Matrix Calculus

Jack David Carson

January 31, 2025

Contents

1 Introduction and Motivation **3**
1.1 Why Matrix Calculus? . . . 3
1.2 A Quick Example in Words . . . 3

2 Part I: Fundamentals of Matrix Calculus **3**
2.1 Notation and Shapes . . . 3
2.2 Basic Principles . . . 4
2.2.1 Scalar-by-Vector Differentiation . . . 4
2.2.2 Scalar-by-Matrix Differentiation . . . 4
2.2.3 Matrix-by-Matrix Differentiation . . . 4
2.3 Chain Rule for Matrices . . . 4

3 Part II: Illustrative Examples **4**
3.1 Example 1: A Simple Linear Function . . . 5
3.1.1 Coordinate Expansion . . . 5
3.2 Example 2: Squared Error Loss, Single Layer . . . 5
3.2.1 Chain Rule Computation . . . 5
3.3 Example 3: A Matrix-Valued Output . . . 6

4 Part III: Matrix Gradients in Deep Learning **6**
4.1 Forward Pass Recap . . . 6
4.2 Backward Pass: Matrix Gradient Computation . . . 6
4.3 Tying Back to Examples . . . 7

5 Part IV: Advanced Notes and Best Practices **7**
5.1 Notation Systems . . . 7
5.2 Higher-Order Tensors . . . 8
5.3 Practical Tips . . . 8

6 Conclusion and References **8**

Preface

Matrix calculus plays an essential role in deep learning and related areas of machine learning. From basic linear regression to multi-layer neural networks, gradient-based optimization underlies most modern techniques. This document aims to provide a *compact but intensive* treatment of matrix calculus, focusing on *how* and *why* to compute gradients with respect to matrix-shaped parameters, both in theory and practice.

We assume readers have some familiarity with basic linear algebra (matrix multiplication, vector spaces) and multivariable calculus (partial derivatives, chain rule). That said, we start at fundamentals and build up to more advanced examples.

The structure is as follows:

- **Part I: Fundamentals.** We clarify the notation and essential concepts of matrix calculus, including scalar-by-matrix derivatives and matrix-by-matrix derivatives.
- **Part II: Illustrative Examples.** We work through classical examples (e.g. linear layers, squared-error losses) and see step-by-step partial derivatives.
- **Part III: Deep Learning Context.** We tie these derivations to backpropagation in neural networks, showing how the same rules apply repeatedly in a chain of transformations.
- **Part IV: Advanced Notes.** Some advanced remarks on notation systems, higher-dimensional tensors, and best practices.
- **Appendices.** Additional resources, references, and recommended reading.

1 Introduction and Motivation

Gradient-based methods power the vast majority of modern machine learning, from linear and logistic regression up to large transformer-based language models. At their core lies *matrix calculus*, the field that generalizes partial derivatives to matrix or tensor functions.

1.1 Why Matrix Calculus?

In traditional calculus, we often see a scalar function $f(x)$, where $x \in \mathbb{R}$, and $\frac{df}{dx}$ is a single derivative. When $x \in \mathbb{R}^n$, the gradient $\nabla_x f$ becomes an n-dimensional vector, collecting all partial derivatives. However, in deep learning, parameters typically live in matrices or higher-order tensors. Hence we need to carefully define and compute partial derivatives *with respect to* matrix entries.

One might flatten matrices into vectors and proceed with standard multivariable calculus. Indeed, frameworks like `PyTorch` or `TensorFlow` internally store parameters as 1D memory arrays. But mathematically, it is often more transparent to *retain the shape*, because each row and column in your weight matrix can have distinct roles (e.g. each row might correspond to a neuron).

1.2 A Quick Example in Words

Consider a simple single-layer neural network with weight matrix W. Our network output is $y = Wx$ for an input vector x. If the loss is the mean squared error versus a target t, the gradient $\nabla_W L$ needs to tell us: *"How does each entry W_{ij} of W affect the final loss L?"* We will see that the derivative is $(Wx - t)x^\mathsf{T}$ in matrix form. This document explains *why* and *how* this arises, and how it generalizes to deeper or more nonlinear models.

2 Part I: Fundamentals of Matrix Calculus

In this section, we provide a thorough introduction to matrix calculus. We start with notation and the definition of partial derivatives when inputs and outputs are matrices or vectors, then move on to some essential identities.

2.1 Notation and Shapes

Vectors and Matrices. We typically denote vectors by bold lowercase letters (e.g. $\mathbf{x} \in \mathbb{R}^n$) and matrices by uppercase letters (e.g. $A \in \mathbb{R}^{m \times n}$). The entry in row i and column j of A is A_{ij}.

Differentiation. When we write $\frac{\partial f}{\partial x}$ for $f : \mathbb{R}^n \to \mathbb{R}$, we mean the gradient (an n-dimensional vector). When $f : \mathbb{R}^{m \times n} \to \mathbb{R}$, i.e. f is a real-valued function of a matrix $A \in \mathbb{R}^{m \times n}$, $\frac{\partial f}{\partial A}$ is an $m \times n$ matrix whose (i, j)-th entry is $\frac{\partial f}{\partial A_{ij}}$.

Similarly, if we have a matrix-valued function $F(A) \in \mathbb{R}^{p \times q}$ of $A \in \mathbb{R}^{m \times n}$, its "gradient" with respect to A is a 4D array (or "4D tensor") collecting all partial derivatives $\frac{\partial F_{ab}}{\partial A_{ij}}$. In practice, we usually flatten or otherwise reshape these for computational convenience.

Common Conventions. Some references (e.g. machine learning frameworks) adopt a "row-major" or "column-major" flattening by default. Moreover, you may see small differences in notation, such as $\nabla_A f$ vs. $\partial f / \partial A$. We will use them interchangeably when the context is clear.

2.2 Basic Principles

2.2.1 Scalar-by-Vector Differentiation

Recall that if $f(\mathbf{x})$ is a scalar function of a vector $\mathbf{x} \in \mathbb{R}^n$, then the gradient is defined componentwise:

$$(\nabla_{\mathbf{x}} f)_i \;=\; \frac{\partial f}{\partial x_i}.$$

If $\mathbf{y} = A\mathbf{x}$, we also have the familiar result $\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = A$, where the shape of $\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$ is $(m \times n)$ if A is $(m \times n)$.

2.2.2 Scalar-by-Matrix Differentiation

Let $f : \mathbb{R}^{m \times n} \to \mathbb{R}$. Then the gradient $\nabla_A f$ (also written $\frac{\partial f}{\partial A}$) is the $m \times n$ matrix of partial derivatives:

$$(\nabla_A f)_{ij} \;=\; \frac{\partial f}{\partial A_{ij}}.$$

To compute $\nabla_A f$ in practice, *expand f in terms of the entries of A*, then differentiate with respect to each A_{ij}.

Example: Frobenius Norm. As a quick example, if $f(A) = \frac{1}{2}\|A\|_F^2 = \frac{1}{2}\sum_{i,j} A_{ij}^2$, then

$$\frac{\partial f}{\partial A_{ij}} = A_{ij},$$

so $\frac{\partial f}{\partial A} = A$ (in shape $m \times n$).

2.2.3 Matrix-by-Matrix Differentiation

When the output is also a matrix, say $F(A) \in \mathbb{R}^{p \times q}$, we define the *Jacobian tensor* $\frac{\partial F}{\partial A}$ whose entries are

$$\frac{\partial F_{ab}}{\partial A_{ij}} \quad \text{for all } a = 1, \ldots, p,\ b = 1, \ldots, q,\ i = 1, \ldots, m,\ j = 1, \ldots, n.$$

This collection can be viewed as a 4D object in $\mathbb{R}^{p \times q \times m \times n}$. In many settings (e.g. neural network backprop), we do not keep a 4D array explicitly, but reason about transformations and chain rules in terms of smaller building blocks.

2.3 Chain Rule for Matrices

A crucial tool is the chain rule. If $Z = g(A)$ is an intermediate matrix, and $f(Z)$ is a scalar function, then by chain rule:

$$\frac{\partial f}{\partial A_{ij}} \;=\; \sum_{p,q} \frac{\partial f}{\partial Z_{pq}} \frac{\partial Z_{pq}}{\partial A_{ij}}.$$

We will use this extensively when A is a parameter matrix and Z is some layer output or residual.

3 Part II: Illustrative Examples

In this section, we ground the definitions and rules of matrix calculus with step-by-step concrete examples. These examples reflect common patterns in deep learning.

3.1 Example 1: A Simple Linear Function

Let $A \in \mathbb{R}^{m\times n}$ and vectors $x \in \mathbb{R}^n, y \in \mathbb{R}^m$. Define a scalar

$$f(A) \;=\; y^{\mathsf{T}}\, A\, x.$$

We wish to compute $\nabla_A f$.

3.1.1 Coordinate Expansion

Write $f(A)$ as

$$f(A) \;=\; \sum_{i=1}^{m}\sum_{j=1}^{n} y_i\, A_{ij}\, x_j.$$

Then

$$\frac{\partial f}{\partial A_{ij}} \;=\; y_i\, x_j.$$

Hence the gradient matrix $\nabla_A f$ (of size $m \times n$) is

$$\nabla_A f \;=\; y\, x^{\mathsf{T}}.$$

In a deep-learning context, if A were a weight matrix, this example corresponds to the derivative of a single "dot-product" output with respect to A.

3.2 Example 2: Squared Error Loss, Single Layer

Suppose we have:

$$L(W) \;=\; \tfrac{1}{2}\,\big\|W\,x \;-\; t\big\|^2,$$

where $W \in \mathbb{R}^{m\times n}$, $x \in \mathbb{R}^n$, and $t \in \mathbb{R}^m$. Let $r = Wx - t$ be the residual. Then

$$L(W) \;=\; \tfrac{1}{2}\sum_{i=1}^{m}(r_i)^2, \quad \text{where} \quad r_i \;=\; \sum_{j=1}^{n} W_{ij}\, x_j \;-\; t_i.$$

3.2.1 Chain Rule Computation

By direct differentiation:

$$\frac{\partial L}{\partial W_{ij}} \;=\; \left(\frac{\partial}{\partial W_{ij}}\,\tfrac{1}{2}\sum_{p=1}^{m} r_p^2\right) \;=\; \sum_{p=1}^{m} r_p\,\frac{\partial r_p}{\partial W_{ij}}.$$

But r_p depends on W_{ij} *only* if $p = i$. Thus

$$\frac{\partial r_p}{\partial W_{ij}} \;=\; \begin{cases} x_j, & \text{if } p = i, \\ 0, & \text{otherwise.} \end{cases}$$

Hence

$$\frac{\partial L}{\partial W_{ij}} \;=\; r_i\, x_j.$$

Rewriting in matrix form:

$$\nabla_W L \;=\; (Wx - t)\, x^{\mathsf{T}}.$$

Interpretation. In neural network terms, this says that the gradient at W_{ij} (the connection from x_j into the i-th output neuron) depends on how large the i-th residual r_i is, times the input x_j. This exactly matches the usual backprop intuition: the error signal at neuron i times the value of input j drives the update direction for W_{ij}.

3.3 Example 3: A Matrix-Valued Output

Sometimes, the output itself is a matrix. For instance, if $R \in \mathbb{R}^{m\times n}$, we might define $F(R) = R^{\mathsf{T}}R$. Then $F(R) \in \mathbb{R}^{n\times n}$. Its (p,q)-th entry is

$$F(R)_{pq} \;=\; \sum_{i=1}^{m} R_{ip}\, R_{iq}.$$

The "gradient" with respect to R is a 4D array $\frac{\partial F_{pq}}{\partial R_{ij}}$. Typically, we flatten or otherwise keep track of each entry's dependence when applying chain rules in a larger computation graph.

4 Part III: Matrix Gradients in Deep Learning

We now connect these ideas to the forward and backward pass of a (simplified) deep neural network. Matrix gradients appear at every layer, because the weights are naturally stored as matrices.

4.1 Forward Pass Recap

A typical layer in a neural network transforms an input $\mathbf{z}^{(\ell-1)}$ into

$$\mathbf{z}^{(\ell)} \;=\; \sigma\big(W^{(\ell)}\,\mathbf{z}^{(\ell-1)} + \mathbf{b}^{(\ell)}\big),$$

where $W^{(\ell)} \in \mathbb{R}^{m\times n}$, $\mathbf{b}^{(\ell)} \in \mathbb{R}^{m}$, and $\sigma(\cdot)$ is a nonlinear activation (e.g. ReLU, sigmoid, etc.). The forward pass uses these weight matrices $W^{(\ell)}$ to map from layer $\ell-1$ to layer ℓ.

If we have L layers, the final output might be $\mathbf{z}^{(L)}$. Then we define a scalar loss function $L\big(\mathbf{z}^{(L)}, \text{target}\big)$.

4.2 Backward Pass: Matrix Gradient Computation

To train via gradient-based methods (e.g. stochastic gradient descent), we need $\nabla_{W^{(\ell)}} L$ for each layer ℓ. That is exactly a *matrix* of partial derivatives:

$$(\nabla_{W^{(\ell)}} L)_{ij} \;=\; \frac{\partial L}{\partial W_{ij}^{(\ell)}}.$$

We apply the chain rule repeatedly:

$$\frac{\partial L}{\partial W_{ij}^{(\ell)}} = \sum_{p=1}^{m} \frac{\partial L}{\partial z_p^{(\ell)}}\, \frac{\partial z_p^{(\ell)}}{\partial W_{ij}^{(\ell)}}.$$

But $z_p^{(\ell)} = \sigma\big(\underbrace{[W^{(\ell)}\,\mathbf{z}^{(\ell-1)} + \mathbf{b}^{(\ell)}]}_{u^{(\ell)}}\big)_p$, where $u_p^{(\ell)} = \sum_k W_{p,k}^{(\ell)}\, z_k^{(\ell-1)} + b_p^{(\ell)}$.

For typical elementwise nonlinear σ,

$$\frac{\partial z_p^{(\ell)}}{\partial W_{ij}^{(\ell)}} \;=\; \sigma'\big(u_p^{(\ell)}\big)\,\frac{\partial u_p^{(\ell)}}{\partial W_{ij}^{(\ell)}}.$$

Further,

$$\frac{\partial u_p^{(\ell)}}{\partial W_{ij}^{(\ell)}} \;=\; \begin{cases} z_j^{(\ell-1)}, & \text{if } p = i, \\ 0, & \text{otherwise.} \end{cases}$$

Putting it all together yields

$$\frac{\partial z_p^{(\ell)}}{\partial W_{ij}^{(\ell)}} \;=\; \delta_{p,i}\;\sigma'(u_p^{(\ell)})\;z_j^{(\ell-1)},$$

where $\delta_{p,i}$ is the Kronecker delta (1 if $p = i$, else 0).

Hence

$$\frac{\partial L}{\partial W_{ij}^{(\ell)}} \;=\; \frac{\partial L}{\partial z_i^{(\ell)}}\;\sigma'(u_i^{(\ell)})\;z_j^{(\ell-1)}.$$

Defining the *backpropagated signal* $\delta_i^{(\ell)} = \frac{\partial L}{\partial z_i^{(\ell)}}\;\sigma'(u_i^{(\ell)})$, we can write

$$\nabla_{W^{(\ell)}} L \;=\; \delta^{(\ell)}\,\big(z^{(\ell-1)}\big)^{\mathsf{T}},$$

analogous to our earlier examples: $(\mathbf{r})\,(\mathbf{x})^{\mathsf{T}}$.

Key Takeaway. Each row i of $W^{(\ell)}$ gets multiplied by the i-th component of the backpropagated error $\delta^{(\ell)}$, and each column j gets multiplied by the corresponding component of the input $z_j^{(\ell-1)}$. Hence the gradient is the outer product of *error signals* and *input activations*.

4.3 Tying Back to Examples

Compare this to the squared-error example (§3.2), where $\delta = (Wx - t)$ took the role of the error signal, and x was the input. Everything generalizes seamlessly once you realize the shape of partial derivatives is always an outer product in linear layers—plus a factor from the derivative of nonlinearities.

5 Part IV: Advanced Notes and Best Practices

This section comments on more nuanced aspects of matrix calculus, including different notational systems, higher-order tensors, and practical tips.

5.1 Notation Systems

Different fields use slightly different notation conventions for matrix calculus. Three common ones are:

1. **Numerical / Flattening Convention:** Flatten all matrices/vectors into 1D arrays, then treat everything as standard multivariable calculus. This is how many software libraries implement automatic differentiation internally.

2. **Denominator Layout vs. Numerator Layout:** Some authors define the gradient as the transpose of what others might define. For instance, in neural networks, we often write $\nabla_x f$ as a column vector by default, whereas some references might stack partial derivatives in a row vector. Always confirm which layout is being used.

3. **Index Notation / Einstein Summation:** In physics or more mathematically rigorous treatments, index notation is used heavily: $A_{ij}B_{jk} = C_{ik}$. It can streamline the chain rule but often looks intimidating to beginners.

Overall, the *results* are the same. In a single project, it is best to fix one approach and stay consistent.

5.2 Higher-Order Tensors

Many deep learning frameworks extend beyond $\mathbb{R}^{m\times n}$ to 4D or 5D tensors, especially for convolutional layers (where weights are [filters $\times$ channels $\times$ height $\times$ width]). Conceptually, each dimension just adds another index to partial derivatives. The chain rule is unaffected in principle, though it can be more unwieldy to write by hand. Automatic differentiation tools handle these higher-order derivatives seamlessly.

5.3 Practical Tips

- **Use Vectorized Expressions.** In actual code, keep shapes explicit but exploit matrix operations to speed up computations (e.g. use BLAS libraries).
- **Check Gradients Numerically.** When implementing new architectures, it is prudent to verify by finite differences that your analytical gradients match the numerical approximations. This is often done on small "toy" inputs to confirm correctness.
- **Keep Track of Shapes.** Most dimension errors come from mixing up row vs. column vectors or reversing the shape of a weight matrix. Write out shape diagrams to avoid confusion.
- **Leverage Automatic Differentiation.** While matrix calculus is vital for conceptual understanding, we often let frameworks like `PyTorch`, `TensorFlow`, `JAX`, or `Autograd` perform the actual gradient computations. Knowing the math ensures you can debug or design new operations correctly.

6 Conclusion and References

Matrix calculus is the bedrock of gradient-based optimization in machine learning. We have seen how the concept of partial derivatives naturally extends to matrices: the gradient of a scalar function with respect to a matrix is another matrix of matching shape, and matrix-by-matrix derivatives lead to higher-dimensional arrays. Key formulas (like $\nabla_W \frac{1}{2}\|Wx - t\|^2 = (Wx - t)\,x^\mathsf{T}$) reveal patterns repeated throughout neural networks (outer products between error signals and activations). Understanding these step-by-step derivations provides the insight behind the backpropagation algorithm and helps debug dimensional misalignments or design new architectures.

Recommended Reading

- **Matrix Differential Calculus with Applications to Statistics and Econometrics**, Magnus and Neudecker.
- **Deep Learning**, Goodfellow, Bengio, and Courville (especially the chapters on backprop and optimization).
- Online Resources, e.g. Matrix Calculus Explained, or the `cs231n` *Convolutional Neural Networks for Visual Recognition* notes for step-by-step gradient derivations.
- Official documentations of `PyTorch`, `TensorFlow`, or `JAX` for examples of automatic differentiation in practice.

Appendix: Symbols and Conventions

Symbol	Meaning
$\mathbf{x}, \mathbf{y}, \mathbf{z}$	Vectors (bold lowercase)
A, B, W	Matrices (capital letters)
$\nabla_A f$	Gradient of scalar f wrt matrix A
$\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$	Jacobian (matrix) when $\mathbf{y}$ is vector-valued
$\sigma(\cdot)$	Nonlinear activation (ReLU, sigmoid, etc.)

End of Document