IN402

Machine Learning

Chapter

9

Backpropagation from First Principles

Author: Abbas El-Hajj Youssef

University: Lebanese University

Department: Department of Computer Science

These notes extend course materials taught by Prof. Ahmad Faour with additional content from textbooks and supplementary resources.

Disclaimer: *This is not an official course document.*

Contents

1 The Problem of Gradients in Deep Computation

Machine learning, at its computational core, is an exercise in optimization. We have a loss function $\mathcal{L}$ that measures how poorly our model performs, and we seek parameters $\boldsymbol{\theta}$ that minimize this loss. The workhorse of modern optimization is gradient descent: compute the direction of steepest increase in $\mathcal{L}$, then step in the opposite direction.

This strategy is elegantly simple when stated abstractly. But it conceals a computational question that, for deep neural networks, is far from trivial: **how do we actually compute the gradient $\nabla_\theta \mathcal{L}$?**

1.1 When Differentiation Is Easy

Consider a function given by a simple formula:

$$f(\theta) = \theta^2 + 3\theta + 7$$

The derivative is immediate: $f'(\theta) = 2\theta + 3$. We apply familiar rules—power rule, sum rule, constant rule—and the answer emerges.

Now consider a slightly more structured computation:

$$u = \theta^2 \tag{1}$$

$$v = 3\theta \tag{2}$$

$$f = u + v + 7 \tag{3}$$

This is the same function, but expressed as a *sequence of operations*. The chain rule tells us how to differentiate through this sequence:

$$\frac{df}{d\theta} = \frac{\partial f}{\partial u} \cdot \frac{du}{d\theta} + \frac{\partial f}{\partial v} \cdot \frac{dv}{d\theta} = 1 \cdot 2\theta + 1 \cdot 3 = 2\theta + 3$$

So far, no difficulty. The chain rule decomposes the problem into manageable pieces.

1.2 When Differentiation Becomes Hard

Now consider what happens in a neural network. The loss $\mathcal{L}$ depends on the network's prediction $\hat{y}$, which depends on the final layer's activations, which depend on the previous layer's activations, and so on through dozens or hundreds of layers, ultimately depending on millions of parameters.

Consider a network with L layers, where layer ℓ has $n^{[\ell]}$ neurons. A single weight w in the first layer influences all $n^{[1]}$ outputs of that layer. Each of those influences all $n^{[2]}$ outputs of layer 2. The cascade continues: the influence of w on the final loss travels through every possible path from w to the output.

The total derivative can be written as a sum over all such paths:

$$\frac{\partial \mathcal{L}}{\partial w} = \sum_{\text{paths } p: w \to \mathcal{L}} \prod_{(u \to v) \in p} \frac{\partial v}{\partial u} \tag{4}$$

Each path contributes a product of local derivatives along its edges. The number of paths equals the product of layer widths:

$$\text{Number of paths} = \prod_{\ell=1}^{L} n^{[\ell]}$$

Combinatorial Explosion

For a modest network with 5 layers of 100 neurons each, this gives $100^5 = 10^{10}$ paths—ten billion distinct routes from one parameter to the loss.

Yet modern neural networks train successfully, computing gradients for billions of parameters in seconds. How is this possible?

1.3 The Key Insight: Shared Structure

The combinatorial explosion arises from treating each path as independent. But the paths are not independent—they *share structure*. Many paths pass through the same intermediate nodes, and the contribution of those shared segments can be computed once rather than repeatedly.

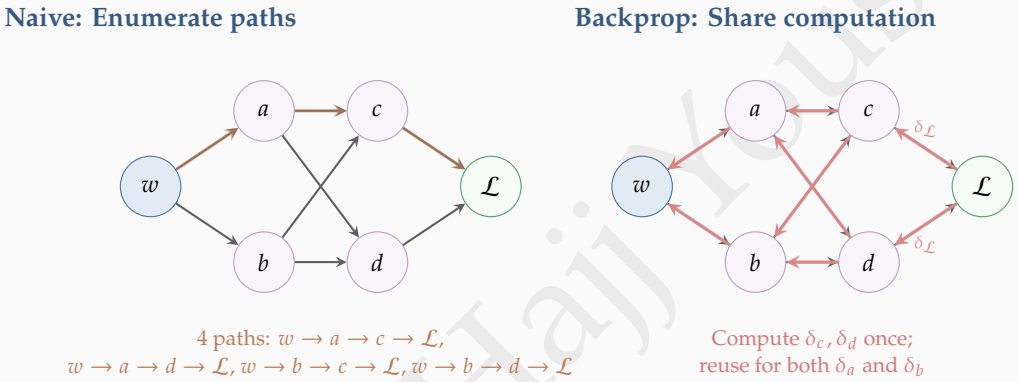

Figure 1: *Path enumeration versus shared computation.* ***Left****: Naively enumerating all paths leads to redundant computation—the segment $c \to ℒ$ is traversed twice.* ***Right****: Backpropagation computes each error signal once and reuses it, transforming exponential work into linear work.*

This insight—that shared computation can be factored out—transforms exponential work into linear work. The sum-of-products in equation (4) can be reorganized into a product-of-sums, computed efficiently by working backward through the network.

The algorithm that exploits this structure is called **backpropagation**. As we shall see, it is not merely one efficient algorithm among alternatives. Given the constraints of the problem, it is the *unique* efficient algorithm—the one that anyone reasoning carefully about gradient computation must inevitably discover.[1]

2 Computation as a Graph

The first step toward efficient gradient computation is to represent the computation itself in a form that reveals its structure. Rather than viewing the loss as a monolithic formula, we decompose it into a network of simple operations, each depending on a small number of inputs.

[1] Backpropagation was discovered independently by Bryson & Ho (1969), Werbos (1974), and Rumelhart, Hinton & Williams (1986). This repeated discovery reflects inevitability: anyone confronting efficient gradient computation in composite systems must arrive at the same answer.

2.1 The Running Example: Network Architecture

We consider the task of fitting a nonlinear function to data. Our network takes a single input x and produces a prediction $\hat{y}$:

- **Input**: A scalar x
- **Hidden layer**: Two units with sigmoid activation, $\sigma(z) = 1/(1 + e^{-z})$
- **Output**: A linear combination of hidden activations

The forward computation proceeds as follows:

$$z_1 = w_1 x + b_1 \qquad \text{(pre-activation, hidden unit 1)} \tag{5}$$

$$z_2 = w_2 x + b_2 \qquad \text{(pre-activation, hidden unit 2)} \tag{6}$$

$$a_1 = \sigma(z_1) \qquad \text{(activation, hidden unit 1)} \tag{7}$$

$$a_2 = \sigma(z_2) \qquad \text{(activation, hidden unit 2)} \tag{8}$$

$$\hat{y} = v_1 a_1 + v_2 a_2 + c \qquad \text{(network output)} \tag{9}$$

For a dataset of m training examples $\{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^{m}$, we measure performance using mean squared error:

$$\mathcal{L} = \frac{1}{2m} \sum_{i=1}^{m} \left(\hat{y}^{(i)} - y^{(i)}\right)^2 \tag{10}$$

This network has seven learnable parameters: w_1, w_2 (input weights), b_1, b_2 (hidden biases), v_1, v_2 (output weights), and c (output bias).

2.2 The Computational Graph

The equations above define a sequence of computations that we visualize as a directed graph. Each node represents a quantity; each edge represents a direct dependency.

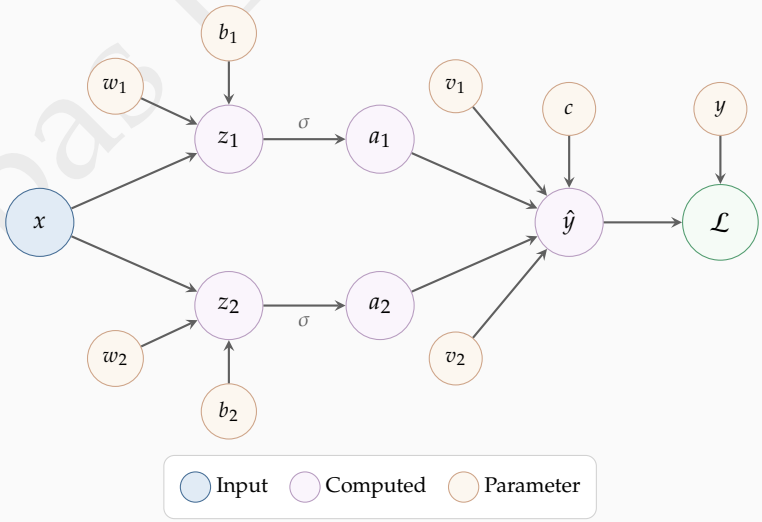

Figure 2: *Computational graph for the neural network. Information flows left to right. Each arrow indicates that the destination node depends on the source node. The loss $\mathcal{L}$ ultimately depends on all parameters through chains of intermediate computations.*

This graph makes explicit every dependency in the computation. The key observation is **shared downstream structure**: parameters w_1 and b_1 both feed into z_1, sharing the entire

downstream path $z_1 \to a_1 \to \hat{y} \to \mathcal{L}$. Similarly, the input x fans out to both z_1 and z_2, but all paths eventually merge at $\hat{y}$. This sharing—where many upstream nodes influence common downstream nodes—is the foundation of efficient gradient computation.

2.3 Formal Definition

Computational Graph

A **computational graph** is a directed acyclic graph (DAG) $G = (V, E)$ where:

- Each node $v \in V$ represents an *input*, a *parameter*, or a *computed value*.
- Each directed edge $(u, v) \in E$ indicates that u is an *immediate dependency* of v.
- Each computed node has an associated *local operation* determining its value from its parents.

The acyclic structure guarantees the existence of a *topological ordering*—a sequence in which every node appears after all its parents.

2.4 Paths and the Gradient

From the graph, we can trace every path from a parameter to the loss. Consider w_1:

$$w_1 \to z_1 \to a_1 \to \hat{y} \to \mathcal{L}$$

By the chain rule, the gradient along this path is the product of local derivatives:

$$\frac{\partial \mathcal{L}}{\partial w_1} = \frac{\partial z_1}{\partial w_1} \cdot \frac{\partial a_1}{\partial z_1} \cdot \frac{\partial \hat{y}}{\partial a_1} \cdot \frac{\partial \mathcal{L}}{\partial \hat{y}}$$

Now consider the input x, which has *two* paths to $\mathcal{L}$:

$$\begin{aligned} \text{Path 1:} \quad & x \to z_1 \to a_1 \to \hat{y} \to \mathcal{L} \\ \text{Path 2:} \quad & x \to z_2 \to a_2 \to \hat{y} \to \mathcal{L} \end{aligned}$$

The total gradient is the *sum* of contributions from both paths:

$$\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial z_1}{\partial x} \cdot \frac{\partial a_1}{\partial z_1} \cdot \frac{\partial \hat{y}}{\partial a_1} \cdot \frac{\partial \mathcal{L}}{\partial \hat{y}} + \frac{\partial z_2}{\partial x} \cdot \frac{\partial a_2}{\partial z_2} \cdot \frac{\partial \hat{y}}{\partial a_2} \cdot \frac{\partial \mathcal{L}}{\partial \hat{y}}$$

This is equation (4) made concrete. Backpropagation computes the gradient using work proportional to edges, not paths—a transformation we now develop.

3 Sensitivity: Local and Global

We now develop the conceptual foundation for gradient computation. The key insight is to distinguish two fundamentally different questions about how quantities respond to perturbations.

3.1 Two Types of Sensitivity

Consider any node v in our computational graph. We can ask two distinct questions:

1. **Local**: If I perturb an input to node v, how does the output of v change?
2. **Global**: If I perturb the value at node v, how does the final loss $\mathcal{L}$ change?

These questions have fundamentally different characters:

- The **local question** depends only on what happens *at* node v—the operation it performs and the current values of its inputs.
- The **global question** depends on everything *downstream* of v—all the ways that v's output eventually influences $\mathcal{L}$.

Local and Global Sensitivity

For any node v in a computational graph with loss $\mathcal{L}$:

1. The **local sensitivity** of v with respect to parent u is $\partial v/\partial u$—how much v changes per unit change in u, holding other inputs fixed.
2. The **global sensitivity** of $\mathcal{L}$ with respect to v, denoted $\delta_v = \partial \mathcal{L}/\partial v$, is how much $\mathcal{L}$ changes per unit change in v. We call δ_v the **error signal** at node v.

Local sensitivities are easy to compute—they involve only a single operation. Global sensitivities are what we ultimately need, but they seem to require knowledge of the entire downstream computation.
The remarkable fact: global sensitivities can be computed from local sensitivities, working backward through the graph.

3.2 The Chain Rule for a Single Path

Consider a simple chain: $u \to v \to \mathcal{L}$.

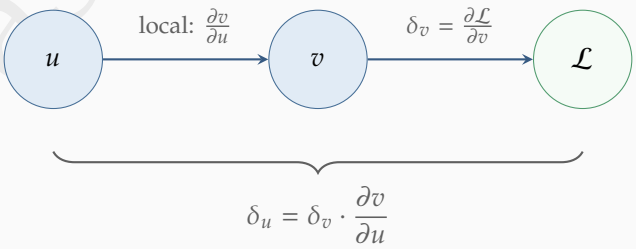

Figure 3: *The chain rule for a linear path. The error signal at u equals the error signal at v times the local derivative.*

The chain rule states:

$$\delta_u = \frac{\partial \mathcal{L}}{\partial u} = \frac{\partial \mathcal{L}}{\partial v} \cdot \frac{\partial v}{\partial u} = \delta_v \cdot \frac{\partial v}{\partial u} \tag{11}$$

Read operationally: *the error signal at a parent equals the error signal at the child times the local derivative between them.* This means error signals propagate backward: knowing δ_v lets us compute δ_u using only local information.

3.3 The Chain Rule for Multiple Paths

When a node influences the loss through multiple paths, the total effect is the sum of effects along each path.

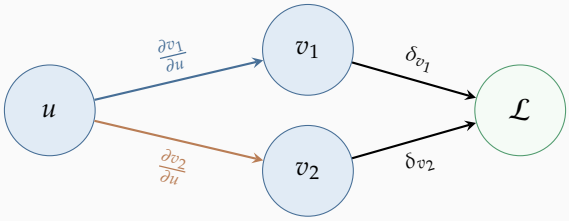

Figure 4: *When u has multiple children, its error signal is the sum of contributions from each child.*

Mathematically, if u has children $v_1, \ldots, v_k$:

$$\delta_u = \frac{\partial \mathcal{L}}{\partial u} = \sum_{j=1}^{k} \frac{\partial \mathcal{L}}{\partial v_j} \cdot \frac{\partial v_j}{\partial u} = \sum_{j=1}^{k} \delta_{v_j} \cdot \frac{\partial v_j}{\partial u} \tag{12}$$

This is the backpropagation formula: each parent's error signal is the sum of (child error signal) $\times$ (local derivative) over all children. Each child reports how much it cares about changes in the parent; the parent's total responsibility is the sum of these reports.

3.4 Local Derivatives for Our Network

Each local derivative is determined by elementary calculus. For our network:

Node	Operation	Local Derivatives
z_k	$z_k = w_k x + b_k$	$\frac{\partial z_k}{\partial w_k} = x, \quad \frac{\partial z_k}{\partial b_k} = 1, \quad \frac{\partial z_k}{\partial x} = w_k$
a_k	$a_k = \sigma(z_k)$	$\frac{\partial a_k}{\partial z_k} = \sigma(z_k)(1-\sigma(z_k)) = a_k(1-a_k)$[2]
$\hat{y}$	$\hat{y} = v_1 a_1 + v_2 a_2 + c$	$\frac{\partial \hat{y}}{\partial a_k} = v_k, \quad \frac{\partial \hat{y}}{\partial v_k} = a_k, \quad \frac{\partial \hat{y}}{\partial c} = 1$
$\mathcal{L}$	$\mathcal{L} = \frac{1}{2m}\sum_i (\hat{y}^{(i)} - y^{(i)})^2$	$\frac{\partial \mathcal{L}}{\partial \hat{y}^{(i)}} = \frac{1}{m}(\hat{y}^{(i)} - y^{(i)})$

Note the pattern for linear operations: the derivative with respect to a weight equals the value it multiplies.

3.5 The Backward Pass

We now have all ingredients to compute gradients. Starting from $\delta_{\mathcal{L}} = 1$, we propagate error signals backward.

[2] The sigmoid derivative $\sigma'(z) = \sigma(z)(1-\sigma(z))$ can be verified directly: $\frac{d\sigma}{dz} = \frac{d}{dz}\left(\frac{1}{1+e^{-z}}\right) = \frac{e^{-z}}{(1+e^{-z})^2} = \sigma(z)(1-\sigma(z))$. The derivative is maximal at $z = 0$ where $\sigma'(0) = 0.25$, and approaches zero as $|z| \to \infty$.

Backward Pass Protocol

1. **Initialize**: $\delta_{\mathcal{L}} = 1$
2. **Output layer**: For each training example i:

$$\delta_{\hat{y}}^{(i)} = \delta_{\mathcal{L}} \cdot \frac{\partial \mathcal{L}}{\partial \hat{y}^{(i)}} = 1 \cdot \frac{1}{m}(\hat{y}^{(i)} - y^{(i)}) = \frac{1}{m}(\hat{y}^{(i)} - y^{(i)})$$

3. **Hidden activations**:

$$\delta_{a_k}^{(i)} = \delta_{\hat{y}}^{(i)} \cdot \frac{\partial \hat{y}}{\partial a_k} = \delta_{\hat{y}}^{(i)} \cdot v_k$$

4. **Pre-activations**:

$$\delta_{z_k}^{(i)} = \delta_{a_k}^{(i)} \cdot \frac{\partial a_k}{\partial z_k} = \delta_{a_k}^{(i)} \cdot a_k^{(i)}(1 - a_k^{(i)})$$

5. **Parameter gradients**: Sum over training examples:

$$\frac{\partial \mathcal{L}}{\partial v_k} = \sum_i \delta_{\hat{y}}^{(i)} \cdot a_k^{(i)} \tag{13}$$

$$\frac{\partial \mathcal{L}}{\partial c} = \sum_i \delta_{\hat{y}}^{(i)} \tag{14}$$

$$\frac{\partial \mathcal{L}}{\partial w_k} = \sum_i \delta_{z_k}^{(i)} \cdot x^{(i)} \tag{15}$$

$$\frac{\partial \mathcal{L}}{\partial b_k} = \sum_i \delta_{z_k}^{(i)} \tag{16}$$

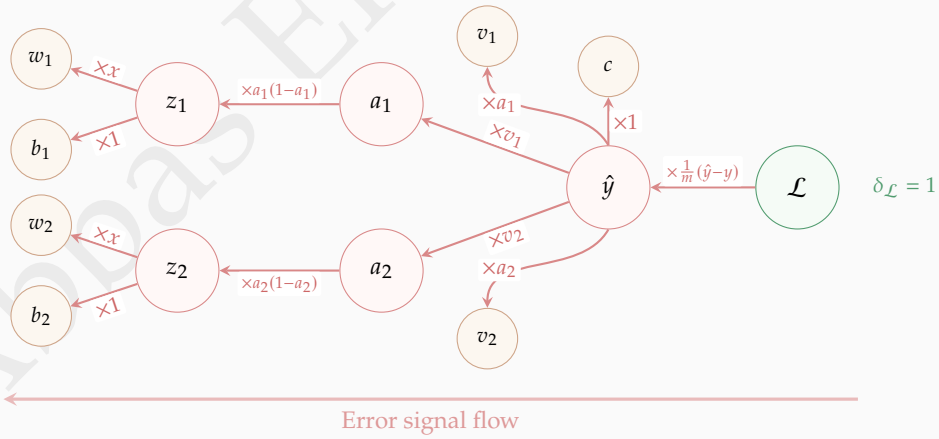

Figure 5: *Error signals flow backward through the network. Each edge shows the local derivative (the multiplier). Starting from $\delta_{\mathcal{L}} = 1$, we multiply along edges to propagate error signals to all nodes.*

3.6 Tracing a Complete Gradient

Let us trace the gradient for w_1 explicitly. The path is $w_1 \to z_1 \to a_1 \to \hat{y} \to \mathcal{L}$.
For a single training example (dropping the superscript for clarity):

$$\delta_{\mathcal{L}} = 1 \tag{17}$$

$$\delta_{\hat{y}} = \delta_{\mathcal{L}} \cdot \frac{1}{m}(\hat{y} - y) = \frac{1}{m}(\hat{y} - y) \tag{18}$$

$$\delta_{a_1} = \delta_{\hat{y}} \cdot v_1 \tag{19}$$

$$\delta_{z_1} = \delta_{a_1} \cdot a_1(1 - a_1) = \delta_{\hat{y}} \cdot v_1 \cdot a_1(1 - a_1) \tag{20}$$

$$\frac{\partial \mathcal{L}}{\partial w_1} = \delta_{z_1} \cdot x = \frac{1}{m}(\hat{y} - y) \cdot v_1 \cdot a_1(1 - a_1) \cdot x \tag{21}$$

For the full gradient, we sum over all training examples:

$$\frac{\partial \mathcal{L}}{\partial w_1} = \sum_{i=1}^{m} \frac{1}{m}(\hat{y}^{(i)} - y^{(i)}) \cdot v_1 \cdot a_1^{(i)}(1 - a_1^{(i)}) \cdot x^{(i)} \tag{22}$$

Shared Computation

Both $\partial \mathcal{L}/\partial w_1$ and $\partial \mathcal{L}/\partial b_1$ require the error signal δ_{z_1}. Backpropagation computes δ_{z_1} once and reuses it:

$$\frac{\partial \mathcal{L}}{\partial w_1} = \sum_i \delta_{z_1}^{(i)} \cdot x^{(i)} \tag{23}$$

$$\frac{\partial \mathcal{L}}{\partial b_1} = \sum_i \delta_{z_1}^{(i)} \cdot 1 \tag{24}$$

This sharing—computing each error signal exactly once—is what makes backpropagation efficient.

4 Why Backpropagation Must Be

We have described how to propagate error signals backward through a computational graph. Now we demonstrate something deeper: this backward propagation is not merely *one* efficient approach—it is the *only* efficient approach. The structure of the problem leaves no alternative.

4.1 The Constraints

Given our constraints—a DAG structure, a single scalar loss, and exponentially many paths—the chain rule must be applied strategically to share computation. The question is not *whether* to factor the computation, but *how*.

4.2 The Factorization Principle

The sum-of-products formula (4) has special structure that permits factorization. Consider a node x that fans out to nodes a and b, which both feed into $\mathcal{L}$. Let $\alpha_1 = \partial a/\partial x$, $\alpha_2 = \partial b/\partial x$, $\beta_1 = \partial \mathcal{L}/\partial a$, and $\beta_2 = \partial \mathcal{L}/\partial b$.
The naive path-sum approach computes:

$$\frac{\partial \mathcal{L}}{\partial x} = \alpha_1\beta_1 + \alpha_2\beta_2$$

This requires 4 multiplications. But we can factor by computing intermediate error signals:

$$\delta_a = \beta_1, \quad \delta_b = \beta_2 \tag{25}$$

$$\frac{\partial \mathcal{L}}{\partial x} = \alpha_1 \cdot \delta_a + \alpha_2 \cdot \delta_b \tag{26}$$

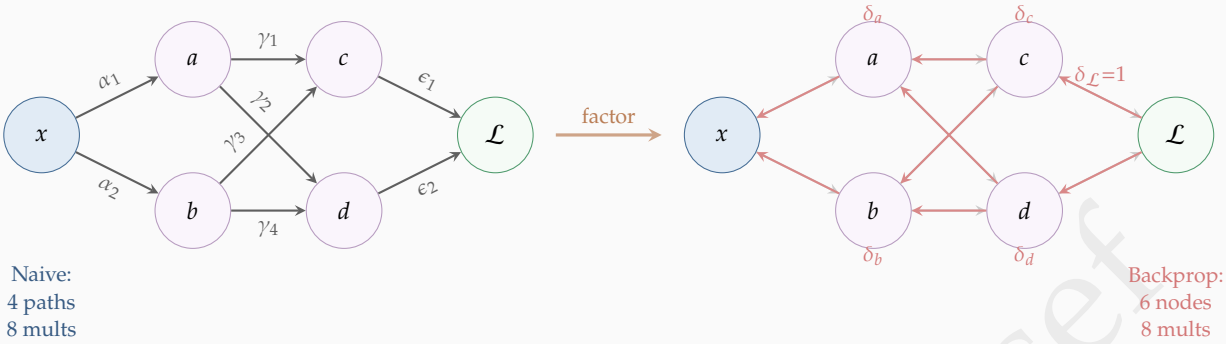

Figure 6: *The factorization principle.* ***Left****: A network where x influences $\mathcal{L}$ through 4 paths.* ***Right****: Backpropagation computes error signals δ_c, δ_d first, then uses them to compute δ_a, δ_b, finally combining to get δ_x. The work is proportional to edges (8 multiplications), not paths.*

For this small example, the savings are minimal. But consider a deeper network where a and b each have multiple paths to $\mathcal{L}$. Computing δ_a and δ_b first—each as a sum over their downstream paths—and then combining with α_1 and α_2 avoids redundant computation of shared downstream structure. This factorization—computing $\delta_u = \sum_v \delta_v \cdot \partial v / \partial u$—transforms exponential work into linear work, with each error signal computed exactly once.

4.3 Why Backward Order Is Forced

The factorization principle dictates not only *what* to compute but *when*:

- To compute δ_a, we need $\delta_{\mathcal{L}}$ (the error signal at a's child).
- To compute δ_u for any node u, we need δ_v for all children v of u.

In a topological ordering (where parents precede children), children appear after their parents. Therefore, to satisfy dependencies, we must process nodes in *reverse* topological order—from outputs back to inputs. No other order satisfies the dependencies. This is why the algorithm is called *back*propagation.

4.4 The Algorithm

Backpropagation

Input: Computational graph $G = (V, E)$ with nodes in topological order $v_1, \ldots, v_K$ where $v_K = \mathcal{L}$.
Output: Gradients $\partial \mathcal{L} / \partial \theta$ for all parameters $\theta \in \boldsymbol{\theta}$.

1. **Forward pass**: Traverse $v_1, v_2, \ldots, v_K$. Compute and store each node's value.
2. **Initialize**: Set $\delta_{v_K} = \delta_{\mathcal{L}} = 1$.
3. **Backward pass**: Traverse $v_K, v_{K-1}, \ldots, v_1$. For each node v:
   1. For each parent u of v, compute local derivative $\partial v / \partial u$.
   2. Add $\delta_v \cdot (\partial v / \partial u)$ to the accumulator for δ_u.

4. **Return**: The accumulated δ_θ for each parameter θ.

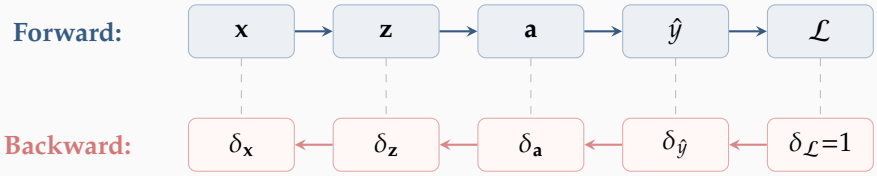

Figure 7: *Forward and backward passes are mirror images. The forward pass computes and stores values; the backward pass uses stored values to compute error signals in reverse order.*

4.5 Computational Complexity

Efficiency of Backpropagation

Backpropagation computes gradients for all parameters in $O(|E|)$ time, where $|E|$ is the number of edges. This is proportional to the forward pass cost.

Analysis:

- **Forward pass**: Each edge traversed once. Cost: $O(|E|)$.
- **Backward pass**: Each edge traversed once (to propagate error signal). Cost: $O(|E|)$.
- **Total**: $O(|E|)$, independent of the number of paths.

Any correct algorithm must use each local derivative at least once. Backpropagation uses each exactly once, achieving the lower bound.

The 3× Rule

In practice, the backward pass costs roughly twice the forward pass (additional memory access and derivative computation). A complete training iteration—forward, loss, backward—costs approximately 3× an inference-only forward pass.

4.6 Forward Mode: The Alternative

There is another way to apply the chain rule: propagate derivatives *forward* from inputs to outputs.

Forward Mode vs. Reverse Mode

Forward mode: For each input, compute how it affects all downstream nodes.

- Cost: $O(|E|)$ per input variable
- For n parameters: $O(n \cdot |E|)$ total

Reverse mode (backpropagation): For the output, compute how all upstream nodes affect it.

- Cost: $O(|E|)$ per output variable
- For 1 scalar loss: $O(|E|)$ total

Neural networks have many parameters (millions or billions) but one scalar loss. Forward mode would require one pass per parameter—catastrophically expensive. Reverse mode requires one pass total. This asymmetry makes backpropagation the only practical choice.

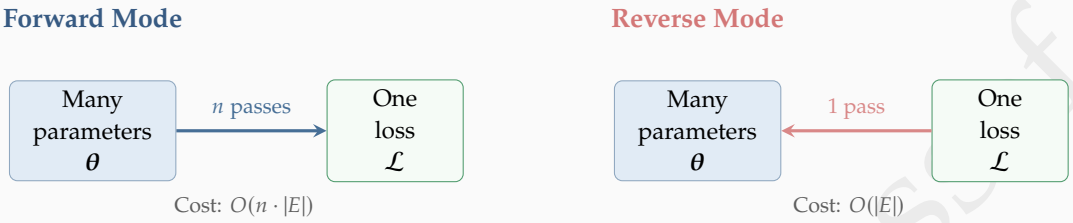

Figure 8: *Forward mode versus reverse mode differentiation. With many parameters and one scalar output, reverse mode (backpropagation) is exponentially more efficient.*

5 Complete Worked Example: Regression

We now verify the theory with a complete numerical example, computing all gradients by hand.

5.1 Setup

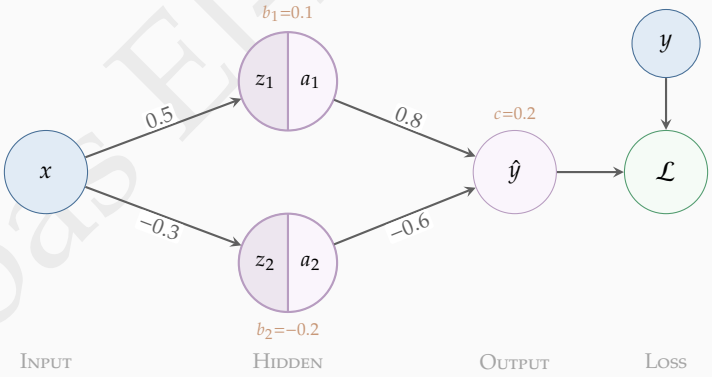

Figure 9: *Network for the regression example. Hidden units show pre-activation (z, lighter) and post-activation (a, darker) with sigmoid applied at the boundary.*

Network parameters:

$$w_1 = 0.5, \quad b_1 = 0.1, \quad v_1 = 0.8$$
$$w_2 = -0.3, \quad b_2 = -0.2, \quad v_2 = -0.6, \quad c = 0.2$$

Training data ($m = 3$ examples):

i	$x^{(i)}$	$y^{(i)}$
1	1.0	0.8
2	2.0	0.5
3	0.5	0.9

Loss function: Mean squared error with the $\frac{1}{2m}$ convention:

$$\mathcal{L} = \frac{1}{2m}\sum_{i=1}^{m}(\hat{y}^{(i)} - y^{(i)})^2 = \frac{1}{6}\sum_{i=1}^{3}(\hat{y}^{(i)} - y^{(i)})^2$$

5.2 Forward Pass

We compute all intermediate values for each training example.
Training example 1 ($x = 1.0$, $y = 0.8$):

$$z_1 = w_1 x + b_1 = 0.5(1.0) + 0.1 = 0.6$$
$$z_2 = w_2 x + b_2 = -0.3(1.0) - 0.2 = -0.5$$
$$a_1 = \sigma(z_1) = \sigma(0.6) = \frac{1}{1+e^{-0.6}} = 0.646$$
$$a_2 = \sigma(z_2) = \sigma(-0.5) = \frac{1}{1+e^{0.5}} = 0.378$$
$$\hat{y} = v_1 a_1 + v_2 a_2 + c = 0.8(0.646) - 0.6(0.378) + 0.2 = 0.490$$

Repeating for all training examples:

i	z_1	z_2	a_1	a_2	$\hat{y}$	y	$\hat{y} - y$	$(\hat{y} - y)^2$
1	0.60	−0.50	0.646	0.378	0.490	0.8	−0.310	0.096
2	1.10	−0.80	0.750	0.310	0.614	0.5	0.114	0.013
3	0.35	−0.35	0.587	0.413	0.422	0.9	−0.478	0.228

Total loss:

$$\mathcal{L} = \frac{1}{6}(0.096 + 0.013 + 0.228) = \frac{0.337}{6} = 0.056$$

5.3 Backward Pass

Step 1: Output error signals
For MSE loss with $m = 3$ and the $\frac{1}{2m}$ convention:

$$\delta_{\hat{y}}^{(i)} = \frac{\partial \mathcal{L}}{\partial \hat{y}^{(i)}} = \frac{1}{m}(\hat{y}^{(i)} - y^{(i)}) = \frac{1}{3}(\hat{y}^{(i)} - y^{(i)})$$

i	$\hat{y}^{(i)} - y^{(i)}$	$\delta_{\hat{y}}^{(i)} = \frac{1}{3}(\hat{y}^{(i)} - y^{(i)})$
1	−0.310	−0.103
2	0.114	0.038
3	−0.478	−0.159

Step 2: Hidden layer error signals
First, propagate to activations:

$$\delta_{a_k}^{(i)} = \delta_{\hat{y}}^{(i)} \cdot v_k$$

Then, propagate through the sigmoid:

$$\delta_{z_k}^{(i)} = \delta_{a_k}^{(i)} \cdot a_k^{(i)}(1 - a_k^{(i)})$$

For hidden unit 1 ($v_1 = 0.8$):

i	$\delta_{\hat{y}}^{(i)}$	$\delta_{a_1}^{(i)} = \delta_{\hat{y}}^{(i)} \cdot 0.8$	$a_1(1-a_1)$	$\delta_{z_1}^{(i)}$
1	−0.103	−0.083	$0.646 \times 0.354 = 0.229$	−0.019
2	0.038	0.030	$0.750 \times 0.250 = 0.188$	0.006
3	−0.159	−0.128	$0.587 \times 0.413 = 0.242$	−0.031

For hidden unit 2 ($v_2 = -0.6$):

i	$\delta_{\hat{y}}^{(i)}$	$\delta_{a_2}^{(i)} = \delta_{\hat{y}}^{(i)} \cdot (-0.6)$	$a_2(1-a_2)$	$\delta_{z_2}^{(i)}$
1	−0.103	0.062	$0.378 \times 0.622 = 0.235$	0.015
2	0.038	−0.023	$0.310 \times 0.690 = 0.214$	−0.005
3	−0.159	0.096	$0.413 \times 0.587 = 0.242$	0.023

5.4 Parameter Gradients

Each gradient is a sum over training examples of (error signal) × (local input).
Output layer parameters:

$$\frac{\partial \mathcal{L}}{\partial c} = \sum_{i=1}^{3} \delta_{\hat{y}}^{(i)} = -0.103 + 0.038 - 0.159 = \boxed{-0.224} \tag{27}$$

$$\frac{\partial \mathcal{L}}{\partial v_1} = \sum_{i=1}^{3} \delta_{\hat{y}}^{(i)} \cdot a_1^{(i)} = (-0.103)(0.646) + (0.038)(0.750) + (-0.159)(0.587)$$
$$= -0.067 + 0.029 - 0.093 = \boxed{-0.131} \tag{28}$$

$$\frac{\partial \mathcal{L}}{\partial v_2} = \sum_{i=1}^{3} \delta_{\hat{y}}^{(i)} \cdot a_2^{(i)} = (-0.103)(0.378) + (0.038)(0.310) + (-0.159)(0.413)$$
$$= -0.039 + 0.012 - 0.066 = \boxed{-0.093} \tag{29}$$

Hidden layer parameters:

$$\frac{\partial \mathcal{L}}{\partial b_1} = \sum_{i=1}^{3} \delta_{z_1}^{(i)} = -0.019 + 0.006 - 0.031 = \boxed{-0.044} \tag{30}$$

$$\frac{\partial \mathcal{L}}{\partial w_1} = \sum_{i=1}^{3} \delta_{z_1}^{(i)} \cdot x^{(i)} = (-0.019)(1.0) + (0.006)(2.0) + (-0.031)(0.5)$$
$$= -0.019 + 0.012 - 0.016 = \boxed{-0.023} \tag{31}$$

$$\frac{\partial \mathcal{L}}{\partial b_2} = \sum_{i=1}^{3} \delta_{z_2}^{(i)} = 0.015 - 0.005 + 0.023 = \boxed{0.033} \tag{32}$$

$$\frac{\partial \mathcal{L}}{\partial w_2} = \sum_{i=1}^{3} \delta_{z_2}^{(i)} \cdot x^{(i)} = (0.015)(1.0) + (-0.005)(2.0) + (0.023)(0.5)$$
$$= 0.015 - 0.010 + 0.012 = \boxed{0.017} \tag{33}$$

Gradient Summary

All seven parameter gradients computed from error signals:

Parameter	Gradient	Update direction
c (output bias)	-0.224	increase
v_1 (output weight)	-0.131	increase
v_2 (output weight)	-0.093	increase
b_1 (hidden bias)	-0.044	increase
w_1 (input weight)	-0.023	increase
b_2 (hidden bias)	$+0.033$	decrease
w_2 (input weight)	$+0.017$	decrease

Gradient descent moves each parameter opposite to its gradient, reducing the loss.

6 Extensions

The single-input network demonstrates all essential mechanisms of backpropagation. We now extend to multiple inputs and classification, showing that the core algorithm remains unchanged.

6.1 Multiple Input Features

Consider a network with $n = 2$ input features x_1 and x_2:

$$z_1 = w_{11}x_1 + w_{12}x_2 + b_1 \tag{34}$$
$$z_2 = w_{21}x_1 + w_{22}x_2 + b_2 \tag{35}$$
$$a_1 = \sigma(z_1), \quad a_2 = \sigma(z_2) \tag{36}$$
$$\hat{y} = v_1 a_1 + v_2 a_2 + c \tag{37}$$

The only change is that each hidden unit now has multiple incoming weights.

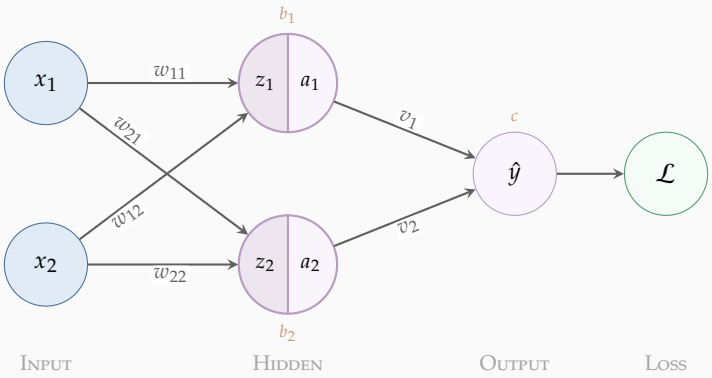

Figure 10: *Network with two input features. Each hidden unit shows pre-activation (z, lighter) and post-activation (a, darker) with sigmoid applied at the boundary.*

Local derivatives: For the pre-activation $z_k = w_{k1}x_1 + w_{k2}x_2 + b_k$:

$$\frac{\partial z_k}{\partial w_{k1}} = x_1, \quad \frac{\partial z_k}{\partial w_{k2}} = x_2, \quad \frac{\partial z_k}{\partial b_k} = 1$$

Backward pass: The error signal δ_{z_k} is computed exactly as before. The gradients are:

$$\frac{\partial \mathcal{L}}{\partial w_{kj}} = \sum_{i=1}^{m} \delta_{z_k}^{(i)} \cdot x_j^{(i)} \tag{38}$$

$$\frac{\partial \mathcal{L}}{\partial b_k} = \sum_{i=1}^{m} \delta_{z_k}^{(i)} \tag{39}$$

The pattern remains: gradient = (error signal) $\times$ (upstream activation).

6.2 Extension to Classification

For binary classification with labels $y \in \{0, 1\}$, we make two changes:

1. **Output activation**: Apply sigmoid to the output:

$$\hat{y} = \sigma(v_1 a_1 + v_2 a_2 + c)$$

2. **Loss function**: Use binary cross-entropy:

$$\mathcal{L} = -\frac{1}{m} \sum_{i=1}^{m} \left[y^{(i)} \ln \hat{y}^{(i)} + (1 - y^{(i)}) \ln(1 - \hat{y}^{(i)}) \right]$$

The key derivative: For cross-entropy loss with sigmoid output, the error signal simplifies elegantly. Let $s = v_1 a_1 + v_2 a_2 + c$ (the pre-sigmoid output), so $\hat{y} = \sigma(s)$.

$$\frac{\partial \mathcal{L}}{\partial \hat{y}^{(i)}} = -\frac{1}{m} \left(\frac{y^{(i)}}{\hat{y}^{(i)}} - \frac{1 - y^{(i)}}{1 - \hat{y}^{(i)}} \right) \tag{40}$$

$$\frac{\partial \hat{y}}{\partial s} = \hat{y}(1 - \hat{y}) \tag{41}$$

Combining these:

$$\delta_s^{(i)} = \frac{\partial \mathcal{L}}{\partial s^{(i)}} = \frac{\partial \mathcal{L}}{\partial \hat{y}^{(i)}} \cdot \frac{\partial \hat{y}}{\partial s}$$

$$= -\frac{1}{m}\left(\frac{y^{(i)}}{\hat{y}^{(i)}} - \frac{1-y^{(i)}}{1-\hat{y}^{(i)}}\right) \cdot \hat{y}^{(i)}(1-\hat{y}^{(i)})$$
$$= -\frac{1}{m}\left(y^{(i)}(1-\hat{y}^{(i)}) - (1-y^{(i)})\hat{y}^{(i)}\right)$$
$$= -\frac{1}{m}\left(y^{(i)} - \hat{y}^{(i)}\right)$$
$$= \frac{1}{m}(\hat{y}^{(i)} - y^{(i)}) \tag{42}$$

Cross-Entropy Gradient

For sigmoid output with cross-entropy loss:

$$\delta_s^{(i)} = \frac{1}{m}(\hat{y}^{(i)} - y^{(i)})$$

This has the same form as the MSE gradient with our $\frac{1}{2m}$ convention. The sigmoid and cross-entropy are "matched"—their derivatives cancel to produce this clean expression.

Backward pass for classification: With $\delta_s^{(i)} = \frac{1}{m}(\hat{y}^{(i)} - y^{(i)})$, we propagate exactly as before:

$$\delta_{a_k}^{(i)} = \delta_s^{(i)} \cdot v_k \tag{43}$$
$$\delta_{z_k}^{(i)} = \delta_{a_k}^{(i)} \cdot a_k^{(i)}(1-a_k^{(i)}) \tag{44}$$

The gradients for all parameters follow the same pattern.

6.3 Worked Classification Example

Consider a multi-output classifier that distinguishes between two categories (nail vs. screw) based on two input features.

Network architecture:

- **Inputs**: Two features x_1, x_2
- **Hidden layer**: Three units with sigmoid activation
- **Output layer**: Two units with sigmoid activation
- **Loss**: Binary cross-entropy per output

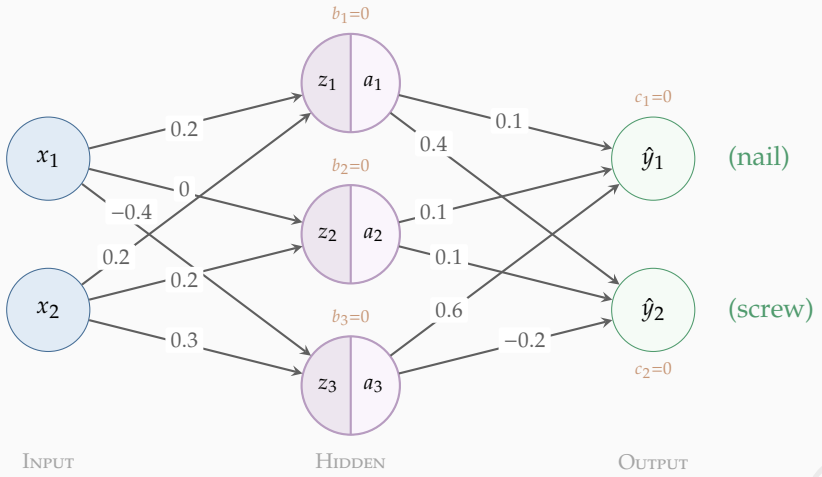

Figure 11: *Classification network with two inputs, three hidden units, and two outputs. Hidden units show pre-activation (z, lighter) and post-activation (a, darker). Edge labels show connection weights. All biases are zero.*

Network parameters:

$$\begin{aligned} \text{Input to hidden:} \quad & w_{11} = 0.2,\ w_{12} = 0.2 \\ & w_{21} = 0.0,\ w_{22} = 0.2 \\ & w_{31} = -0.4,\ w_{32} = 0.3 \\ \text{Hidden to output:} \quad & v_{11} = 0.1,\ v_{12} = 0.1,\ v_{13} = 0.6 \\ & v_{21} = 0.4,\ v_{22} = 0.1,\ v_{23} = -0.2 \end{aligned}$$

All biases are zero: $b_1 = b_2 = b_3 = c_1 = c_2 = 0$.
Training data ($m = 2$ examples):

i	$x_1^{(i)}$	$x_2^{(i)}$	$y_1^{(i)}$ (nail)	$y_2^{(i)}$ (screw)
1	1.0	0.5	0.6	0.1
2	0.8	0.9	0.1	0.9

Forward pass:
Training example 1 ($x_1 = 1.0$, $x_2 = 0.5$, targets: $y_1 = 0.6$, $y_2 = 0.1$):
Hidden layer pre-activations:

$$\begin{aligned} z_1 &= 0.2(1.0) + 0.2(0.5) = 0.30 \\ z_2 &= 0.0(1.0) + 0.2(0.5) = 0.10 \\ z_3 &= -0.4(1.0) + 0.3(0.5) = -0.25 \end{aligned}$$

Hidden layer activations:

$$\begin{aligned} a_1 &= \sigma(0.30) = 0.574 \\ a_2 &= \sigma(0.10) = 0.525 \\ a_3 &= \sigma(-0.25) = 0.438 \end{aligned}$$

Output layer pre-activations:

$$s_1 = 0.1(0.574) + 0.1(0.525) + 0.6(0.438) = 0.373$$

$$s_2 = 0.4(0.574) + 0.1(0.525) + (-0.2)(0.438) = 0.265$$

Output predictions:

$$\hat{y}_1 = \sigma(0.373) = 0.592$$
$$\hat{y}_2 = \sigma(0.265) = 0.566$$

Training example 2 ($x_1 = 0.8$, $x_2 = 0.9$, targets: $y_1 = 0.1$, $y_2 = 0.9$):
Hidden layer:

$$z_1 = 0.2(0.8) + 0.2(0.9) = 0.34 \qquad a_1 = \sigma(0.34) = 0.584$$
$$z_2 = 0.0(0.8) + 0.2(0.9) = 0.18 \qquad a_2 = \sigma(0.18) = 0.545$$
$$z_3 = -0.4(0.8) + 0.3(0.9) = -0.05 \qquad a_3 = \sigma(-0.05) = 0.488$$

Output layer:

$$s_1 = 0.1(0.584) + 0.1(0.545) + 0.6(0.488) = 0.406 \qquad \hat{y}_1 = \sigma(0.406) = 0.600$$
$$s_2 = 0.4(0.584) + 0.1(0.545) + (-0.2)(0.488) = 0.291 \qquad \hat{y}_2 = \sigma(0.291) = 0.572$$

Backward pass:
Output error signals (using equation 42 with $m = 2$):

$$\delta_{s_1}^{(1)} = \frac{1}{2}(\hat{y}_1^{(1)} - y_1^{(1)}) = \frac{1}{2}(0.592 - 0.6) = -0.004$$
$$\delta_{s_1}^{(2)} = \frac{1}{2}(\hat{y}_1^{(2)} - y_1^{(2)}) = \frac{1}{2}(0.600 - 0.1) = \quad 0.250$$
$$\delta_{s_2}^{(1)} = \frac{1}{2}(\hat{y}_2^{(1)} - y_2^{(1)}) = \frac{1}{2}(0.566 - 0.1) = \quad 0.233$$
$$\delta_{s_2}^{(2)} = \frac{1}{2}(\hat{y}_2^{(2)} - y_2^{(2)}) = \frac{1}{2}(0.572 - 0.9) = -0.164$$

Hidden layer error signals for unit 3 (showing one complete path):

$$\delta_{a_3}^{(1)} = \delta_{s_1}^{(1)} \cdot v_{13} + \delta_{s_2}^{(1)} \cdot v_{23}$$
$$= (-0.004)(0.6) + (0.233)(-0.2) = -0.049$$
$$\delta_{z_3}^{(1)} = \delta_{a_3}^{(1)} \cdot a_3^{(1)}(1 - a_3^{(1)}) = (-0.049)(0.438)(0.562) = -0.012$$
$$\delta_{a_3}^{(2)} = (0.250)(0.6) + (-0.164)(-0.2) = 0.183$$
$$\delta_{z_3}^{(2)} = (0.183)(0.488)(0.512) = 0.046$$

Selected gradients:
For hidden-to-output weight v_{13}:

$$\frac{\partial \mathcal{L}}{\partial v_{13}} = \sum_i \delta_{s_1}^{(i)} \cdot a_3^{(i)} = (-0.004)(0.438) + (0.250)(0.488)$$
$$= -0.002 + 0.122 = \boxed{0.120} \tag{45}$$

For input-to-hidden weight w_{31}:

$$\frac{\partial \mathcal{L}}{\partial w_{31}} = \sum_i \delta_{z_3}^{(i)} \cdot x_1^{(i)} = (-0.012)(1.0) + (0.046)(0.8)$$

$$= -0.012 + 0.037 = \boxed{0.025} \tag{46}$$

For input-to-hidden weight w_{32}:

$$\frac{\partial \mathcal{L}}{\partial w_{32}} = \sum_i \delta_{z_3}^{(i)} \cdot x_2^{(i)} = (-0.012)(0.5) + (0.046)(0.9)$$
$$= -0.006 + 0.041 = \boxed{0.035} \tag{47}$$

Multi-Output Networks

In multi-output networks, each hidden unit's error signal receives contributions from *all* outputs it connects to. For hidden unit 3:

$$\delta_{a_3} = \delta_{s_1} \cdot v_{13} + \delta_{s_2} \cdot v_{23}$$

The gradient flows back through both output paths simultaneously, and the hidden unit's total responsibility is the sum of its effects on all outputs.

7 Practical Considerations

The mathematical elegance of backpropagation meets physical constraints when implemented on real hardware. These constraints shape every aspect of modern deep learning practice.

7.1 Memory Requirements

The backward pass requires values computed during the forward pass. To compute $\delta_{z_k} = \delta_{a_k} \cdot a_k(1 - a_k)$, we need the stored activation a_k. This creates a fundamental memory cost. For a network with L layers, each producing activations of dimension $n^{[\ell]}$, processing a batch of B examples:

$$\text{Activation memory} = O\left(B \cdot \sum_{\ell=1}^{L} n^{[\ell]}\right)$$

This often exceeds the memory needed for parameters. For deep networks like ResNet-152 processing batches of images, activation storage can require several gigabytes.

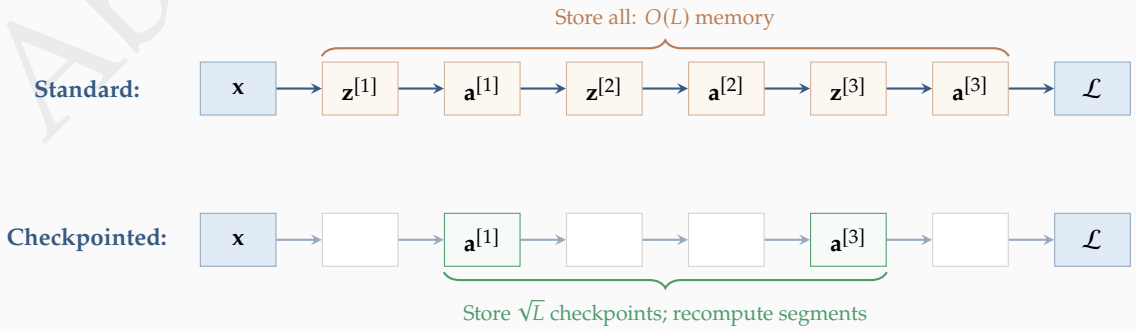

Figure 12: *Memory trade-off in backpropagation.* ***Top****: Standard approach stores all intermediate activations ($O(L)$ memory).* ***Bottom****: Gradient checkpointing stores only selected checkpoints ($O(\sqrt{L})$ memory) and recomputes intermediate values during the backward pass.*

Gradient checkpointing trades computation for memory. Instead of storing all activations, store only selected "checkpoints." When the backward pass needs intermediate values, recompute them from the nearest checkpoint.
With $\sqrt{L}$ checkpoints for a network of depth L:

- Memory: reduced from $O(L)$ to $O(\sqrt{L})$
- Computation: one additional forward pass (recomputing segments)

7.2 Numerical Stability

The error signal at layer ℓ involves a product of local derivatives from the output back to layer ℓ:

$$\delta^{[\ell]} \propto \prod_{j=\ell}^{L-1} \frac{\partial \mathbf{a}^{[j]}}{\partial \mathbf{z}^{[j]}} \cdot \frac{\partial \mathbf{z}^{[j+1]}}{\partial \mathbf{a}^{[j]}}$$

The behavior of this product depends critically on the magnitude of each factor.

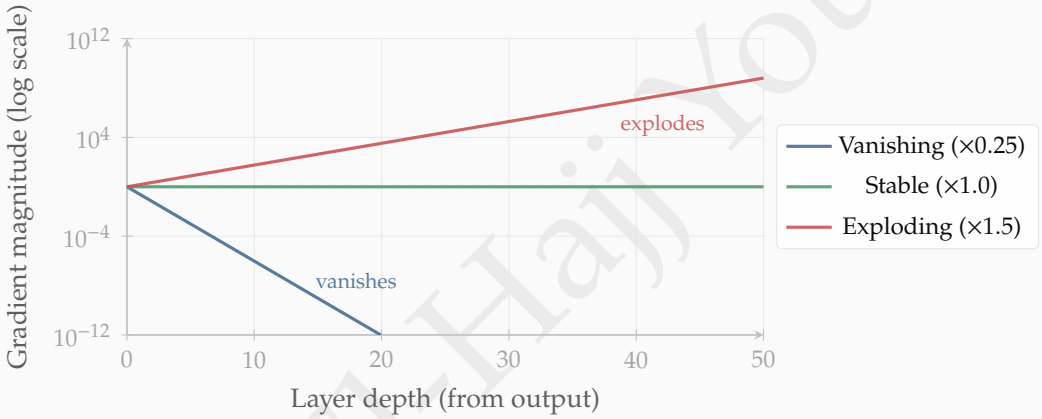

Figure 13: *Gradient magnitude versus depth. The sigmoid derivative is at most 0.25, causing gradients to shrink exponentially. Factors above 1 cause exponential growth.*

Vanishing and Exploding Gradients

Vanishing gradients: When local derivatives are consistently less than 1, gradients shrink exponentially with depth. Early layers receive negligible error signals and cannot learn.
Exploding gradients: When local derivatives exceed 1, gradients grow exponentially. Weight updates become unstable, and training diverges.

The sigmoid activation has maximum derivative $\sigma'(0) = 0.25$. In a deep sigmoid network, gradients decay by at least a factor of 4 per layer. After 20 layers, the gradient is attenuated by $0.25^{20} \approx 10^{-12}$—effectively zero in floating-point arithmetic.

Solutions developed by the field:

Technique	Mechanism
ReLU activation	$\text{ReLU}'(z) = 1$ for $z > 0$, avoiding the bounded derivative of sigmoids
Careful initialization	Xavier/He initialization sets weight variance to preserve signal magnitude across layers
Batch normalization	Normalizes layer inputs, preventing distribution shift during training
Residual connections	Skip connections provide gradient highways: $\partial(\mathbf{x} + f(\mathbf{x}))/\partial\mathbf{x} = \mathbf{I} + \partial f/\partial\mathbf{x}$
Gradient clipping	Rescales gradients when their norm exceeds a threshold

Why Residual Connections Work

A residual block computes $\mathbf{a}^{[\ell+1]} = \mathbf{a}^{[\ell]} + f(\mathbf{a}^{[\ell]})$. The gradient is:

$$\frac{\partial \mathbf{a}^{[\ell+1]}}{\partial \mathbf{a}^{[\ell]}} = \mathbf{I} + \frac{\partial f}{\partial \mathbf{a}^{[\ell]}}$$

Even if $\partial f/\partial \mathbf{a}^{[\ell]}$ is small, the gradient includes an additive identity. This "identity shortcut" allows gradients to flow directly through the network, enabling training of hundreds of layers.

7.3 Batching and Parallelism

We have presented gradients as sums over training examples:

$$\frac{\partial \mathcal{L}}{\partial \theta} = \sum_{i=1}^{m} \frac{\partial \mathcal{L}^{(i)}}{\partial \theta}$$

In practice, we rarely use all m examples at once. Instead, training proceeds in *mini-batches*: subsets of B examples (typically 32–512) processed together.

Stochastic gradient descent uses the mini-batch gradient as an estimate of the full gradient:

$$\frac{\partial \mathcal{L}}{\partial \theta} \approx \frac{1}{B} \sum_{i \in \text{batch}} \frac{\partial \mathcal{L}^{(i)}}{\partial \theta}$$

This estimate is unbiased—its expectation equals the true gradient—but has variance that decreases as $1/B$. The choice of batch size involves a fundamental trade-off:

- **Smaller batches**: Higher variance in gradient estimates, but more frequent parameter updates per epoch. Often leads to better generalization due to the regularizing effect of gradient noise.
- **Larger batches**: Lower variance, more stable convergence, and better utilization of parallel hardware. However, may require careful learning rate tuning and can generalize less well.

Parallelism: Within a batch, each example's forward and backward pass is independent. Modern hardware (GPUs, TPUs) exploits this by processing all B examples simultaneously. Activations become tensors of shape (batch_size, layer_width), and backpropagation operates on these batched tensors.
The computational graph remains the same; only the data flowing through it gains an extra dimension.

7.4 Automatic Differentiation

No one implements backpropagation by hand for real networks. Modern frameworks—PyTorch, TensorFlow, JAX—provide *automatic differentiation*: you specify the forward computation, and the framework constructs and executes the backward pass.
The key abstraction: each operation is implemented as a pair:

1. **Forward**: Compute output from inputs; save values needed for backward
2. **Backward**: Given output error signal, compute input error signals

Autodiff for Sigmoid

The sigmoid operation:
Forward:

- Input: z
- Compute: $a = 1/(1 + e^{-z})$
- Save: a (needed for backward)
- Return: a

Backward:

- Input: δ_a (error signal at output)
- Retrieve: saved a
- Compute: $\delta_z = \delta_a \cdot a \cdot (1 - a)$
- Return: δ_z

The framework chains these primitives automatically, building the backward pass from the forward computation graph.

8 Summary

Backpropagation computes gradients for all parameters in time proportional to the forward pass. The algorithm rests on three principles:

1. **Computation as a graph**: Represent the network as a directed acyclic graph where nodes are values and edges are dependencies.
2. **Local-to-global via chain rule**: The error signal at any node is

$$\delta_u = \sum_{\text{children } v} \delta_v \cdot \frac{\partial v}{\partial u}$$

Each child reports its sensitivity to the parent; the parent sums these contributions.

3. **Reverse traversal**: Computing δ_u requires δ_v for all children, forcing reverse topological order—hence *back*propagation.

The gradient for any parameter follows a universal pattern:

$$\frac{\partial \mathcal{L}}{\partial \theta} = \sum_{i=1}^{m} \delta^{(i)} \cdot a^{(i)}$$

where $\delta^{(i)}$ is the error signal reaching the parameter and $a^{(i)}$ is the upstream activation it multiplies.

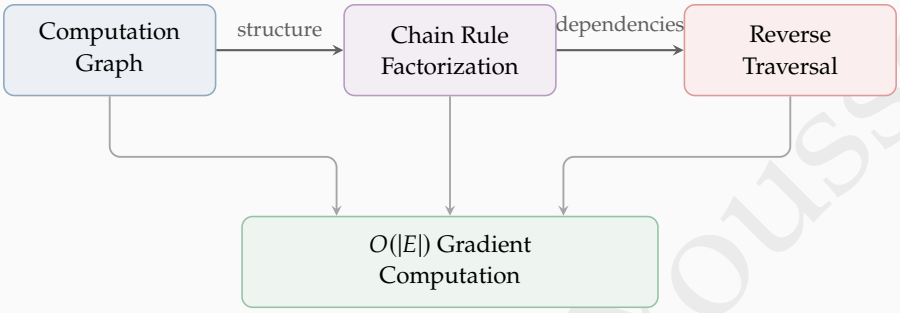

Figure 14: *The three pillars of backpropagation. The graph structure enables factorization via the chain rule, which imposes dependency constraints that force reverse traversal. Together, these yield linear-time gradient computation.*

This algorithm achieves $O(|E|)$ complexity by computing each error signal exactly once, compared to the exponential cost of naively enumerating paths. The efficiency, combined with automatic differentiation in modern frameworks, enables training networks with billions of parameters.

The Essence of Backpropagation

Backpropagation is not a heuristic or an approximation—it is the *exact* and *unique* efficient algorithm for computing gradients in computational graphs. Its discovery was inevitable: anyone reasoning carefully about the structure of gradient computation in composite systems must arrive at the same answer.

Appendix: Notation

Network Architecture

L	Number of layers (excl. input)
ℓ	Layer index, $\ell \in \{0, 1, \ldots, L\}$
$n^{[\ell]}$	Number of units in layer ℓ
n	Input features (equiv. $n^{[0]}$)
$n_{\text{in}}, n_{\text{out}}$	Input/output dimensions

Data and Indexing

m	Number of training examples
K	Number of classes (classification)
$(\cdot)^{(i)}$	Quantity for training example i
$(\cdot)^{[\ell]}$	Quantity for layer ℓ
$(\cdot)_j$	j-th component of a vector

Inputs, Outputs, and Targets

$\mathbf{x} \in \mathbb{R}^n$	Input vector
$\mathbf{X}$	Data matrix (examples as columns)
y	Target value (scalar)
$\mathbf{y}$	Target vector
$\hat{y}$	Predicted output (scalar)
$\hat{\mathbf{y}}$	Predicted output (vector)

Parameters

$\mathbf{W}^{[\ell]}$	Weight matrix for layer ℓ
$\mathbf{b}^{[\ell]}$	Bias vector for layer ℓ
$w_{jk}^{[\ell]}$	Weight: unit k (layer $\ell-1$) to j (layer ℓ)
w, b	Scalar weight and bias (single unit)
$\boldsymbol{\theta}$	All parameters: $\{\mathbf{W}^{[\ell]}, \mathbf{b}^{[\ell]}\}_{\ell=1}^{L}$

Forward Propagation

$\mathbf{z}^{[\ell]}$	Pre-activation at layer ℓ
$\mathbf{a}^{[\ell]}$	Activation at layer ℓ
$\mathbf{a}^{[0]} = \mathbf{x}$	Input layer convention
$g^{[\ell]}(\cdot)$	Activation function for layer ℓ
$\sigma(z)$	Sigmoid: $1/(1+e^{-z})$
$\tanh(z)$	Hyperbolic tangent
$\text{ReLU}(z)$	Rectified linear unit: $\max(0, z)$
$\text{softmax}(\mathbf{z})_k$	Softmax: $e^{z_k}/\sum_j e^{z_j}$

Loss and Optimization

$\mathcal{L}(\boldsymbol{\theta})$	Loss function
$\nabla_{\boldsymbol{\theta}} \mathcal{L}$	Gradient of loss w.r.t. parameters
$\partial \mathcal{L}/\partial \theta$	Partial derivative w.r.t. θ
α	Learning rate

Backpropagation

$\delta^{[\ell]}$	Error signal at layer ℓ: $\partial \mathcal{L}/\partial \mathbf{z}^{[\ell]}$
$\delta_j^{[\ell]}$	Error signal for unit j in layer ℓ

General Mathematics

$\mathbb{R}$	Set of real numbers
$\mathbb{R}^n$	n-dimensional Euclidean space
$\mathbf{v}^\top$	Transpose of vector $\mathbf{v}$
$\|\mathbf{v}\|$	Euclidean norm of vector $\mathbf{v}$
$\|\mathbf{v}\|_2$	Euclidean (L_2) norm
$\odot$	Element-wise (Hadamard) product

Conventions

- Vectors are lowercase bold: $\mathbf{x}, \mathbf{z}, \mathbf{a}$
- Matrices are uppercase bold: $\mathbf{W}, \mathbf{X}$
- Scalars are italic: m, L, α
- Layer indices as superscripts in brackets: $\mathbf{W}^{[\ell]}$
- Example indices as superscripts in parentheses: $\mathbf{x}^{(i)}$
- Vector components as subscripts: x_j, $a_k^{[\ell]}$