Matrix Calculus for Machine Learning and Beyond

Guillaume Collin
Mentor: Miguel Ayala

DRP Winter 2024, McGill University Mathematics & Statistics

1 Introduction

This paper explores mathematical techniques essential for the optimization of machine learning algorithms, employing the Julia programming language as our computational tool. As modern applications like machine learning and large-scale optimization increasingly demand a deeper understanding of calculus beyond traditional univariate and vector realms, our focus extends to the sophisticated landscape of "matrix calculus" and calculus on arbitrary vector spaces.

Central to our discussion is the reinterpretation of derivatives as linear operators and the linear approximation on arbitrary vector spaces. These concepts extend beyond conventional gradients and Jacobians, providing a fresh perspective on how matrices can be viewed as entities enabling the computation of derivatives of essential matrix factorizations and complex operations with clarity.

Our discussion traverses through key optimization methods pivotal for machine learning tasks, including forward-mode and reverse-mode differentiation, also called backpropagation, which are all integral components in gradient-based optimization for machine learning models.

We also examine automatic differentiation methods such as forward-mode automatic differentiation via dual numbers, enabling efficient gradient computation for complex neural networks. Additionally, we explore sensitivity analysis for differential equations and derivatives of random functions, highlighting their relevance in machine learning model refinement.

This report summarizes information from the lectures of MIT's Matrix Calculus for Machine Learning and Beyond course, guided by the expertise of Professors Steven G. Johnson and Alan Edelman, which we completed in its entirety.

2 Preliminaries

2.1. Matrices and Vectors

When extending our discussions to matrices and vectors, certain principles hold true, albeit with variations.

- For matrices A and B, the derivative of their product AB is expressed as:

$$d(AB) = (dA)B + A(dB)$$

- However, in the realm of vectors, the product rule takes on a different form. For a vector x, the derivative of $x^T x$ is given by:

$$d(x^T x) = (dx^T)x + x^T(dx)$$

  It's worth noting that when dealing with vector dot products, as in $x^T x$, commutativity simplifies the expression:

$$d(x^T x) = (2x)^T dx$$

  However, for matrices, generally the products do not commute:

$$AB \neq BA$$

2.2. Product Rule for Vectors and Matrices

In understanding the product rule for vectors and matrices, transposes play a crucial role in maintaining the integrity of the operations.

Examples:

1. $d(u^T v) = du^T v + u^T dv$ - It's important to recognize that $du^T v = v^T du$ due to the commutativity of dot products.
2. $d(uv^T) = duv^T + udv^T$

3 Derivatives as linear operators

The key to generalizing derivatives is to realize that they are linear operators, relating small changes in a function input to small changes in the output.

3.1. Linearization

For a function $f(x)$, if the input is changed by a small amount δx, the change in the output can be approximated by a linear expression in δx:

$$f(x + \delta x) \approx f(x) + f'(x)\delta x + o(\delta x)$$

Here, $o(\delta x)$ represents higher-order terms that become negligible as δx becomes very small. This approximation can be written in terms of the change in f:

$$\delta f = f(x + \delta x) - f(x) \approx f'(x)\delta x + o(\delta x)$$

3.2. Differential Notation

In differential notation, we express the change in f in terms of dx, where dx is an arbitrarily small change in x:

$$df = f(x + dx) - f(x) = f'(x)dx$$

Here, we implicitly drop any higher-order terms. The relationship can be summarized as:

Thus,

$$df = f'(x)dx$$

3.3. Linearity

A linear operator L satisfies the property of linearity:

$$L[v_1 + v_2] = L[v_1] + L[v_2]$$

3.4. Gradient and Scalar Functions

For a scalar function f with a vector input $x \in \mathbb{R}^m$ (an m-component column vector), the differential df can be expressed as”

$$df = f(x + dx) - f(x) = f'(x)dx$$

In this context, $f'(x)$ is a linear operator, and the gradient ∇f is the vector that, when dotted with dx, gives the differential:

$$df = (\nabla f) \cdot dx$$

Example

Consider the quadratic form $f(x) = x^T Ax$, where A is a matrix and x is a vector. The differential df is calculated as follows:

$$\begin{aligned} df &= f(x + dx) - f(x) \\ &= (x + dx)^T A(x + dx) - x^T Ax \\ &= x^T Ax + dx^T Ax + x^T Adx + dx^T Adx - x^T Ax \\ &= dx^T Ax + x^T Adx + dx^T Adx \end{aligned}$$

Ignoring the higher-order term $dx^T Adx$, we get:

$$df \approx dx^T Ax + x^T Adx$$

Multivariable Functions and Derivatives

If the input x is a vector and the function f is a scalar, the derivative is a linear operator acting on a vector. Here, dx is an arbitrary change in the input. The linear operator on a vector that gives a scalar is a row vector:

$$df = f(x + dx) - f(x) = f'(x)dx = (\nabla f) \cdot dx$$

Here, $f'(x)$ is the transpose of the gradient of f. In this context, $f'(x)$ is the row vector. The size of the Jacobian matrix is $m \times n$ for a function $f : \mathbb{R}^n \to \mathbb{R}^m$. The rows are the outputs (df) and the columns are the inputs (dx).

Important Takeaway

We think about $f'(x)$ as a linear operator acting on the change of x (dx), and we can write it as a Jacobian matrix if we want.

Example 1

$$\begin{aligned} df &= f(x + dx) - f(x) \\ &= Adx \\ &= Ax + Adx - Ax \\ &= f'(x)dx \end{aligned}$$

Here, $f'(x) = A$, where A is just the Jacobian. Note that the Jacobian doesn't depend on x; it is the same A everywhere in the n-dimensional space.

Example 2

Consider $f(x) = Ax$, a matrix function:

$$df = d(Ax) + Adx = Adx \quad (\text{since } dA = \mathbf{0}_{m\times n})$$

Associativity Matters Practically

Associativity plays a crucial role in practical applications, particularly in the field of machine learning. While we generally consider functions of the form $f : \mathbb{R}^n \to \mathbb{R}^m$, in machine learning, it is common to encounter scenarios where n is extremely large and $m = 1$. Given this context, the efficiency gains from using backpropagation (reverse mode) are significantly influenced by the associative property of operations. This highlights the practical importance of associativity in optimizing computational performance.

Associativity in Function Composition

Consider three functions f, a, b, c where:

$$f(x) = a(b(c(x)))$$

with $f, a \in \mathbb{R}^m$, $b \in \mathbb{R}^q$, $c \in \mathbb{R}^p$, and $x \in \mathbb{R}^n$.

The derivative f' of f with respect to x can be computed in two distinct ways due to the associative property:

1. **Reverse Mode (left to right):**
$$f'(x) = (a'b')c'$$
2. **Forward Mode (right to left):**
$$f'(x) = a'(b'c')$$

Here, a' is an $m \times q$ matrix, b' is a $q \times p$ matrix, and c' is a $p \times n$ matrix. It matters because the cost of multiplying matrices depends mostly on their shape.

Cost of Matrix Multiplication

Matrix multiplication cost is highly dependent on the shape of the matrices involved. For two matrices of dimensions $(m \times q)$ and $(q \times p)$, the cost of their multiplication is proportional to:

$$\text{Cost} = m \times p \text{ dot products of length } q$$

Each dot product involves q multiplications and $(q - 1)$ additions, leading to approximately q scalar operations per dot product. Hence, the total computational cost is:

$$\text{Total Cost} \approx mpq \text{ scalar operations}$$

In computer science notation, this complexity is often represented as $\mathcal{O}(mpq)$.

Example

For matrices of dimensions $(1 \times m)$ and $(m \times n)$, the cost is:

$$\mathcal{O}(m^2)$$

Importance of Chain Rule Order

In large-scale optimization problems, such as those in machine learning, the order in which the chain rule is applied can drastically affect computational efficiency. These problems typically involve a large number of inputs (n is large) and a single output ($m = 1$).

Consider the following scenario:

- **Optimization parameters**: Numerous
- **"Loss" or "objective" function**: Single output
- **Intermediate values**: Many (denoted by $q, p \sim n$)

In neural networks, for instance, you might have a billion inputs but only one output to optimize. Here, the derivative f' can be computed as:

1. **Reverse Mode (Backpropagation)**:

   $$f' = (\nabla f)^T = a'b'c' = (a'b')c'$$

   - Cost: $\mathcal{O}(n^2)$

2. **Forward Mode**:

   $$f' = a'(b'c')$$

   - Cost: $\mathcal{O}(n^3)$

In this context, f' is a $1 \times n$ vector, a' is a $1 \times n$ matrix, b' is a $n \times n$ matrix, and c' is a $n \times n$ matrix.

The choice of reverse mode (backpropagation) or forward mode significantly impacts computational efficiency. In reverse mode, you perform a vector-matrix multiplication, resulting in $\mathcal{O}(n^2)$ operations. In forward mode, you perform matrix-matrix multiplications, resulting in $\mathcal{O}(n^3)$ operations.

Practical Implications in Machine Learning

In machine learning, the gradient indicates the direction of steepest ascent (or descent, when considering the negative gradient). This gradient information is crucial for adjusting parameters to optimize the system. Reverse mode, known as backpropagation, is essential for efficiently computing derivatives with respect to a large number of variables. This process involves multiplying Jacobians from left to right, highlighting the practical importance of associativity in these calculations.

In summary, the associative property of operations and the order of applying the chain rule are critical for computational efficiency, especially in the context of large-scale optimization problems like those encountered in machine learning. Understanding and leveraging these properties can lead to significant performance improvements in practical applications.

Vectorization of Matrix Functions

We can view matrix functions (with matrix inputs and outputs) as "ordinary" multivariable calculus by "vectorizing" a matrix into a column vector. Inputs and outputs in more general vector spaces can also be considered. Recall: A vector space is abstractly anything you can add and subtract or multiply by a scalar.

Example

Consider the function $f(A) = A^{-1}$.

We know that:

$$A^{-1}A = I$$

Taking the differential, we get:

$$d(A^{-1}A) = d(I) = 0$$

Applying the product rule:

$$d(A^{-1})A + A^{-1}dA = 0$$

Solving for $d(A^{-1})$:

$$d(A^{-1})A = -A^{-1}dA$$

Thus, we have:

$$d(A^{-1}) = -A^{-1}dAA^{-1} = f'(A)[dA]$$

This shows how the derivative of the matrix inverse function can be derived using the concept of vectorization and differentials.

4 Automatic Differentiation (AutoDiff)

Automatic differentiation (AutoDiff) is a computational technique for evaluating the derivative of a function specified by a computer program. It leverages the fact that any computer program that performs numerical computations, no matter how complex, can be decomposed into a sequence of elementary arithmetic operations and functions. By applying the chain rule systematically to these operations, AutoDiff can compute derivatives accurately and efficiently.

4.1 Jacobians in Julia Using Automatic Differentiation

In Julia, automatic differentiation can be utilized to compute Jacobians, which are matrices of all first-order partial derivatives of a vector-valued function. Various packages, such as `ForwardDiff.jl` and `ReverseDiff.jl`, facilitate this process.

4.1.1 Linear Transformation Jacobian

A linear transformation can be represented by a matrix A, and its Jacobian is simply the matrix A itself when differentiating with respect to the input vector. If $\mathbf{y} = A\mathbf{x}$, the Jacobian $\mathbf{J}$ is:

$$\mathbf{J} = \frac{\partial \mathbf{y}}{\partial \mathbf{x}} = A$$

4.1.2 Notations for Jacobians

There are two common notations for Jacobians:

1. $\mathbf{J}_{ij} = \frac{\partial y_i}{\partial x_j}$
2. $\mathbf{J} = \left(\frac{\partial \mathbf{y}}{\partial \mathbf{x}}\right)^T$

These notations differ by transposition, but both convey the same information: the rate of change of each output component with respect to each input component.

5 The LU Decomposition

The LU decomposition is a method of decomposing a square matrix A into the product of a lower triangular matrix L and an upper triangular matrix U. Mathematically:

$$A = LU$$

For an $n \times n$ matrix A, L contains $n(n-1)/2$ elements, and U contains $n(n+1)/2$ elements. Thus, the function $f : A \to LU$ maps from $\mathbb{R}^{n^2}$ to $\mathbb{R}^{n^2}$, and its Jacobian will be an $n^2 \times n^2$ matrix.

5.1 Jacobian of LU Decomposition

The Jacobian of the LU decomposition with respect to A involves understanding how perturbations in A propagate to perturbations in L and U. Using the product rule:

$$d(LU) = dL \cdot U + L \cdot dU$$

This can be rewritten using Kronecker products:

$$(I \otimes L + U^T \otimes I) \begin{bmatrix} dL \\ dU \end{bmatrix}$$

For lower and upper triangular perturbations, dL and dU have specific forms:

$$dL = \begin{bmatrix} 0 & 0 \\ dL_{21} & 0 \end{bmatrix}, \quad dU = \begin{bmatrix} dU_{11} & dU_{12} \\ 0 & dU_{22} \end{bmatrix}$$

6 Eigenvalues of a 2x2 Matrix

Computing the eigenvalues of a 2×2 matrix is a transformation from $\mathbb{R}^2$ to $\mathbb{R}^2$. The Jacobian of this transformation can be derived as well, relating changes in the matrix entries to changes in the eigenvalues.

The eigendecomposition changes coordinates in a manner similar to converting from Cartesian to polar coordinates, simplifying the mathematical analysis.

7 The 2x2 Symmetric Eigenproblem

A 2×2 symmetric matrix can be represented as a vector in a 3-dimensional space, with three parameters: the two diagonal elements and the off-diagonal element.

8 Finite Difference Approximations

Finite difference approximations are crucial for numerical differentiation, especially when dealing with functions too complex for symbolic differentiation or when AutoDiff is not feasible.

8.1 Advantages and Limitations of AutoDiff

AutoDiff is precise and less error-prone than manual differentiation, which is particularly important in numerical optimization, root finding, and sensitivity analysis. However, AutoDiff can struggle with external library calls or languages not fully compatible with its framework. In such cases, manual differentiation or hybrid approaches combining manual and automatic differentiation may be necessary.

8.2 Forward and Backward Differentiation

Finite differences involve approximating derivatives by perturbing the input. The forward difference formula for a scalar function f at point x is:

$$f'(x) \approx \frac{f(x + \delta x) - f(x)}{\delta x}$$

Backward differentiation uses a perturbation in the negative direction. Both methods approximate the derivative but are not the same as forward or reverse mode automatic differentiation.

8.3 Example: Product Rule Verification

To verify a derived formula using finite differences, consider the product rule:

$$df = A\,dA + dA\,A$$

By introducing a small perturbation δA, the finite difference approximation can be compared with the analytical derivative:

$$A\,\delta A - \delta A\,A \neq 0$$

(they do not commute)

The error analysis shows the accuracy of the finite difference approximation relative to the exact derivative.

8.4 Norm and Error Analysis

The Frobenius norm can be used to measure the distance between the approximate and exact derivatives. Plotting the relative error against the norm of δA on a log-log scale reveals the trade-off between truncation error and roundoff error. The optimal choice of δx balances these errors, often around $\sqrt{\epsilon}$, where ϵ is the machine epsilon.

9 Accuracy of Finite Differences

Finite difference accuracy improves as δx decreases, up to the point where roundoff error dominates. The Taylor series expansion explains the first-order accuracy:

$$f(x + \delta x) \approx f(x) + f'(x)\delta x + o(\delta x^2)$$

Choosing δx appropriately minimizes truncation and roundoff errors, ensuring reliable numerical differentiation. The machine epsilon, approximately 2^{-52}, guides the choice of δx, balancing significant digits and computational precision.

10 Finite Differences and Error Analysis

Finite differences are a fundamental method for approximating derivatives. The simplest form of a finite difference approximation is the forward difference, given by:

$$f'(x) \approx \frac{f(x + \delta x) - f(x)}{\delta x}$$

However, this method is prone to roundoff errors, particularly when the step size, δx, is very small. Initially, as δx decreases, the error decreases linearly. But beyond a certain point, the roundoff error starts to dominate, causing the total error to increase. This behavior makes forward differences a first-order method, as the error is proportional to δx.

To improve the accuracy, higher-order techniques can be employed. One such technique is the central difference method:

$$f'(x) \approx \frac{f(x+\delta x) - f(x-\delta x)}{2\delta x} + O(\delta x^2)$$

The central difference method is more accurate because its error term decreases quadratically with δx. When plotted on a logarithmic scale, the quadratic error reduction appears linear. This makes the central difference method superior for finite difference approximations.

Further improvements can be made using higher-order differences by considering more points. For example, Richardson extrapolation involves starting with an initial δx, then iteratively reducing it (e.g., by dividing by 2) and fitting higher degree polynomials. This process helps in finding the optimal δx and polynomial degree for the best approximation.

11 Gradients in Higher Dimensions

When extending these concepts to functions with vector inputs (i.e., $x \in \mathbb{R}^n$), the formula for the finite difference approximation remains valid:

$$\delta f = f(x+\delta x) - f(x) = f'(x)\delta x + o(\|\delta x\|)$$

For a scalar-valued function f with vector input x, the gradient ∇f is the vector of partial derivatives. Computing this gradient requires n finite differences, one for each dimension. This can be computationally expensive, especially for functions that are themselves computationally intensive.

In higher dimensions, calculating finite differences for all directions can become impractical. Analytical derivatives are preferred, especially in applications like machine learning, where high-dimensional gradients are necessary. Analytical derivatives can provide more accuracy and efficiency, which is crucial when dealing with millions of dimensions.

12 Generalized Gradients via Inner Product

For scalar-valued functions $f(x)$ with vector inputs x in $\mathbb{R}^n$, the derivative can be expressed as:

$$df = f(x+dx) - f(x) = f'(x)[dx]$$

Here, $f'(x)$ acts as a linear operator, mapping the vector dx to a scalar. This operator is essentially a row vector, making the expression for the derivative:

$$df = (\nabla f)^T \cdot dx$$

To generalize this concept to any vector space V, we need a suitable inner product for V. The inner product (or dot product) is defined as follows for vectors $x, y \in V$:

$$\langle x, y \rangle \in \mathbb{R}$$

The inner product must satisfy three properties:

1. **Symmetry:** $\langle x, y \rangle = \langle y, x \rangle$
2. **Linearity:** $\langle x, ay + bz \rangle = a\langle x, y \rangle + b\langle x, z \rangle$ for scalars a and b
3. **Non-negativity:** $\langle x, x \rangle = \|x\|^2 \geq 0$

A vector space with an inner product is called a Hilbert space. For a scalar-valued function $f(x)$ where x is in a Hilbert space, the gradient ∇f must satisfy:

$$f'(x)[dx] \in \mathbb{R}$$

$$f'(x)[dx] = (\nabla f) \cdot dx$$

The gradient ∇f always has the same structure as x.

13 Examples of Inner Products

13.1 Euclidean Space ($\mathbb{R}^n$)

The standard inner product for vectors $x, y \in \mathbb{R}^n$ is:

$$x \cdot y = x^T y = \sum_{i=1}^{n} x_i y_i$$

13.2 Matrix Space ($\mathbb{R}^{m \times n}$)

For matrices $A, B \in \mathbb{R}^{m \times n}$, the Frobenius inner product is:

$$A \cdot B = \sum_{i,j} A_{ij} B_{ij} = \mathrm{tr}(A^T B)$$

This inner product induces the Frobenius norm:

$$\|A\| = \sqrt{\operatorname{tr}(A^T A)}$$

14 Gradient of Scalar Functions of Matrix Inputs

For scalar functions of matrix inputs, the gradient can be derived using the properties of the trace function. The trace function satisfies:

$$\operatorname{tr}(B) = \operatorname{tr}(B^T)$$

$$\operatorname{tr}(AB) = \operatorname{tr}(BA)$$

These properties can be used to manipulate and compute gradients involving matrix inputs. For instance, the gradient of a scalar function $f(A)$ with respect to a matrix A can be expressed using the trace function and the appropriate inner product. This allows for efficient computation of derivatives in more complex vector spaces.

15 Newton's Method: Nonlinear Equations via Linearization

Newton's method is a powerful tool for finding solutions to nonlinear equations by leveraging linear approximations. This method, named after Sir Isaac Newton, is rooted in the principle of linearizing a nonlinear function around an initial guess and iteratively refining this guess to converge to an accurate solution.

15.1 Solving $f(x) = 0$

To solve a nonlinear equation $f(x) = 0$ using Newton's method, we follow these steps:

1. **Linearize** $f(x + \delta x) \approx f(x) + f'(x)\delta x$:
    - Here, δx represents a small change in x.
    - The function $f(x)$ is approximated by its tangent line at x.
2. **Solve the Linear Equation**:

  - Set the linear approximation to zero: $f(x) + f'(x)\delta x = 0$.
  - Solve for δx: $\delta x = -\frac{f(x)}{f'(x)}$.

3. **Update** x:
   - Refine the guess for x by updating it: $x \leftarrow x - \frac{f(x)}{f'(x)}$.

These steps are iterated until x converges to a root of the function. Newton's method is particularly effective because it approximates the function by its tangent, making the problem linear and easier to solve at each step.

15.2 Multi-dimensional Newton's Method

In real-world applications, functions often have multiple variables, making the problem nonlinear and more complex. The multi-dimensional Newton's method extends the single-variable method to handle vector-valued functions.

Consider solving $f(x) = 0$, where $x \in \mathbb{R}^n$ and f maps $\mathbb{R}^n$ to $\mathbb{R}^n$:

1. **Linearize** $f(x + \delta x) \approx f(x) + J_f(x)\delta x$:
   - $J_f(x)$ is the Jacobian matrix of f at x.
2. **Solve the Linear Equation**:
   - Set the linear approximation to zero: $f(x) + J_f(x)\delta x = 0$.
   - Solve for δx: $\delta x = -J_f(x)^{-1} f(x)$.
3. **Update** x:
   - Refine the guess for x by updating it: $x \leftarrow x - J_f(x)^{-1} f(x)$.

This iterative process continues until convergence. The Jacobian matrix $J_f(x)$ plays a crucial role, as it generalizes the derivative for multi-dimensional functions. Each step requires solving a system of linear equations, making the method computationally intensive but highly efficient when close to the root, exhibiting quadratic convergence.

16 Other Applications of Derivatives: Optimization

Derivatives are not only crucial in solving equations but also in optimization problems. In nonlinear optimization, we aim to minimize (or maximize) a scalar function $f(x)$ where $x \in \mathbb{R}^n$.

16.1 Gradient Descent

One common technique is gradient descent, which iteratively moves x in the direction of the negative gradient of f:

- The gradient ∇f points in the direction of the steepest ascent. Thus, $-\nabla f$ points downhill, towards a local minimum.
- Gradient descent updates x by $x \leftarrow x - \alpha \nabla f(x)$, where α is the step size.

This method is especially effective in high-dimensional spaces, such as training neural networks, where the gradient computation via backpropagation is efficient. Backpropagation computes ∇f with a cost comparable to evaluating $f(x)$ once, making large-scale optimization practical.

16.2 Complications in Nonlinear Optimization

Optimization faces several challenges:

- **Step Size**: Choosing an appropriate step size α is critical. Line search and trust-region methods help in dynamically adjusting the step size to ensure convergence.
- **Constraints**: When optimizing subject to constraints $g_k(x) \leq 0$, gradients ∇g_k of the constraints are also needed.
- **Momentum and Conjugate Gradients**: To improve convergence in narrow valleys, methods like momentum terms and conjugate gradients are used, which incorporate information from previous steps.
- **Second Derivative Information**: Advanced methods like BFGS approximate the Hessian matrix (second derivatives) for faster convergence.

17 Adjoint Differentiation

Adjoint differentiation, or reverse-mode differentiation, is crucial for efficiently computing gradients in high-dimensional optimization problems. It is particularly useful when a scalar function f depends on a solution x of a system of equations parameterized by p.

Given $f(x(p))$ where $A(p)x = b$:

- The gradient df can be computed as $df = -(f'(x)A^{-1})dA \cdot x$.
- This involves solving the original system $Ax = b$ and an adjoint system $A^T v = f'(x)^T$.

The adjoint method efficiently computes gradients by solving two systems of equations, leveraging sparsity in $\partial A / \partial p_k$ for cheap dot products.

Norms and Derivatives

To define a gradient, an inner product is indispensable. When dealing with a scalar function's derivative, this derivative must be a linear function that accepts a vector and outputs a scalar. However, the necessity of an inner product extends to the definition of norms, which measure the "size" or "length" of vectors.

Norms

Given a vector space V, a norm $||v||$ for $v \in V$ is a mapping $V \to \mathbb{R}$ satisfying:

1. **Non-negativity**: $||v|| \geq 0$, and $||v|| = 0$ if and only if $v = 0$.
2. **Scaling**: $||av|| = |a| \cdot ||v||$ for any scalar $a \in \mathbb{R}$.
3. **Triangle Inequality**: $||u + v|| \leq ||u|| + ||v||$ for all $u, v \in V$.

An example of a norm derived from an inner product is $||u|| = \sqrt{u \cdot u}$, where $u \cdot u$ represents the inner product.

Derivatives and Norms

Derivatives necessitate norms for both input and output spaces. For a function f and a perturbation δx, the difference can be expressed as:

$$f(x + \delta x) - f(x) = f'(x)[\delta x] + o(\delta x)$$

where $o(\delta x)$ is any function such that $\lim_{\delta x \to 0} \frac{|o(\delta x)|}{|\delta x|} = 0$.

Commonly, norms are derived from inner products, making the inner product essential in defining what constitutes a "small" perturbation by mapping the vector length to a real number.

The Gradient of the Determinant

Theorem: Gradient of the Determinant

The gradient of the determinant of a matrix A is given by:

$$\nabla(\det A) = \text{cofactor}(A) = (\det A)A^{-T} = \text{adj}(A^T)$$

Derivative of the Determinant

For a small perturbation dA, the derivative of the determinant can be expressed as:

$$d(\det A) = \text{tr}((\det A)A^{-1}dA) = \text{tr}(\text{adj}(A)dA) = \text{tr}(\text{cofactor}(A)^T dA)$$

Adjugate and Cofactor

The adjugate matrix, denoted as adj(A), is defined as:

$$\text{adj}(A) = (\det A)A^{-1} = \text{cofactor}(A)^T$$

The cofactor matrix of A has its (i, j)-th entry as $(-1)^{i+j}$ times the determinant of the submatrix obtained by deleting the i-th row and j-th column of A.

Key Formulas at a Glance

$$\begin{aligned} A^{-1} &= \frac{\text{adj}(A)}{\det(A)} = \frac{\text{cofactor}(A)^T}{\det(A)} \\ \text{adj}(A) &= (\det A)A^{-1} = \text{cofactor}(A)^T \\ \text{cofactor}(A) &= \frac{A^{-T}}{\det(A)} = \text{adj}(A)^T \end{aligned}$$

Proof via Cofactor Expansion

A direct proof uses the cofactor expansion of the determinant:

$$\det(A) = A_{i1}C_{i1} + A_{i2}C_{i2} + \cdots + A_{in}C_{in}$$

Since the determinant is a linear function of any element with a slope equal to the cofactor, we obtain:

$$\frac{\partial \det(A)}{\partial A_{ij}} = C_{ij}$$

Thus, $\nabla(\det A) = C$.

Linearization Near the Identity

Consider linearization near the identity matrix I:

$$\det(I + dA) \approx \text{trace}(dA)$$

$$\det(A+A(A^{-1}dA)) = \det(A)\det(I+A^{-1}dA) = \det(A)\text{tr}(A^{-1}dA) = \text{tr}((\det A)A^{-1}dA)$$

Application: Logarithmic Derivative

The derivative of the logarithm of the determinant is given by:

$$d(\log(\det A)) = \det(A^{-1})d(\det A) = \text{tr}(A^{-1}dA)$$

This logarithmic derivative is particularly useful in applied mathematics, such as in Newton's method where $\frac{f(x)}{f'(x)}$ can be expressed as $\frac{1}{(\log f(x))'}$.

Dual Number Notation

Dual numbers, an extension of real numbers, are written in the form $a + b\epsilon$ where $\epsilon^2 = 0$. This concept can be compared to imaginary numbers where i satisfies $i^2 = -1$.

Operations with Dual Numbers

$$\begin{aligned}(a + b\epsilon) \pm (c + d\epsilon) &= (a \pm c) + (b \pm d)\epsilon \\ (a + b\epsilon) \cdot (c + d\epsilon) &= (ac) + (bc + ad)\epsilon \\ \frac{a + b\epsilon}{c + d\epsilon} &= \frac{a}{c} + \frac{bc - ad}{c^2}\epsilon\end{aligned}$$

Dual numbers are instrumental in automatic differentiation, simplifying the computation of derivatives by truncating higher-order terms.

18 Ordinary Differential Equations (ODEs)

An Ordinary Differential Equation (ODE) is an equation involving a function and its derivatives. ODEs are fundamental in describing various dynamical systems in physics, biology, economics, and engineering.

19 Initial Value Problem

An initial value problem for an ODE consists of finding a function $u(t)$ that satisfies the differential equation and the initial condition:

$$u(t_0) = u_0, \quad \frac{du}{dt}(t) = f(t, u(t), p),$$

where $u(t)$ is the unknown function, t is the independent variable, u_0 is the initial state, and p represents parameters.

20 Numerical Solvers: Euler's Method

Euler's method is the simplest numerical technique to approximate solutions of ODEs. It iteratively updates the solution using:

$$u_{n+1} = u_n + \Delta t f(t_n, u_n, p), \quad t_n = t_0 + n\Delta t,$$

where Δt is the time step. While not the most accurate, Euler's method is straightforward and often provides good results for small time steps.

21 Sensitivity Analysis of ODEs

Sensitivity analysis examines how variations in parameters affect the solution of the ODE. There are two main approaches to differentiate through an ODE:

21.1 Discrete Sensitivity Analysis

Automatic Differentiation (AD) is applied to the solver operations, yielding the exact gradient of the discretized approximation, often termed "discretize-then-differentiate."

21.2 Continuous Sensitivity Analysis

Custom differentiation rules are applied before discretizing, providing an approximation of the exact gradient, referred to as "differentiate-then-discretize."

21.3 Modes of Sensitivity Analysis

Both methods have two modes each (forward/tangent and reverse/adjoint):

- **Forward/Tangent Mode**: Efficient for a small number of parameters.
- **Reverse/Adjoint Mode**: Preferred for many parameters or high memory demand.

21.4 Applications of Sensitivity Analysis

- **Sensitivity Analysis**: Measures how the solution changes with initial conditions or parameters.
- **Parameter Estimation**: Identifies parameters that best match observed data.
- **Control**: Determines how to drive the solution to a desired state.

22 Continuous-Adjoint Sensitivity Analysis

Continuous-adjoint sensitivity analysis computes the gradient of a cost function $G(u, p)$ where u is a function of p, and G is expressed as:

$$G(u,p) = \int_{t_0}^{T} g(u(t,p),p)\,dt,$$

with the constraint $\frac{du}{dt} = f(u,p,t)$.

22.1 Derivation

To find $\frac{dG}{dp}$, we introduce a Lagrange multiplier $\lambda(t)$:

$$G(u,p) = \int_{t_0}^{T} g(u(t,p),p)\,dt - \int_{t_0}^{T} \lambda^T \left(\frac{du}{dt} - f(u,p,t)\right) dt.$$

Applying the chain rule, we get:

$$\frac{dG}{dp} = \int_{t_0}^{T} \left[\frac{\partial g}{\partial u}\frac{du}{dp} + \frac{\partial g}{\partial p} \right] dt - \int_{t_0}^{T} \lambda^T \left(\frac{d}{dp}\frac{du}{dt} - \frac{\partial f}{\partial u}\frac{du}{dp} - \frac{\partial f}{\partial p} \right) dt.$$

Integrating by parts, and choosing λ to satisfy the adjoint ODE:

$$\lambda^T(t) \left[\frac{d}{dp} u(t) \right]_{t_0}^{T} - \int_{t_0}^{T} \left[\lambda'^T s + \lambda^T f_u s + \lambda^T f_p \right] dt.$$

To simplify, we choose λ such that:

$$\lambda'^T = -\lambda^T f_u - g_u, \quad \lambda(T) = 0,$$

resulting in:

$$\frac{dG}{dp} = \int_{t_0}^{T} \left[g_p + \lambda^T f_p \right] dt.$$

22.2 Summary

1. Solve the forward ODE $\frac{du}{dt} = f(u, p, t)$.
2. Solve the adjoint ODE $\lambda'^T = -\lambda^T f_u - g_u$, with $\lambda(T) = 0$.
3. Compute $\frac{dG}{dp}$ using the integral $\int_{t_0}^{T} \left[g_p + \lambda^T f_p \right] dt$.

23 Efficient Evaluation Techniques

- **Checkpointing**: Store intermediate states to mitigate errors from non-reversible numerical solvers.
- **Reverse Mode**: Efficient for problems with many parameters.
- **Numerical Integration**: Convert integration into an ODE problem using methods like Euler's method for approximating integrals.

24 Calculus of Variations and Gradient Functionals

The calculus of variations extends these ideas to functionals, where derivatives are taken with respect to functions. For example, given a functional:

$$f(u) = \int_0^1 \sin(u(x))\, dx,$$

the derivative (gradient) with respect to $u(x)$ can be computed as:

$$df = \int_0^1 \cos(u(x))\, du(x)\, dx.$$

24.1 Euler-Lagrange Equation

For functionals of the form $f(u) = \int_a^b F(u, u', x)\, dx$, the extremum is found using the Euler-Lagrange equation:

$$\frac{\partial F}{\partial u} - \frac{d}{dx}\left(\frac{\partial F}{\partial u'}\right) = 0.$$

In summary, the calculus of variations and gradient functionals enable the extension of sensitivity analysis to more complex systems and functionals, maintaining the principles of differentiation within the functional space.

25 Derivatives of Random Functions

When dealing with functions that incorporate randomness, such as stochastic functions, we can still define a concept of a derivative "on average." This derivative's expectation value corresponds to the derivative of the expectation value of the function. This concept is particularly relevant in fields like machine learning, for instance, in stochastic gradient descent.

26 Motivation

Consider the function $f(A) = A^2$ where A is a matrix. Here, both the input and output spaces are matrices, complicating the differentiation process. Ultimately, derivatives measure how the output changes when the input is perturbed. For the given function f:

$$df = (A + dA)^2 - A^2 = dA \cdot A + A \cdot dA + (dA)^2$$

The derivative aims to capture the sensitivity of f with respect to A, focusing on the first-order change:

$$dA \cdot A + A \cdot dA$$

Since $(dA)^2$ is a higher-order term, it is neglected, simplifying our function to:

$$df = dA \cdot A + A \cdot dA$$

This represents a matrix-to-matrix function and illustrates how small changes in A influence the output.

27 Random Functions and Their Derivatives

For random functions, we extend the notion of derivatives to more complex spaces. Consider a random function X, a random variable-valued function defined as $X : p \to X(p)$, where p is a real number. Examples include:

- $X(p) \sim \text{Bernoulli}(p)$
- $X(p) \sim \text{Exp}(p)$

In these cases, $X(p)$ represents a Bernoulli or Exponential random variable parameterized by p. For instance, $X(p)$ may yield true 60% of the time and false 40% of the time.

28 Motivation for Sensitivity Analysis

For matrix-valued functions, gradients are useful for optimization. While optimizing deterministic functions is straightforward, the need for gradients in random functions is less clear. The goal is to maximize the likelihood of certain events, typically involving statistical quantities based on expectations.

Consider a variational autoencoder (VAE) in machine learning, where randomness is intrinsic. The loss function $L(p) = \mathbb{E}[X(p)]$ is defined as an expectation. Although computing $L(p)$ analytically is challenging, sampling from $X(p)$ is computationally simple.

To find $\frac{dL(p)}{dp}$, we start from $X(p)$. In physical sciences, inherently stochastic models describe interactions averaged over time, motivating the need for gradient estimators.

29 Characterizing Sensitivity and Derivatives

Given $X(p)$, we are interested in how the output changes with respect to the input:

$$dX(\epsilon) = X(p+\epsilon) - X(p)$$

Here, $dX(\epsilon)$ is a stochastic difference and thus a random variable. To fully understand $dX(\epsilon)$, we need to consider the joint distribution of $X(p+\epsilon)$ and $X(p)$. Independent sampling often leads to large variance, complicating sensitivity analysis.

30 Defining Random Variables

A random variable $X(p)$ is defined as a map $\Omega \to \mathbb{R}$, where Ω is a sample space equipped with a probability distribution P. The randomness stems from P, independent of p. The map, parametrized by p, determines the specific random variable in the family.

For an exponential distribution $X(p) \sim \text{Exp}(p)$ with $\Omega = [0,1]$ and P being uniform, the function for $X(p)$ is:

$$-p\log(1-\omega) = X(p)(\omega)$$

Here, ω is sampled from Ω according to P, and plugged into the map.

31 Sampling and Variance Reduction

To accurately estimate $\frac{dX(\epsilon)}{dp}$, we sample ω once and use it across different values of p. This approach, known as the reparameterization trick, reduces variance. The differential $dX(\epsilon)$ behaves like $O(\epsilon)$ for small ϵ, allowing us to compute derivatives pointwise.

32 Generalizing to the Discrete Case

For discrete random variables, defining derivatives is challenging due to the discrete nature of the functions. The differential between two points is often zero except at discrete jumps, where it is large. This complicates sensitivity analysis.

The stochastic triple—a concept extending automatic differentiation—captures the function value, its derivative, and the probability of jumps. This approach provides a comprehensive understanding of how discrete functions perturb under small changes in the input.

In summary, by considering differentials and leveraging joint distributions, we can define meaningful derivatives for random functions, facilitating optimization and sensitivity analysis in various applications.

Conclusion

In conclusion, understanding and applying derivatives in complex contexts is crucial in today's world, where machine learning, large-scale optimization, and other advanced fields frequently require the differentiation of intricate functions. While basic calculus techniques may suffice for simple functions, the need for more sophisticated approaches becomes evident when dealing with matrix determinants, differential equations, or large-scale engineering calculations. This demand is amplified by the prevalence of machine learning, where derivatives with respect to numerous parameters are often necessary.

The material covered in this paper highlights the significance of matrix calculus and its foundational role in linear algebra, providing a deeper understanding of differentiation beyond the basic rules learned in early calculus courses. By focusing on the linearization aspect of derivatives, we gain the tools needed to efficiently compute derivatives in complex scenarios, enabling advancements in machine learning, engineering design, physical modeling, sensitivity analysis, and data science.

As we continue to advance in technology and data-driven fields, proficiency in matrix calculus will remain a cornerstone for innovation and problem-solving. Its ability to handle the differentiation of complicated systems efficiently ensures its ongoing significance in numerous scientific and engineering applications.