
Training Neural Networks

Last updated October 16, 2026

David I. Inouye

| We still use gradient descent—but where do the gradients come from?

$$\theta \leftarrow \theta - \eta \nabla_{\theta} J.$$

The parameters now sit inside a composition of many layers.

Backpropagation computes $\nabla_{\theta} J$ efficiently. Gradient descent (or Adam) uses those gradients to update the parameters.

Training loop: forward computation $\rightarrow$ backpropagation $\rightarrow$ parameter update.

| Backpropagation evaluates **the chain rule from the loss backward**

For $a = f(\theta)$, $b = g(a)$, and $\ell = q(b)$:

$$\frac{\partial \ell}{\partial \theta} = \frac{\partial \ell}{\partial b} \frac{\partial b}{\partial a} \frac{\partial a}{\partial \theta}.$$

Forward: $\theta \rightarrow a \rightarrow b \rightarrow \ell$; save the values needed by the derivatives.

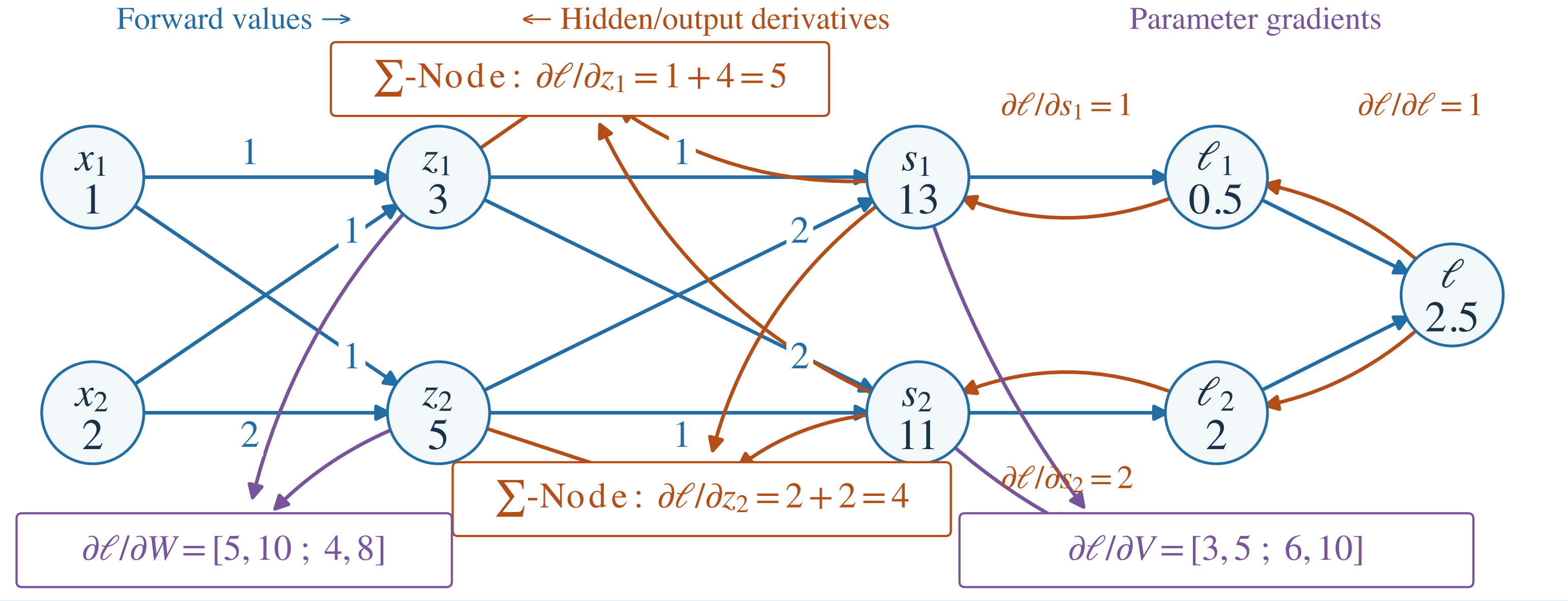
Backward: start with $\frac{\partial \ell}{\partial \ell} = 1$, then compute

$$\frac{\partial \ell}{\partial b} = \frac{\partial q(b)}{\partial b}, \quad \frac{\partial \ell}{\partial a} = \frac{\partial \ell}{\partial b} \frac{\partial b}{\partial a}, \quad \frac{\partial \ell}{\partial \theta} = \frac{\partial \ell}{\partial a} \frac{\partial a}{\partial \theta}.$$

Same chain rule, reversed evaluation order. We need the forward output to evaluate the loss's derivative, and saved intermediate values to evaluate the local derivatives.

| Backward computation sums paths at hidden features

13/13 Matrix Summary: Accumulate, Then Form Parameter Gradients



$$\frac{\partial \ell}{\partial \mathbf{z}} = \mathbf{V}^\top \frac{\partial \ell}{\partial \mathbf{s}} = [5, 4]^\top$$
$$\frac{\partial \ell}{\partial \mathbf{V}} = \begin{bmatrix} 3 & 5 \\ 6 & 10 \end{bmatrix} \qquad \frac{\partial \ell}{\partial \mathbf{W}} = \begin{bmatrix} 5 & 10 \\ 4 & 8 \end{bmatrix}$$

Matrix multiplication performs the same path sums. No gradient with respect to $\mathbf{x}$ is computed.

| Training repeats **forward, backward, and parameter updates**

For a batch $\mathcal{B}$ of B examples, use the mean loss:

$$J_{\mathcal{B}}(\boldsymbol{\theta}) = \frac{1}{B} \sum_{i \in \mathcal{B}} \ell_i(\boldsymbol{\theta}), \quad \nabla_{\boldsymbol{\theta}} J_{\mathcal{B}} = \frac{1}{B} \sum_{i \in \mathcal{B}} \nabla_{\boldsymbol{\theta}} \ell_i.$$

1. **Start:** initialize the parameters once; choose a batch and clear old gradient accumulators.
2. **Forward:** compute predictions and the batch loss; save values needed for backward.
3. **Backward:** accumulate gradients for **all trainable parameters**, using the same parameter values.
4. **Update:** $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} J_{\mathcal{B}}$; repeat with a fresh forward pass.

An **epoch** visits the training set once. Monitor validation performance across epochs to choose when to stop.

| Libraries combine these rules **to differentiate the whole network**

1. **Forward:** record the operations and retain the values their backward rules need.
2. **Backward:** start at the scalar loss, apply local rules in reverse order, and sum contributions from shared paths.
3. **Update:** use the resulting gradient for every trainable parameter.

For an MLP with biases, that includes W, b, V, c .

We specify the model and loss; automatic differentiation organizes the chain rule.

Next topic: implement this training cycle in PyTorch.

Backpropagation computes **all** parameter gradients efficiently

$$\text{In 1D: } \frac{\partial J}{\partial \theta} \approx \frac{J(\theta + \epsilon) - J(\theta)}{\epsilon}.$$

One objective evaluation takes **1 second**. Reuse the unperturbed value $J(\theta)$.

Finite differences: about how long for **10,000 parameters**?

- A. 2 seconds
- B. 100 seconds
- C. 10,000 seconds

C: 10,000 parameters → **about 10,000 seconds**.

One new evaluation per parameter.

Backpropagation: how long for the **same full gradient**?

Assume backward $\approx$ forward cost.

- A. 2 seconds
- B. 100 seconds
- C. 10,000 seconds

A: about 2 seconds for forward + backward.

One reverse pass reuses work for **all parameters**.