An Exercise in Backpropagation

CS231N: Deep Learning for Computer Vision

Favour Nerrise

Stanford University

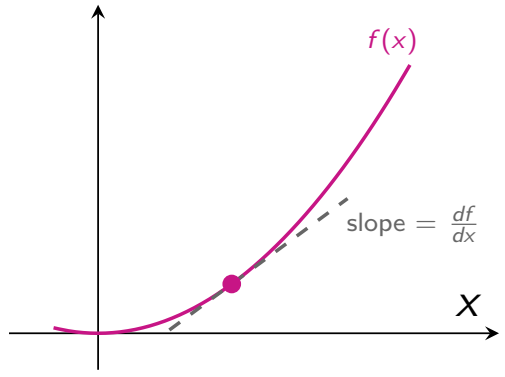
Spring 2026

Quick review: what is a gradient?

Derivative (1 variable)

How much does $f(x)$ change when we slightly change x?

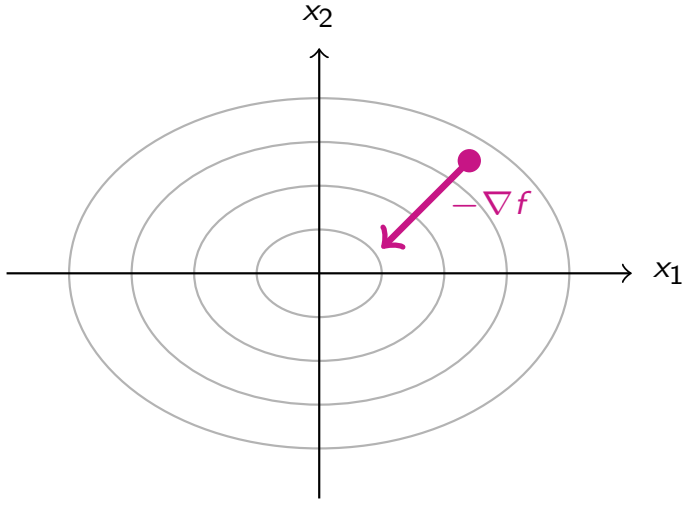
$$\frac{df}{dx} \approx \frac{f(x+h) - f(x)}{h}$$



Gradient (many variables)

Stack all partial derivatives into a vector:

$$\nabla f = \left[\frac{\partial f}{\partial x_1} \quad \cdots \quad \frac{\partial f}{\partial x_n} \right]^\top$$



In neural networks: f = loss, inputs = weights.

The gradient tells us how to **adjust each weight** to reduce the *loss*.

What, when, and why backpropagation?

What

A method for calculating gradients of the output of a chain of operations with respect to each component in the chain.

When

Used in anything and everything neural networks related.

Why

Computing gradients by hand doesn't scale. Backprop gives us a modular, efficient algorithm by applying the chain rule on a computational graph.

Why backpropagation? (in more detail)

We need gradient $\nabla_\theta \mathcal{L}$ ($\mathcal{L}$ = loss) to do gradient descent.
How do we get it?

Option 1: by hand $\times$

- Tedious, error-prone
- Doesn't scale
- Change the loss? Rederive!

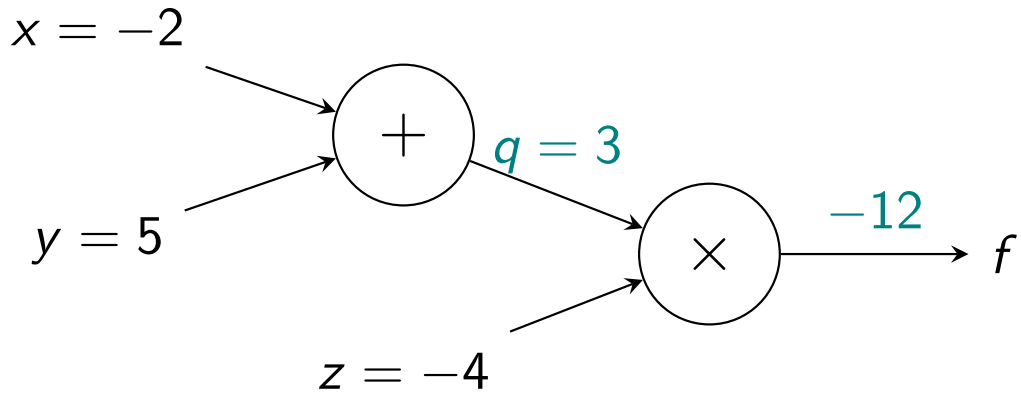
Option 2: backprop ✓

- Modular: each node computes its own local gradient
- Compose any network!

Backpropagation = **recursive chain rule** on a computational graph.

Warm-up: a simple computational graph

Consider $f(x, y, z) = (x + y) \cdot z$. Let $x = -2,\ y = 5,\ z = -4$.

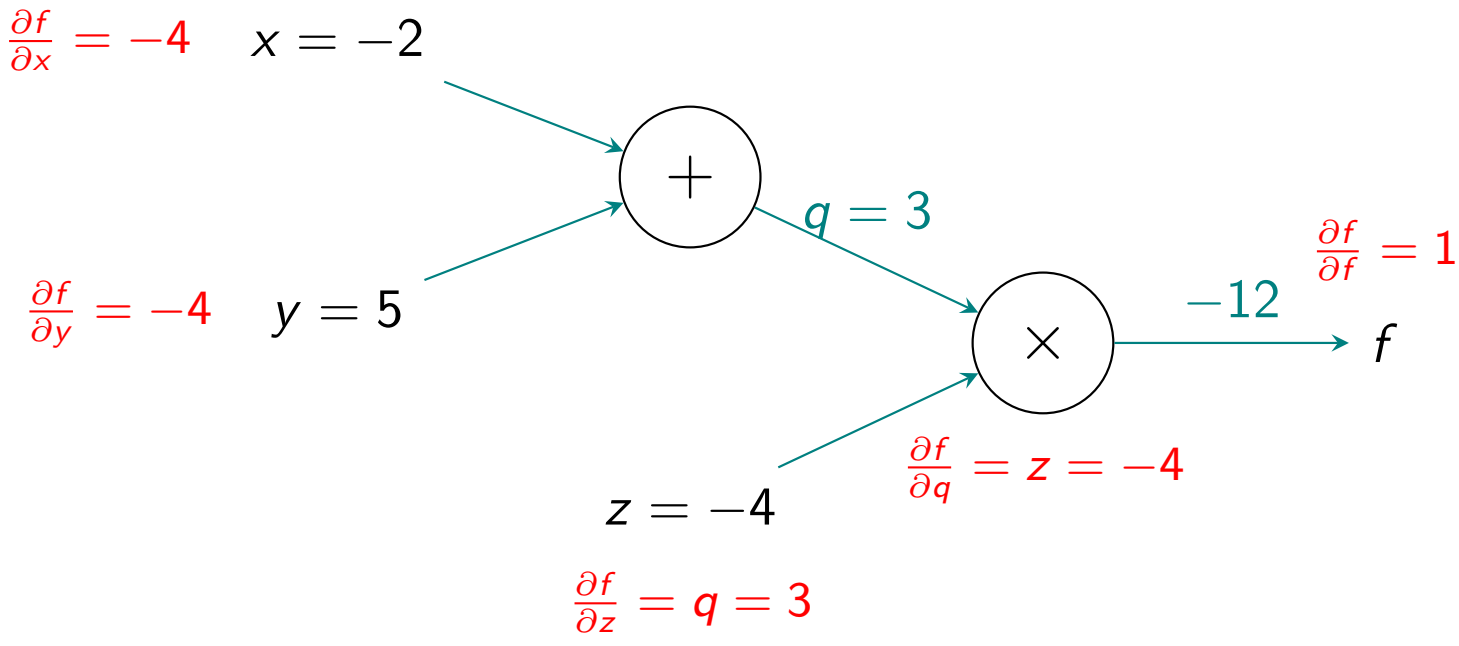


Forward pass (green values): compute the output left-to-right.

Question: How much does f change if we change x, y, or z slightly? That is, what are $\frac{\partial f}{\partial x}$, $\frac{\partial f}{\partial y}$, $\frac{\partial f}{\partial z}$?

Backward pass: tracing gradients

Green = forward pass values. Red = gradients flowing backward.



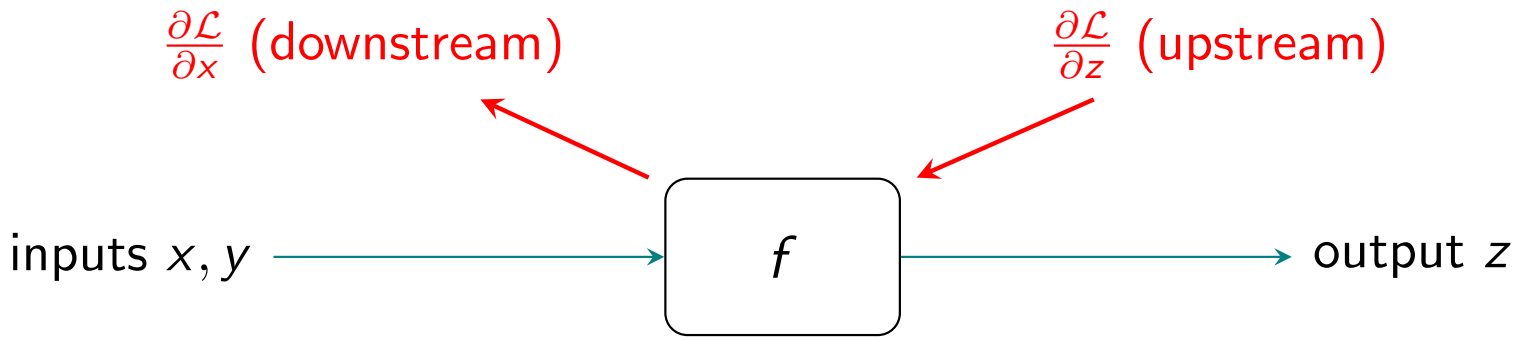
At the $\times$ gate: local gradients are $\frac{\partial(qz)}{\partial q} = z$ and $\frac{\partial(qz)}{\partial z} = q$. Multiply by upstream gradient (1).

At the $+$ gate: local gradients are both 1. Multiply by upstream gradient (-4).

The core pattern: upstream $\times$ local

Each node in the graph receives an **upstream gradient** and computes:

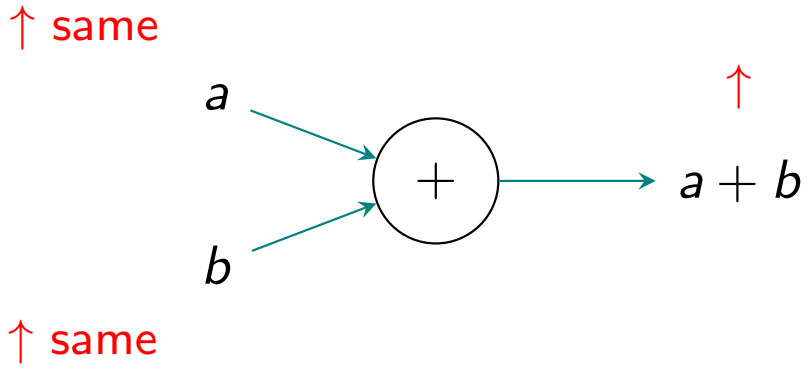
$$\text{downstream gradient} = \underbrace{\frac{\partial \mathcal{L}}{\partial \text{output}}}_{\text{upstream}} \times \underbrace{\frac{\partial \text{ output}}{\partial \text{input}}}_{\text{local}}$$



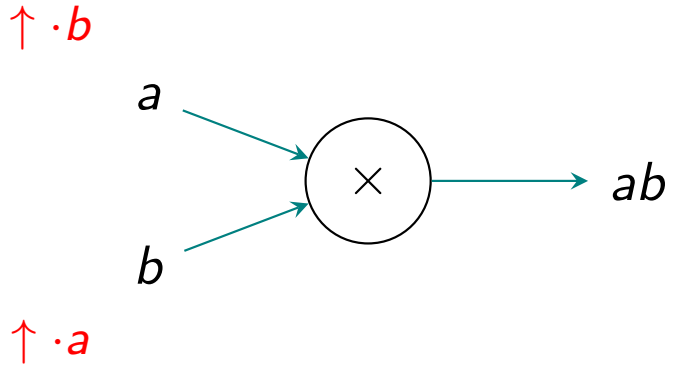
Why this matters: each node only needs to know its own operation and the upstream gradient. Nothing else. That's what makes backprop modular and scalable.

Patterns in gradient flow

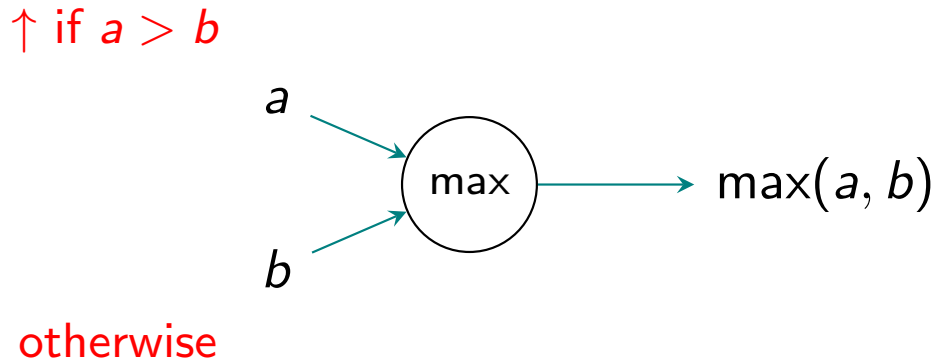
Add gate: distributes gradient



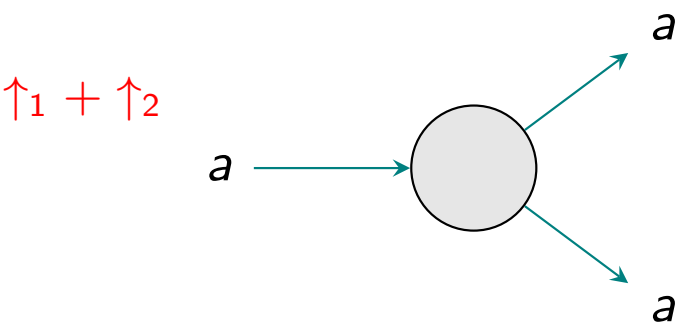
Multiply gate: swaps and scales



Max gate: routes to the winner



Copy / branch: adds gradients



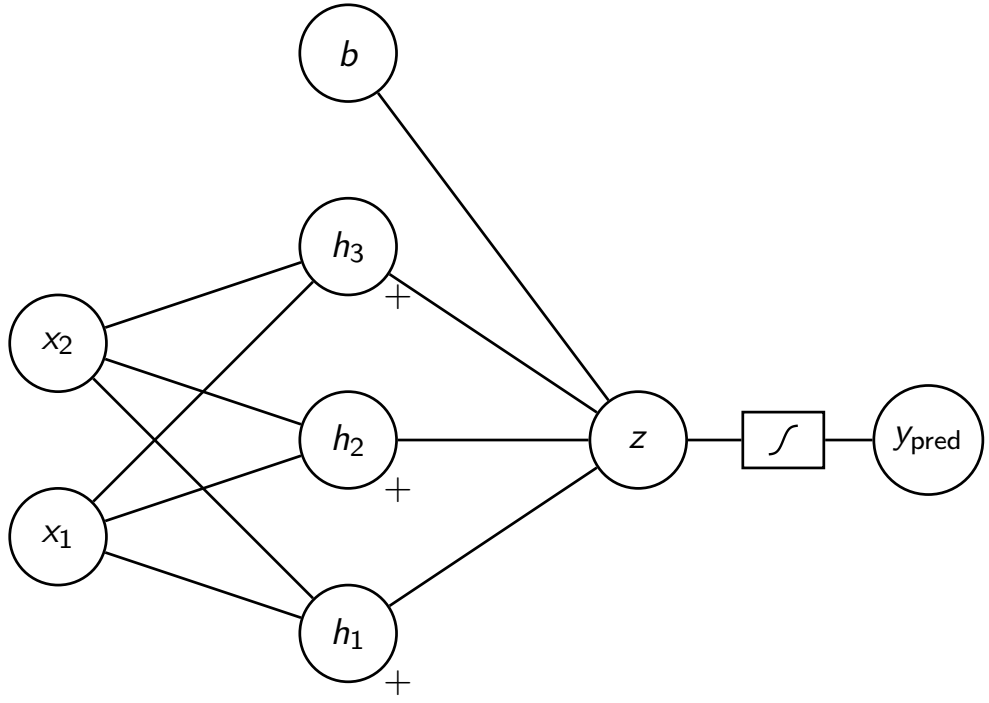
These patterns are your visual toolkit for tracing gradients through any network!

Definitions

We'll be exploring a simple neural network for the binary (0 or 1) classification problem.

- Data: $(\mathbf{X}, \mathbf{y})$ are the dataset and corresponding labels
  - $\mathbf{X} \in \mathbb{R}^{N \times 2}, \mathbf{y} \in \{0, 1\}^N$
- Model: $f_\theta(\mathbf{x}) : \mathbb{R}^2 \to [0, 1]$ (probability $\mathbf{x}$ is in class 1)
  - θ represents all the parameters we want to optimize!
- Activation functions: ReLU (x_+), Sigmoid ($\sigma(x)$)
  - $x_+ = \max(0, x)$
  - $\sigma(x) = \frac{1}{1+e^{-x}}$
- Binary Cross Entropy Loss:
  - $\ell_i(p_i, y_i) = -(y_i \cdot \log p_i + (1 - y_i) \cdot \log(1 - p_i))$
  - $\ell(\mathbf{p}, \mathbf{y}) = \frac{1}{N} \sum_{i=1}^{N} \ell_i(p_i, y_i)$

The Network

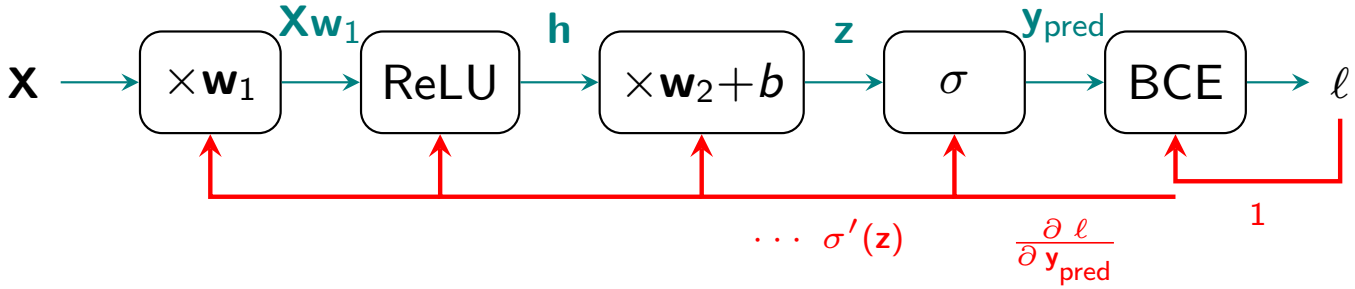


$$f_\theta(\mathbf{x}) = \sigma(\underbrace{\max(0, \mathbf{x}\mathbf{w}_1)\mathbf{w}_2 + b}_{\mathbf{z}})$$

We want to optimize the network's parameters θ which consist of $\mathbf{w}_1 \in \mathbb{R}^{2\times 3}$, $\mathbf{w}_2 \in \mathbb{R}^{3\times 1}$, and $b \in \mathbb{R}^1$.

Gradient flow through our network

Let's draw the computation as a graph and trace gradients backward:



Backward pass (red): start from ℓ and multiply upstream $\times$ local at each node, moving right to left.

We'll compute each local gradient one node at a time.

From scalars to vectors

Our warm-up used scalars. Our network uses vectors and matrices.

Mapping	**Derivative is a...**	**Shape**
Scalar $\to$ Scalar	number	1
Vector $\to$ Scalar	gradient	same as input
Vector $\to$ Vector	Jacobian matrix	$m \times n$

Shape rule

$\frac{\partial \ell}{\partial \mathbf{x}}$ always has the *same shape* as $\mathbf{x}$ (since ℓ is a scalar). Use this to check your work!

Step 1: BCE loss node

Forward

$\ell = -\frac{1}{N}\sum_i \left(y_i \log p_i + (1-y_i)\log(1-p_i)\right) \quad$ where $p_i = (\mathbf{y}_{\text{pred}})_i$

Local gradient (upstream $= 1$ since ℓ is the root)

$$\frac{\partial \ell}{\partial \mathbf{y}_{\text{pred}}} = \frac{1}{N}\left(-\frac{\mathbf{y}}{\mathbf{y}_{\text{pred}}} + \frac{1-\mathbf{y}}{1-\mathbf{y}_{\text{pred}}}\right)$$

Shape check

ℓ is a scalar, $\mathbf{y}_{\text{pred}} \in \mathbb{R}^N \Rightarrow \frac{\partial \ell}{\partial \mathbf{y}_{\text{pred}}} \in \mathbb{R}^N$ ✓

Step 2: Sigmoid node

Forward

$\mathbf{y}_{\text{pred}} = \sigma(\mathbf{z})$ (element-wise)

Local gradient

$$\frac{\partial \mathbf{y}_{\text{pred}}}{\partial \mathbf{z}} = \sigma(\mathbf{z})\,(1 - \sigma(\mathbf{z})) = \mathbf{y}_{\text{pred}} \cdot (1 - \mathbf{y}_{\text{pred}})$$

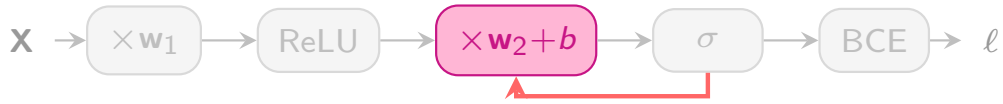
Chain together: upstream × local

$$\frac{\partial \ell}{\partial \mathbf{z}} = \frac{\partial \ell}{\partial \mathbf{y}_{\text{pred}}} \cdot \mathbf{y}_{\text{pred}}(1 - \mathbf{y}_{\text{pred}})$$

Sigmoid trick

$\sigma'(x) = \sigma(x)(1 - \sigma(x))$: the local gradient depends only on the *output* we already cached in the forward pass!

Step 3: Linear layer $\mathbf{z} = \mathbf{h}\mathbf{w}_2 + b$



Upstream gradient $\frac{\partial \ell}{\partial \mathbf{z}}$ already computed. Now we need $\frac{\partial \ell}{\partial \mathbf{w}_2}$, $\frac{\partial \ell}{\partial b}$, and $\frac{\partial \ell}{\partial \mathbf{h}}$.

Gradient w.r.t. $\mathbf{w}_2$

$$\frac{\partial \ell}{\partial \mathbf{w}_2} = \mathbf{h}^\top \frac{\partial \ell}{\partial \mathbf{z}} \qquad (\mathbb{R}^{3\times 1} = \mathbb{R}^{3\times N} . \mathbb{R}^{N\times 1})$$

Gradient w.r.t. b

b is broadcast across the batch, so: $\frac{\partial \ell}{\partial b} = \sum_i \frac{\partial \ell}{\partial z_i}$

Gradient to pass upstream w.r.t. $\mathbf{h}$

$$\frac{\partial \ell}{\partial \mathbf{h}} = \frac{\partial \ell}{\partial \mathbf{z}} \mathbf{w}_2^\top \qquad (\mathbb{R}^{N\times 3} = \mathbb{R}^{N\times 1} . \mathbb{R}^{1\times 3})$$

Step 4: **ReLU** node

Forward

$\mathbf{h} = \max(0,\ \mathbf{X}\mathbf{w}_1)$ (element-wise)

Local gradient

$$\frac{\partial\, h_{ij}}{\partial\, (\mathbf{X}\mathbf{w}_1)_{ij}} = \begin{cases} 1 & \text{if } (\mathbf{X}\mathbf{w}_1)_{ij} > 0 \\ 0 & \text{otherwise} \end{cases}$$

Chain together

$$\frac{\partial\, \ell}{\partial\, \mathbf{X}\mathbf{w}_1} = \frac{\partial\, \ell}{\partial\, \mathbf{h}} \odot \mathbf{1}[\mathbf{X}\mathbf{w}_1 > 0]$$

Max gate pattern

ReLU is just a max gate: it **routes** the gradient where the input was positive and **kills** it where the input was ≤ 0.

Step 5: Linear layer $\mathbf{X}\mathbf{w}_1$

Upstream gradient $\frac{\partial \ell}{\partial \mathbf{X}\mathbf{w}_1}$ already computed.

Gradient w.r.t. $\mathbf{w}_1$

$$\frac{\partial \ell}{\partial \mathbf{w}_1} = \mathbf{X}^\top \frac{\partial \ell}{\partial \mathbf{X}\mathbf{w}_1} \qquad (\mathbb{R}^{2\times 3} = \mathbb{R}^{2\times N} . \mathbb{R}^{N\times 3})$$

Same pattern as Step 3

For $Y = XW$: $\frac{\partial \ell}{\partial W} = X^\top \frac{\partial \ell}{\partial Y}$ and $\frac{\partial \ell}{\partial X} = \frac{\partial \ell}{\partial Y} W^\top$

This is the multiply gate pattern applied to matrices!

We're done!

We now have $\frac{\partial \ell}{\partial \mathbf{w}_1}$, $\frac{\partial \ell}{\partial \mathbf{w}_2}$, and $\frac{\partial \ell}{\partial \boldsymbol{b}}$.

Update via gradient descent: $\theta \leftarrow \theta - \alpha \nabla_\theta \ell$

Additional Resources

Core references:

- CS231N course notes on backpropagation
- Lecture 4 slides: Neural Networks and Backpropagation
- Justin Johnson's notes on matrix gradient calculations

Going deeper:

- Erik Learned-Miller's notes on vector/matrix/tensor derivatives
- The Matrix Cookbook
- "Why Momentum Really Works" (Distill), great interactive visualizations
- Gregory Gundersen's notes on the reparameterization trick

Appendix

Batched Linear Gradients

Consider the equation $Y = XW$ where $Y \in \mathbb{R}^{N \times D_y}$, $X \in \mathbb{R}^{N \times D_x}$, and $W \in \mathbb{R}^{D_x \times D_y}$. On the forward pass, the matrix multiplication can be rewritten as

$$Y_{ij} = \sum_{k=1}^{D_x} X_{ik} W_{kj}$$

Given this formula, how do we calculate the gradients $\frac{\partial Y}{\partial W}$ and $\frac{\partial Y}{\partial X}$? What if we add a loss function $\ell(Y) : \mathbb{R}^{N \times D_y} \to \mathbb{R}$? How do we calculate $\frac{\partial \ell(Y)}{\partial W}$ and $\frac{\partial \ell(Y)}{\partial X}$?

Appendix

Batched Linear Gradients

$$Y_{ij} = \sum_{k=1}^{D_x} X_{ik} W_{kj}$$

First, let us focus on the gradient with respect to one element in the weight matrix $\frac{\partial Y}{\partial W_{kj}}$ which should have the same shape as $Y \in \mathbb{R}^{N \times D_y}$.

$$\frac{\partial Y}{\partial W_{kj}} = \begin{bmatrix} 0 & \cdots & \frac{\partial Y_{1j}}{\partial W_{kj}} & \cdots & 0 \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ 0 & \cdots & \frac{\partial Y_{Nj}}{\partial W_{kj}} & \cdots & 0 \end{bmatrix} = \begin{bmatrix} 0 & \cdots & X_{1k} & \cdots & 0 \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ 0 & \cdots & X_{Nk} & \cdots & 0 \end{bmatrix}$$

Appendix

Batched Linear Gradients

$$Y_{ij} = \sum_{k=1}^{D_x} X_{ik} W_{kj}$$

The loss gradient $\frac{\partial \ell(Y)}{\partial Y}$ also has shape $\mathbb{R}^{N \times D_y}$ since $\ell(Y)$ is a scalar and can be written as

$$\frac{\partial \ell(Y)}{\partial Y} = \begin{bmatrix} \frac{\partial \ell(Y)}{\partial Y_{11}} & \cdots & \frac{\partial \ell(Y)}{\partial Y_{1D_y}} \\ \vdots & \ddots & \vdots \\ \frac{\partial \ell(Y)}{\partial Y_{N1}} & \cdots & \frac{\partial \ell(Y)}{\partial Y_{ND_y}} \end{bmatrix}$$

For a specific W_{kj}, we have

$$\frac{\partial \ell(Y)}{\partial W_{kj}} = \sum_{ij} \frac{\partial \ell(Y)}{\partial Y_{ij}} \cdot \frac{\partial Y_{ij}}{\partial W_{kj}} = \sum_{i=1}^{N} X_{ik} \cdot \frac{\partial \ell(Y)}{\partial Y_{ij}} = \left(X^\top \frac{\partial \ell(Y)}{\partial Y} \right)_{kj}$$

Thus, $\frac{\partial \ell(Y)}{\partial W} = X^\top \frac{\partial \ell(Y)}{\partial Y}$. Intuitively, this makes sense since the gradients of each weight should be aggregated across the batch of inputs.

Appendix

Batched Linear Gradients

$$Y_{ij} = \sum_{k=1}^{D_x} X_{ik} W_{kj}$$

Similarly, for $\frac{\partial Y}{\partial X_{ik}} \in \mathbb{R}^{N \times D_y}$,

$$\frac{\partial Y}{\partial X_{ik}} = \begin{bmatrix} 0 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ \frac{\partial Y_{i1}}{\partial X_{ik}} & \cdots & \frac{\partial Y_{iD_y}}{\partial X_{ik}} \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 0 \end{bmatrix} = \begin{bmatrix} 0 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ W_{k1} & \cdots & W_{kD_y} \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 0 \end{bmatrix}$$

which gives us

$$\frac{\partial \ell(Y)}{\partial X_{ik}} = \sum_{ij} \frac{\partial \ell(Y)}{\partial Y_{ij}} \cdot \frac{\partial Y_{ij}}{\partial X_{ik}} = \sum_{j=1}^{D_y} W_{kj} \cdot \frac{\partial \ell(Y)}{\partial Y_{ij}} = \left(\frac{\partial \ell(Y)}{\partial Y} W^\top \right)_{ik}$$

Thus, $\frac{\partial \ell(Y)}{\partial X} = \frac{\partial \ell(Y)}{\partial Y} W^\top$. This indicates that the gradient of the k-th component of a single input X_i depends on the corresponding row W_k in the weight matrix.