CIS3333: Mathematics of Machine Learning Fall 2025

Lecture: Multivariate Calculus II

Date: November 3rd, 2025 *Author: Surbhi Goel*

Last lecture ended with the gradient $\nabla f(x) \in \mathbb{R}^n$ for scalar-valued functions $f : \mathbb{R}^n \to \mathbb{R}$. The key insight was that the gradient provides the best linear approximation:

$$f(x_0 + h) = f(x_0) + \nabla f(x_0)^\top h + o(\|h\|)$$

This says: near x_0, the nonlinear function f behaves like a linear function. Specifically, consider the linear matrix $L_{x_0} : \mathbb{R}^n \to \mathbb{R}$ defined by

$$L_{x_0}(h) = \nabla f(x_0)^\top h.$$

Since $f(x_0 + h) - f(x_0) \approx \nabla f(x_0)^\top h$, L_{x_0} can be thought of as a linear function that transforms a small change in input $h \in \mathbb{R}^n$ into the corresponding change in output (a scalar).

A natural question. What if our function returns a *vector* instead of a scalar? For example:

- A neural network layer $f : \mathbb{R}^d \to \mathbb{R}^h$ that transforms an input to hidden activations
- A transformation $f : \mathbb{R}^2 \to \mathbb{R}^2$ that rotates and scales points in the plane
- Any function $f : \mathbb{R}^n \to \mathbb{R}^m$ where the output is multi-dimensional

For such vector-valued functions, we need a linear map $L_{x_0} : \mathbb{R}^n \to \mathbb{R}^m$ that captures how the function changes near a point x_0. The mathematical object that represents such a map is a *matrix*. This matrix is called the *Jacobian*, and it's the natural extension of gradients to vector-valued functions.

1 The Jacobian: Generalizing to Vector Functions

For a function $f : \mathbb{R}^n \to \mathbb{R}^m$ to be differentiable at x_0, there exists a matrix $J \in \mathbb{R}^{m \times n}$ such that:

$$f(x_0 + h) = f(x_0) + Jh + o(\|h\|)$$

This matrix J is the *Jacobian* of f at x_0, denoted $J_f(x_0)$. Locally at x_0, f acts like the linear map that maps small input changes h to output changes $J_f(x_0)h$.

Computing the Jacobian. How do we find the entries of this matrix? Since f outputs a vector, write $f(x) = \begin{bmatrix} f_1(x) \\ \vdots \\ f_m(x) \end{bmatrix}$ where each $f_i : \mathbb{R}^n \to \mathbb{R}$ is a scalar function. Each component has its own

gradient $\nabla f_i(x) \in \mathbb{R}^n$. The Jacobian stacks these gradient transposes as rows:

$$J_f(x) = \begin{bmatrix} (\nabla f_1(x))^\top \\ \vdots \\ (\nabla f_m(x))^\top \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix} \in \mathbb{R}^{m \times n}$$

When $m = 1$ (scalar output), $J_f(x) = (\nabla f(x))^\top$ is a $1 \times n$ row vector, recovering our gradient from last lecture.

Example. Consider $f : \mathbb{R}^2 \to \mathbb{R}^2$ defined by $f(x_1, x_2) = \begin{bmatrix} x_1^2 + x_2 \\ x_1 x_2 \end{bmatrix}$. Then:

$$J_f(x_1, x_2) = \begin{bmatrix} 2x_1 & 1 \\ x_2 & x_1 \end{bmatrix}$$

The Chain Rule: Composition of Linear Approximations. Recall from last lecture that the chain rule for scalar functions came from composing linear approximations. The same principle extends to vector functions.

Theorem 1 (Multivariate Chain Rule)**.** *Let $f : \mathbb{R}^n \to \mathbb{R}^k$ and $g : \mathbb{R}^k \to \mathbb{R}^m$ be differentiable. Their composition $h = g \circ f : \mathbb{R}^n \to \mathbb{R}^m$ is differentiable with Jacobian:*

$$J_h(x) = J_g(f(x)) \cdot J_f(x)$$

Interpretation. When we compose functions, we compose their linear approximations (Jacobians). If f transforms input changes by J_f and g transforms changes by J_g, then the composite transformation is $J_g \cdot J_f$—we apply J_f first, then J_g. The dimensions confirm this: $J_g(f(x))$ is $n \times m$ and $J_f(x)$ is $k \times n$, giving $k \times m$ as needed. The proof follows from substituting $f(x + k) = f(x) + J_f(x) \cdot k + o(\|k\|)$ into $g(y + \ell) = g(y) + J_g(y) \cdot \ell + o(\|\ell\|)$.

2 Application: Automatic Differentiation

To minimize a loss function using gradient descent, we need to compute $\nabla_\theta L(\theta)$. For complex models (compositions of many operations), deriving gradient formulas by hand is tedious and error-prone. **Automatic differentiation** (autodiff) uses the chain rule to compute gradients automatically.

The specific algorithm we'll study is called **backpropagation** (or reverse-mode autodiff), which efficiently computes gradients for scalar losses with many parameters. We'll use the terms "autodiff" and "backpropagation" interchangeably, as backpropagation is the dominant form of autodiff in machine learning.

2.1 Example: Least Squares

Recall from Lecture 13 that the least squares problem minimizes the empirical risk:

$$\hat{R}(\theta) = \frac{1}{N}\|X\theta - Y\|_2^2$$

where $X \in \mathbb{R}^{N \times D}$ is the data matrix, $Y \in \mathbb{R}^N$ are targets, and $\theta \in \mathbb{R}^D$ are parameters. We derived the closed-form solution $\hat{\theta} = (X^\top X)^{-1} X^\top Y$ using projections. But what if we want to use gradient descent instead? We need $\nabla_\theta L$.

The key idea: decompose the loss into simple operations, each with an easy-to-compute derivative:

$$\underset{\text{parameters } \theta}{\theta \in \mathbb{R}^D} \xrightarrow{f} \underset{\text{residual}}{z \in \mathbb{R}^N} \xrightarrow{g} \underset{\text{sum of squares}}{s \in \mathbb{R}} \xrightarrow{h} \underset{\text{empirical risk}}{\hat{R} \in \mathbb{R}}$$

The three operations are: (1) $f(\theta) = X\theta - Y$ (residual vector), (2) $g(z) = z^\top z$ (sum of squared residuals), and (3) $h(s) = \frac{1}{N}s$ (scaling by $1/N$).

Forward pass. Starting from θ, we compute each operation in sequence:

$$\begin{aligned} z &= f(\theta) = X\theta - Y \in \mathbb{R}^N \\ s &= g(z) = z^\top z \in \mathbb{R} \\ \hat{R} &= h(s) = \frac{1}{N}s \in \mathbb{R} \end{aligned}$$

This gives us the empirical risk value. To optimize, we need $\nabla_\theta \hat{R}$.

Backward pass. By the chain rule, we compute the Jacobian (a row vector):

$$J_{\hat{R}}(\theta) = J_h(s) \cdot J_g(z) \cdot J_f(\theta)$$

Each component has a simple Jacobian:

- $J_h(s) = \frac{dh}{ds} = \frac{1}{N}$ (scalar)
- $J_g(z) = 2z^\top \in \mathbb{R}^{1 \times N}$ (row vector)
- $J_f(\theta) = X \in \mathbb{R}^{N \times D}$ (matrix)

Multiplying the Jacobians from right to left (chaining VJPs):

$$\begin{aligned} J_{\hat{R}}(\theta) &= J_h(s) \cdot J_g(z) \cdot J_f(\theta) \\ &= \frac{1}{N} \cdot 2z^\top \cdot X \\ &= \frac{2}{N} z^\top X \\ &= \frac{2}{N}(X\theta - Y)^\top X \in \mathbb{R}^{1 \times D} \end{aligned}$$

This is the Jacobian—a $1 \times D$ row vector. The gradient is its transpose:

$$\nabla_\theta \hat{R} = J_{\hat{R}}(\theta)^\top = \frac{2}{N} X^\top (X\theta - Y) \in \mathbb{R}^D$$

Key observation: We never formed full Jacobian matrices. Each multiplication was cheap:

- $\frac{1}{N} \cdot 2z^\top = \frac{2}{N} z^\top$ (scalar multiplication)
- $\frac{2}{N} z^\top \cdot X$ (row vector times matrix, $O(ND)$ cost)

This is a **vector-Jacobian product (VJP)**: multiply a row vector by a Jacobian to get another row vector. Backpropagation chains VJPs rather than materializing full Jacobian matrices.

2.2 How Backpropagation Generalizes This

The least squares example reveals the pattern autodiff libraries use for *any* differentiable function:

Step 1: Build a computational graph. Represent the function as a sequence of primitive operations:

$$\theta \xrightarrow{f_1} v_1 \xrightarrow{f_2} v_2 \xrightarrow{f_3} \cdots \xrightarrow{f_K} \hat{R}$$

Each operation f_i is simple (matrix multiplication, addition, squaring, etc.) with a known derivative.

Step 2: Forward pass. Compute the function value by evaluating operations left-to-right, storing intermediate values $v_1, v_2, \ldots, v_K = \hat{R}$.

Step 3: Backward pass (backpropagation). Compute the Jacobian by applying the chain rule right-to-left:

1. Start with $J_{\hat{R}}(\hat{R}) = 1$ (Jacobian of $\hat{R}$ with respect to itself)
2. For each operation $f_{i+1} : v_i \mapsto v_{i+1}$ going backward, apply the chain rule:

   $$J_{\hat{R}}(v_i) = J_{\hat{R}}(v_{i+1}) \cdot J_{f_{i+1}}(v_i)$$

   where $J_{f_{i+1}}(v_i)$ is the Jacobian of operation f_{i+1}. This is a vector–Jacobian product: a row vector times a matrix.
3. Continue until we reach the parameters: $J_{\hat{R}}(\theta)$
4. Transpose to get the gradient: $\nabla_\theta \hat{R} = J_{\hat{R}}(\theta)^\top$

If a node v_i feeds into multiple operations (has multiple parents), sum the contributions from each outgoing edge:

$$J_{\hat{R}}(v_i) = \sum_{j:\, \text{parent}(v_i)} J_{\hat{R}}(v_j)\, J_{f_j}(v_i).$$

Forward mode. When the input dimension is small, propagate Jacobians with respect to the inputs. Define $J_{v_1}(\theta) := J_{f_1}(\theta)$ and update left-to-right

$$J_{v_{i+1}}(\theta) = J_{f_{i+1}}(v_i)\, J_{v_i}(\theta).$$

In a simple chain this yields $J_{(g \circ \cdots \circ f)}(\theta)$ directly. Reverse mode (backprop) is efficient for scalar $\hat{R}$ with many parameters; forward mode is efficient when there are few inputs and many outputs.

Why backpropagation works. This three-step process (build graph, forward pass, backward pass) has key advantages:

- **Modularity:** Each operation only needs to know its own Jacobian
- **Automation:** Libraries (PyTorch, JAX) implement this automatically—you write the forward pass, they compute gradients
- **Efficiency:** Chaining VJPs (row vector $\times$ matrix) is much cheaper than multiplying Jacobian matrices together. For a chain $\theta \xrightarrow{f_1} \cdots \xrightarrow{f_K} \hat{R}$, computing $((1 \cdot J_{f_K}) \cdot J_{f_{K-1}}) \cdots$ keeps the result as a row vector at each step, costing $O(\text{dimension})$ per step. Materializing all Jacobians and multiplying them would cost $O(\text{dimension}^2)$ per multiplication.

This is why backpropagation scales to models with millions of parameters! (For more details, see Davis, STAT 4830 notes.)