Day 23: Vector-Jacobian Products: The Computational Engine of Backpropagation

How Efficient Linear Algebra Operations Power Modern Deep Learning

1. The Computational Challenge of Jacobians

In deep learning, we often need to compute derivatives of vector-valued functions $f : \mathbb{R}^n \to \mathbb{R}^m$. The full Jacobian matrix:

$$J_f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

can be enormous for modern neural networks with millions of parameters. However, in practice, we rarely need the full Jacobian—we only need its product with certain vectors.

2. Vector-Jacobian Product (VJP) Definition

For a function $f : \mathbb{R}^n \to \mathbb{R}^m$ and a vector $\mathbf{v} \in \mathbb{R}^m$, the Vector-Jacobian Product (VJP) is defined as:

$$\text{VJP}(f, \mathbf{x}, \mathbf{v}) = \mathbf{v}^\top J_f(\mathbf{x})$$

This results in a row vector in $\mathbb{R}^{1\times n}$ that represents how changes in the input $\mathbf{x}$ would affect the output, weighted by the vector $\mathbf{v}$.

3. Why VJPs Are Fundamental to Backpropagation

In backpropagation, we need to compute the gradient of a scalar loss function L with respect to all parameters. For a composition of functions:

$$L = \ell(f(\mathbf{x}))$$

the chain rule gives us:

$$\frac{\partial L}{\partial \mathbf{x}} = \frac{\partial \ell}{\partial f} \cdot \frac{\partial f}{\partial \mathbf{x}} = \mathbf{v}^\top J_f(\mathbf{x})$$

where $\mathbf{v} = \frac{\partial \ell}{\partial f}$ is the gradient of the loss with respect to the output of f.

This is exactly a VJP! Backpropagation is essentially a sequence of VJPs applied layer by layer.

4. A Detailed Example: Step-by-Step VJP Computation

Let's consider the function:

$$f(x_1, x_2) = \begin{bmatrix} x_1^2 + x_2 \\ \sin(x_1) \\ x_1 x_2 \end{bmatrix}$$

Step 1: Compute the Jacobian

$$J_f(x_1, x_2) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} \\ \frac{\partial f_3}{\partial x_1} & \frac{\partial f_3}{\partial x_2} \end{bmatrix} = \begin{bmatrix} 2x_1 & 1 \\ \cos(x_1) & 0 \\ x_2 & x_1 \end{bmatrix}$$

Step 2: Choose a Specific Point and Vector

Let's evaluate at $\mathbf{x} = (1, 2)$ with vector $\mathbf{v} = (1, -1, 2)$.

Step 3: Compute the VJP

First, compute the Jacobian at the point:

$$J_f(1, 2) = \begin{bmatrix} 2(1) & 1 \\ \cos(1) & 0 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 1 \\ \cos(1) & 0 \\ 2 & 1 \end{bmatrix}$$

Now compute the VJP:

$$\mathbf{v}^\top J_f(1, 2) = [1, -1, 2] \begin{bmatrix} 2 & 1 \\ \cos(1) & 0 \\ 2 & 1 \end{bmatrix}$$

$$= [1 \cdot 2 + (-1) \cdot \cos(1) + 2 \cdot 2, \quad 1 \cdot 1 + (-1) \cdot 0 + 2 \cdot 1]$$

$$= [2 - \cos(1) + 4, \quad 1 + 2] = [6 - \cos(1),\ 3]$$

5. Comparison: VJP vs. JVP (Forward Mode)

It's important to distinguish between two fundamental operations in automatic differentiation:

VJP vs. JVP

- **Vector-Jacobian Product (VJP):** $\mathbf{v}^\top J_f(\mathbf{x})$ - used in **reverse-mode** autodiff (backpropagation)
- **Jacobian-Vector Product (JVP):** $J_f(\mathbf{x})\mathbf{u}$ - used in **forward-mode** autodiff

VJPs are more efficient when the output dimension is smaller than the input dimension (common in neural networks where we have a scalar loss), while JVPs are more efficient when the input dimension is smaller than the output dimension.

6. Implementation in Deep Learning Frameworks

Modern deep learning frameworks like PyTorch, TensorFlow, and JAX use VJPs extensively:

How Frameworks Implement VJPs

- **Computational graphs:** Frameworks build a graph of operations during the forward pass
- **Local VJPs:** Each operation defines how to compute its local VJP
- **Chain rule:** Frameworks chain these local VJPs to compute the overall gradient
- **Memory efficiency:** VJPs avoid storing the full Jacobian, saving memory

For example, in PyTorch, the backward() method essentially computes a VJP where the vector is the gradient of the loss with respect to the output.

7. The Efficiency Advantage of VJPs

The computational advantage of VJPs becomes clear when we consider the complexity:

- Computing the full Jacobian: $O(mn)$ time and space
- Computing a VJP: $O(n)$ time (same as function evaluation) and $O(1)$ space for the result

For neural networks with millions of parameters but a scalar loss ($m = 1$), this is a massive savings.

8. Beyond Backpropagation: Other Applications of VJPs

Additional Applications

- **Implicit differentiation:** Solving for gradients in models with equilibrium constraints
- **Meta-learning:** Differentiating through optimization processes
- **Physics-informed neural networks:** Enforcing physical constraints through derivatives
- **Adversarial robustness:** Understanding how input perturbations affect outputs
- **Neural ODEs:** Differentiating through differential equation solvers

9. Historical Context and Modern Developments

The concept of VJPs dates back to the 1960s with the development of reverse-mode automatic differentiation. Key developments include:

- **1960s:** First descriptions of reverse-mode automatic differentiation
- **1980s:** Application to neural networks through backpropagation
- **2010s:** Widespread adoption in deep learning frameworks
- **Recent:** Advanced applications in differentiable programming and scientific machine learning

10. Exercises for Understanding

1. For $f(x, y) = \begin{bmatrix} x^2 y \\ e^x + y \\ \sin(xy) \end{bmatrix}$, compute the VJP with $\mathbf{v} = (1, 2, -1)$ at the point $(1, 1)$
2. Explain why VJPs are more efficient than full Jacobian computation for neural networks with a scalar output
3. Implement a simple VJP for a two-layer neural network in pseudocode
4. Compare the memory requirements of computing a full Jacobian versus a VJP for a function $f : \mathbb{R}^{1000} \to \mathbb{R}^{10}$

Key Takeaway

Vector-Jacobian Products are the computational foundation of backpropagation and modern deep learning. By efficiently computing the product of a vector with a Jacobian matrix without explicitly constructing the full Jacobian, VJPs enable the training of neural networks with millions or even billions of parameters. Understanding VJPs is essential for working with automatic differentiation frameworks, implementing custom layers, and developing new deep learning architectures.