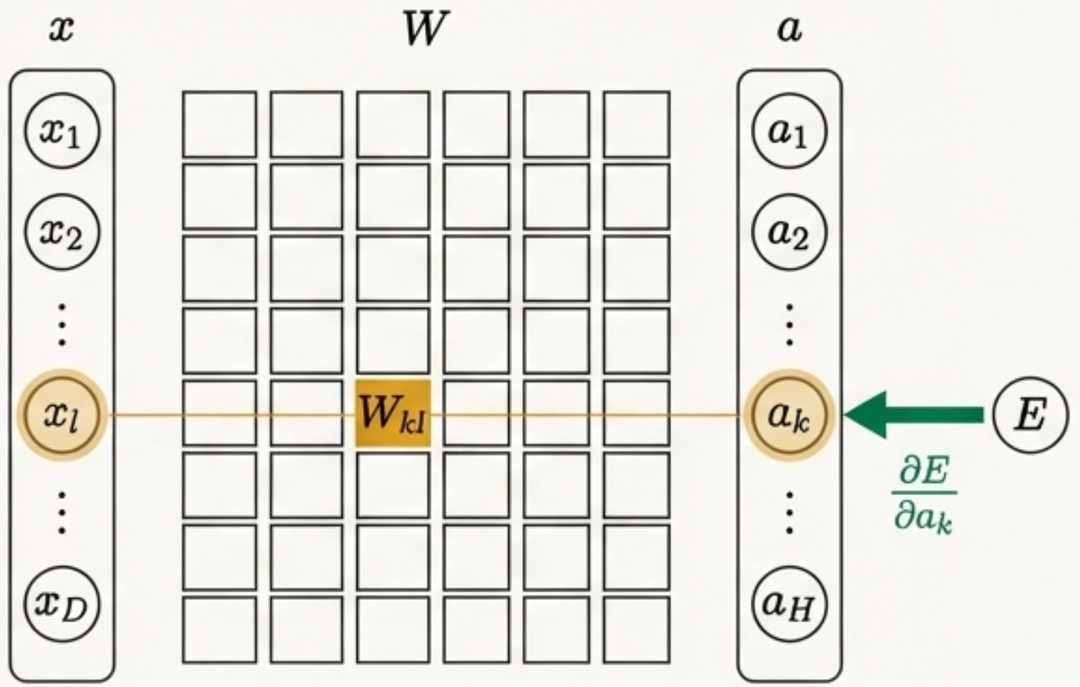
Deep learning frameworks implement this insight. Each layer defines two functions:

- `forward(input)`: Computes the output and caches necessary values.
- `backward(upstream_gradient)`: Takes the upstream gradient *∂E/∂y* and efficiently computes and returns *∂E/∂x* to be passed to the next layer.

Deep Dive: The Backward Pass of an Affine Layer

Let's see how this works for an affine layer: $a = Wx + b$

- Inputs: $x \in \mathbb{R}^D, W \in \mathbb{R}^{H \times D}, b \in \mathbb{R}^H$
- Output: $a \in \mathbb{R}^H$



Our Task

Given the upstream gradient $\frac{\partial E}{\partial a}$ (a vector of size H), we need to compute the gradients with respect to the parameters ($\frac{\partial E}{\partial W}$, $\frac{\partial E}{\partial b}$) and the input ($\frac{\partial E}{\partial x}$).

Step 1: Focus on a Single Weight, W_{kl}

To build intuition, let's find the gradient for just one element of the weight matrix. W_{kl} only influences the k-th output, a_k. The multivariate chain rule simplifies:

$$\frac{\partial E}{\partial W_{kl}} = \sum_{i=1}^{H} \frac{\partial E}{\partial a_i} \frac{\partial a_i}{\partial W_{kl}}.$$

Building the Gradient One Weight at a Time

Let’s trace the influence of W_kl on E.

$$\frac{\partial E}{\partial W_{kl}} = \sum_{i=1}^{H} \frac{\partial E}{\partial a_i} \frac{\partial (Wx + b)_i}{\partial W_{kl}}$$

The term $(Wx+b)_i$ is $\sum_{j=1}^{D} W_{ij} * x_j$. This derivative is zero unless i=k and j=l.

- When i=k, the derivative $\frac{\partial (\sum_j W_{kj} \cdot x_j)}{\partial W_{kl}}$ is simply x_l.
- For all i ≠ k, the derivative is 0.

So, the sum collapses to a single term:

$$\frac{\partial E}{\partial W_{kl}} = \frac{\partial E}{\partial a_k} x_l$$

The Intuitive Result

The gradient for a single weight W_{kl} is the upstream gradient for its corresponding output neuron a_k multiplied by its corresponding input value x_l. This makes perfect sense: the “blame” assigned to the weight depends on the error at its output and the strength of its input.

Scaling Up: From One Weight to the Entire Matrix

We found that for each element (k, l) of the weight matrix:

$$\frac{\partial E}{\partial W_{kl}} = \frac{\partial E}{\partial a_k} * x_l$$

This pattern is exactly the definition of an **outer product** between two vectors. We can compute the entire gradient matrix $\frac{\partial E}{\partial W}$ with a single, highly efficient matrix operation.

The backward Pass in Matrix Form

Given the upstream gradient $\frac{\partial E}{\partial a}$:

- **Gradient for Weights W**:

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

- **Gradient for Input x**: (A column vector times a row vector gives a matrix)

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

- **Gradient for Bias b**:

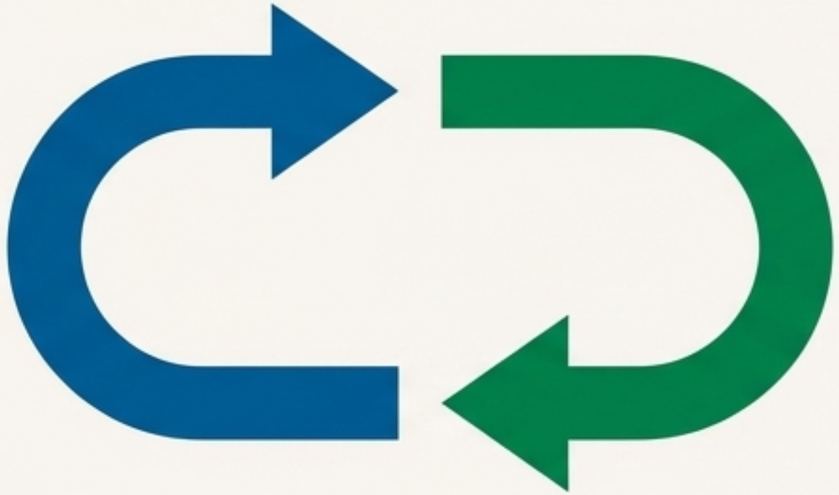
$$\frac{\partial E}{\partial b} = \frac{\partial E}{\partial a}$$

These compact equations are the heart of the backward pass for an affine layer, implemented efficiently in all modern deep learning libraries.

Backpropagation: The Complete Picture

Backpropagation is an elegant and efficient algorithm for computing gradients in a neural network. It's the engine that powers modern deep learning.

Forward Pass: Compute Outputs & Cache Values**



Backward Pass: Compute Gradients via Chain Rule**

The Process in a Nutshell

1. **Define the Graph**: Represent the network's computation as a composition of modules (layers).
2. **Forward Pass**: Run data through the graph from input to output, computing the final loss and caching intermediate values along the way.
3. **Backward Pass**: Start from the final loss and walk backward through the graph. At each module, use the chain rule to compute the gradient with respect to its inputs, using the cached values from the forward pass.
4. **Efficiency**: This process is fast because it never materializes the full Jacobian matrices, instead relying on efficient Jacobian-vector products to pass gradients from one layer to the next.