1 Advanced automatic differentiation

About this chapter

This chapter discusses automatic differentiation in generic vector spaces, as a companion to Chapter 6 in Volume 1, which focused only on functions $f : \mathbb{R}^m \to \mathbb{R}^p$ and their matrix Jacobians. We also describe some implementation issues, extensions for generic computational graphs, and implicit differentiation. Finally, we provide an overview of multilinear (tensor) algebra. Most of the material is adapted or abridged from [BR24, BEJ25].

1.1 Gradients and Jacobians

We begin by reviewing the material from Volume 1, highlighting a few shortcomings as motivation and uniforming the notation to what we will use in the rest of the chapter. We will see that while gradients are generally introduced in terms of vectors and matrices (i.e., gradients and Jacobians), it is significantly better to take a more abstract point of view and treat them as linear maps over generic vector spaces. In fact, automatic differentiation is a fantastic playground to get acquainted with ideas from abstract (multi)linear algebra, and we will spend a significant amount of time moving from one to the other.

1.1.1 Derivatives of scalar functions

Let us start with a few trivial observations that will be useful in the following. We first recall the definition of derivative for a scalar function $f : \mathbb{R} \rightarrow \mathbb{R}$.

Definition D.1.1 (Derivative, scalar case)

For a function $f(x) : \mathbb{R} \rightarrow \mathbb{R}$, the derivative is defined as:

$$\partial f(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \tag{E.1.1}$$

As in Volume 1, we do not consider conditions for the existence of a derivative, and we simply assume all our functions are differentiable. Around the point x, the derivative is the slope of the *best* linear approximation to f, in the sense:

$$f(x+h) - f(x) \approx \partial f(x) \cdot h \tag{E.1.2}$$

where we ignore all possible terms of higher-order (this can be formalized in several ways [BR24, BEJ25], but we keep our treatment as simple as possible).

Importantly, the derivative acts *linearly* over h. Because this is a fundamental concept, we introduce a specialized notation for it. Anytime an object l acts linearly on another object x, we use square brackets, i.e., we use square brackets whenever:

$$l[ax + by] = a \cdot l[x] + b \cdot l[y] \tag{E.1.3}$$

for some scalars a and b. With this notation, we rewrite (E.1.2) by adding square brackets:

$$f(x+h) - f(x) \approx \partial f(x)[h] \tag{E.1.4}$$

In addition, we can highlight the interpretation of the derivative as first-order approximation by moving around terms in (E.1.1):

$$\lim_{h \to 0} \frac{\overbrace{f(x+h) - f(x)}^{\text{Real difference}} - \overbrace{\partial f(x)[h]}^{\text{Linear approximation}}}{h} = 0 \tag{E.1.5}$$

Hence, the derivative $\partial f(x)$ is the best *linear function over h approximating small variations around $f(x)$*.[1] This is all basic material, but it will be useful to keep this notation in mind for the rest of the chapter.

1.1.2 Directional derivatives

Mirroring the discussion in Volume 1, let us now move to vector-valued functions $f(\mathbf{x}) : \mathbb{R}^m \to \mathbb{R}$. This time, in order to discuss linear approximations around a point $\mathbf{x}$, we first need to select some direction $\mathbf{u}$. We can then define the **directional derivative**.

[1] Since $h \in \mathbb{R}$, all linear functions over it are of the form $l[x] = ax$ for some real number a.

Definition D.1.2 (Directional derivative)

For a function $f(\mathbf{x}) : \mathbb{R}^m \to \mathbb{R}$*, an input* $\mathbf{x}$*, and a direction* $\mathbf{u}$*, the* ***directional derivative*** *is defined as:*

$$\partial f(\mathbf{x})[\mathbf{u}] = \lim_{h \to 0} \frac{f(\mathbf{x} + \overset{\text{Small displacement along } \mathbf{u}}{\boxed{h\mathbf{u}}}) - f(\mathbf{x})}{h} \tag{E.1.6}$$

With a bit of foreshadowing, we have used the same notation $\partial f(\mathbf{x})[\mathbf{u}]$ as the previous section, which implies linearity over $\mathbf{u}$. As we will see in Section 1.2.1, this means that we should be able to express the derivative as:

$$\partial f(\mathbf{x})[\mathbf{u}] = \langle \bullet, \mathbf{u} \rangle \tag{E.1.7}$$

for some vector $\bullet$, where $\langle \mathbf{x}, \mathbf{y} \rangle = \mathbf{x}^\top \mathbf{y}$ denotes the standard Euclidean inner product. To see this, note that we can express the vector $\mathbf{u}$ as:

$$\mathbf{u} = \sum_i u_i \mathbf{e}_i \tag{E.1.8}$$

where $u_1, \ldots, u_m$ are the m coordinates of the vector and $\mathbf{e}_i$ the canonical basis vectors:

$$(\mathbf{e}_i)_j = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise} \end{cases} \tag{E.1.9}$$

Substituting in the definition of the directional derivative (and exploiting the linearity of the limit) we obtain:

$$\partial f(\mathbf{x})[\mathbf{u}] = \sum_i \partial f(\mathbf{x})[\mathbf{e}_i] \cdot u_i \tag{E.1.10}$$

where $\partial f(\mathbf{x})[\mathbf{e}_i]$ is known as the i-th **partial derivative** of f. While there are infinite possible directional derivatives, there are only m possible partial derivatives. It is customary to collect them in a column vector known as the **gradient** of the function:

$$\nabla f(\mathbf{x}) = \begin{bmatrix} \partial f(\mathbf{x})[\mathbf{e}_1] \\ \vdots \\ \partial f(\mathbf{x})[\mathbf{e}_m] \end{bmatrix} \tag{E.1.11}$$

We can now complete the definition in (E.1.7):

$$\partial f(\mathbf{x})[\mathbf{u}] = \langle \nabla f(\mathbf{x}), \mathbf{u} \rangle \tag{E.1.12}$$

where once again the inner product is the standard Euclidean one (in more general inner product spaces, the expression changes accordingly, as we show in Section 1.2.2). Hence, if we have access to the gradient of a function, we can compute all possible directional derivatives by taking inner products with the corresponding directions.

1.1.3 Jacobian-vector products

We now move to the case of a vector-to-vector function $f(\mathbf{x}) : \mathbb{R}^m \rightarrow \mathbb{R}^p$. We can immediately use the definition of partial derivative to define pm derivatives $\partial_j f_i(\mathbf{x}) = \frac{\partial f_i(\mathbf{x})}{\partial x_j}$, which we can spatially organize into a matrix format.

Important

Definition D.1.3 (Jacobian)

*The **Jacobian** of a function* $f(\mathbf{x}) : \mathbb{R}^m \to \mathbb{R}^p$ *is defined as the* (p, m) *matrix collecting all partial derivatives:*

$$\partial f(\mathbf{x}) = \begin{pmatrix} \partial_1 f_1(\mathbf{x}) & \cdots & \partial_m f_1(\mathbf{x}) \\ \vdots & \ddots & \vdots \\ \partial_1 f_p(\mathbf{x}) & \cdots & \partial_m f_p(\mathbf{x}) \end{pmatrix} \tag{E.1.13}$$

Jacobians satisfy all the rules we commonly associate with derivatives, including linearity, the product rule, and the fundamental chain rule:

$$\partial(f \circ g)(\mathbf{x}) = \partial f(g(\mathbf{x}))\partial g(\mathbf{x}) \tag{E.1.14}$$

showing that function composition becomes matrix multiplication of the corresponding Jacobians when differentiating. Denoting by $\mathbf{J} = \partial f(\mathbf{x})$ the Jacobian matrix of f evaluated in $\mathbf{x}$, we can define an operation akin to a directional derivative, called a **Jacobian-vector product** (JVP):

$$\partial f(\mathbf{x})[\mathbf{u}] = \mathbf{J}\mathbf{u} \tag{E.1.15}$$

and a similar concept, known as **vector-Jacobian product** (VJP), by transposition (see Volume 1). Being matrix-vector products, JVPs and VJPs are both linear, justifying our notation.

In addition, we saw in Volume 1 that automatic differentiation algorithms can be defined fully in terms of JVPs and VJPs, and that they can often be computed

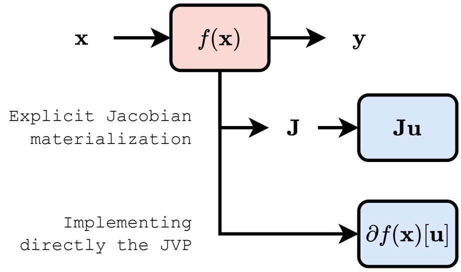

Figure F.1.1: *Two ways of implementing a JVP for a function $f(\mathbf{x})$: materializing the Jacobian $\mathbf{J}$ first (center), or directly providing the output (bottom). The two results are equivalent, but the bottom one may not require $\mathbf{J}$.*

directly without materializing $\mathbf{J}$ first (and are, in fact, the more fundamental objects in automatic differentiation). See Figure F.1.1 for a visualization.

As a simple example, consider an activation function applied elementwise $f(\mathbf{x}) = \phi(\mathbf{x})$. The Jacobian is a diagonal matrix collecting the vector of derivatives $\phi'(\mathbf{x}) \sim (m)$ on the diagonal, while the JVP is a masking operation that does not require the full diagonal matrix:

$$\partial f(\mathbf{x})[\mathbf{u}] = \phi'(\mathbf{x}) \odot \mathbf{u} \qquad \text{(E.1.16)}$$

where we recall $\odot$ stands for elementwise multiplication (Hadamard product) of two arrays.

1.1.4 Drawbacks

While the previous exposition is enough to introduce some of the most important concepts in automatic differentiation, it has a few drawbacks. Firstly, JVPs and VJPs are, in a sense, “more fundamental" than Jacobian matrices in the implementation, but here they arise only indirectly as a by-product of the Jacobians themselves. Secondly, the notation does not extend nicely to higher-order objects. For example, consider squaring a matrix $\mathbf{X} \sim (n, n)$:

$$f(\mathbf{W}) = \mathbf{W}^2 \tag{E.1.17}$$

Intuitively, the Jacobian here should be a hyper-matrix $T \sim (n, n, n, n)$ with 4 dimensions collecting all possible partial derivatives. In analogy with the previous discussion, we can linearly apply it to another matrix $\mathbf{U}$ to define the equivalent of a JVP:

$$(\partial f(\mathbf{W})[\mathbf{U}])_{ij} = \sum_k \sum_z T_{ijkz} U_{kz} \tag{E.1.18}$$

Formally, T is a (non-trivial) example of a **tensor**, and (E.1.18) is a generic *tensor contraction* over two indices, generalizing the notions of matrices and matrix multiplication (tensors will make quite a lot of appearances through the chapter – we will overview them in Section 1.4.3). Alternatively, we can consider an equivalent vectorized version of f:

$$f'(\text{vect}(\mathbf{X})) = \text{vect}(f(\mathbf{X})) \tag{E.1.19}$$

This allows us to define a "standard" Jacobian of shape (n^2, n^2) (and it works for all finite-dimensional objects), at the cost of loosing all information about the structure of the inputs and outputs. Even worse, doing this is wasteful since standard computations [BEJ25] show that:

$$\partial f(\mathbf{X})[\mathbf{U}] = \mathbf{UX} + \mathbf{XU} \tag{E.1.20}$$

and this expression holds irrespective of the chosen bases.

There are several other issues in the previous discussion. For example, (E.1.12) depends on using the standard Euclidean inner product, but what happens if we decide to work with a weighted inner product $\langle \mathbf{x}, \mathbf{y} \rangle_{\mathbf{W}} = \mathbf{x}^\top \mathbf{W} \mathbf{y}$? Worse, how do we handle differentiation in more sophisticated spaces, such as function spaces or spaces of polynomials, where we might not be able to define a vectorization operation?

To solve all these issues, we will take a step back and see how differentiation can be defined in generic vector spaces. To this end, we provide now a brief recap of what is known as *coordinate-free* linear algebra, i.e., algebra that does not depend on a specific choice of basis [Axl15].

1.2 Gradients as linear maps

One issue of the previous discussion is treating different functions (e.g., vector-to-scalar, vector-to-vector, ...) somehow differently, introducing apparently disconnected concepts and notation for each case. We now try to unify the discussion in order to extend it to more interesting scenarios, e.g., matrices-to-matrices functions.

To this end, consider again the matrix function $f(\mathbf{X}) = \mathbf{X}^2$. Following (E.1.5), we could search for a linear function $l_{\mathbf{X}}[\mathbf{U}]$ such that:

$$f(\mathbf{X}+\mathbf{U}) - f(\mathbf{X}) \approx l_{\mathbf{X}}[\mathbf{U}] \tag{E.1.21}$$

holds for “small" displacements (i.e., under a suitable norm). If we can do this, then we could identify $l_{\mathbf{X}}[\mathbf{U}]$ with our derivative $\partial f(\mathbf{X})[\mathbf{U}]$ and say that the function is differentiable.

Note the subtle switch: before, we introduced the derivative as a vector (gradient) or matrix (Jacobian), before interpreting it as a linear function. Here, we are *defining* it as a linear function from the onset! In fact, we have already seen how to recover the matrix view from the function view. To see this, note that we can define a basis for the space of all $n \times n$ matrices by defining n^2 one-hot matrices $\mathbf{E}_{ij}$:

$$(\mathbf{E}_{ij})_{zk} = \begin{cases} 1 & \text{if } i = z, j = k \\ 0 & \text{otherwise} \end{cases} \tag{E.1.22}$$

Trivially, any square matrix can be written as $\mathbf{X} = \sum_i \sum_j X_{ij}\mathbf{E}_{ij}$. As an example:

$$\begin{bmatrix} 3 & -4 \\ 1 & 8 \end{bmatrix} = 3\begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} + -4\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} + 1\begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix} + 8\begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$$

If we have access to the linear function $\partial f(\mathbf{X})[\mathbf{U}]$, we can call it n^2 times on the basis matrices $\partial f(\mathbf{X})[\mathbf{E}_{11}]$, $\partial f(\mathbf{X})[\mathbf{E}_{21}]$, ... Each call returns an $n \times n$ matrix, and stitching them together gives us the 4d materialization of the array as in (E.1.18)! This is exactly what we called "*materializing*" an array before. To clarify these ideas, we now proceed to introduce the linear algebra concepts we need.

1.2.1 A primer on linear algebra

Inner product spaces

For the rest of the chapter, suppose we want to differentiate a function with the following signature:

$$f : \mathscr{E} \to \mathscr{F} \tag{E.1.23}$$

where $\mathscr{E}$ and $\mathscr{F}$ are two *generic* vector spaces. Euclidean vectors in $\mathbb{R}^n$ are examples of vectors, but so are matrices (or specific subsets, such as the set of all symmetric matrices), tensors (Section 1.4.3), products of vector spaces, polynomials, integrable functions, and so on. We focus on the familiar case of real-valued, *finite-dimensional* vector spaces, but the majority of concepts extends to more esoteric use cases, including hypercomplex spaces, function spaces, and manifolds (through the notion of tangent spaces [BR24]).[2]

We want to define linear functions on $\mathscr{E}$ and $\mathscr{F}$ that are "good" approximations in the limit of very small perturbations, so the first thing we need is a notion of *similarity* (closeness) for a vector space. The easiest way to achieve this is to endow it with an inner product.

Important

Definition D.1.4 (Inner product) *An* ***inner product*** *over a vector space $\mathscr{X}$ is a function:*

$$\langle \bullet, \bullet \rangle : \mathscr{X} \times \mathscr{X} \to \mathbb{R} \tag{E.1.24}$$

[2] Any finite-dimensional vector space is isomorphic to $\mathbb{R}^n$ for some dimension n, but we already noted in (E.1.19) that carrying out the transformation by flattening is something we want to avoid.

such that:

- *The function is linear in each argument separately.*[a]
- $\langle x, y\rangle \geq 0$*, with equality only if the two terms are equal.*
- $\langle x, y\rangle = \langle y, x\rangle$*, i.e., the function is symmetric.*

[a] We say the inner product is an example of a *bilinear* functional.

We have already seen the standard dot product $\langle \mathbf{x}, \mathbf{y}\rangle = \mathbf{x}^\top \mathbf{y}$ for Euclidean vectors. Clearly, this is not the only function satisfying the previous properties. In fact, all inner products in $\mathbb{R}^n$ can be written as:

$$\langle \mathbf{x}, \mathbf{y}\rangle = \mathbf{x}^\top \mathbf{W}\mathbf{y} \tag{E.1.25}$$

for some positive-definite matrix $\mathbf{W}$. For matrices, we can easily define an inner product by extension (this is known as the Frobenius inner product of matrices):

$$\langle \mathbf{X}, \mathbf{Y}\rangle = \langle \text{vect}(\mathbf{X}), \text{vect}(\mathbf{Y})\rangle = \text{Tr}(\mathbf{X}^\top \mathbf{Y}) \tag{E.1.26}$$

where Tr is the trace operation (the sum of the elements over the diagonal). For integrable functions defined on $[0, 1]$ we can generalize this inner product as:

$$\langle f, g\rangle = \int_0^1 f(x)g(x)dx \tag{E.1.27}$$

We can also build compound inner products. As an example, if we take the Cartesian product of two vector spaces $\mathcal{X} \times \mathcal{Y}$, i.e., the space of all ordered pair of elements from the two spaces, we can define:

$$\langle x_1 \times y_1, x_2 \times y_2 \rangle = \langle x_1, x_2 \rangle + \langle y_1, y_2 \rangle \tag{E.1.28}$$

from the two inner products of the original spaces. Note that for simplicity we do not distinguish which space the inner product belongs to (e.g., using a subscript $\langle \bullet, \bullet \rangle_{\mathcal{X}}$).

Having an inner product immediately endows the space with a norm given by the inner product of a vector with itself:

$$\|x\| = \sqrt{\langle x, x \rangle} \tag{E.1.29}$$

With this choice of norm, the inner product can be written abstractly (by the Cauchy–Schwarz inequality) as $\langle x, y \rangle = \|x\|\|y\| \cos(\alpha)$ for some value $\alpha \in [0, \pi]$, which we can interpret as the angle between the two vectors (i.e., two non-zero vectors are orthogonal whenever $\langle x, y \rangle = 0$).

Linear functionals and maps

Irrespective of whether an inner product exists, all vector spaces have *linear* functions that map from the vector space to the reals – these are known as **linear functionals** (or **linear forms** or **covectors**).

> **Definition D.1.5 (Linear functionals)**
>
> *A **linear functional** on a (real-valued) vector space $\mathcal{X}$ is a function $l : \mathcal{X} \rightarrow \mathbb{R}$. The space of all linear functionals is called the **dual space** of $\mathcal{X}$ and it is generally denoted as $\mathcal{X}^*$. We use $l[x]$ for the application of a linear functional*

> *on a vector.*

Although every vector space has a dual space, linear functionals on inner product spaces are particularly interesting. The *Riesz representation theorem* from functional analysis [Rud87] states that *all* such functionals can be written as inner products:[3]

$$\overbrace{l[x]}^{\text{Functional (from the dual space)}} = \langle \underbrace{y}_{\text{Vector (from the original space)}}, x \rangle \tag{E.1.30}$$

where the pairing between l and y is known as the natural pairing. We have already implicitly used this theorem when we stated, for example, that all linear functionals on Euclidean vector spaces are of the form (E.1.30). As a result, the distinction between primal and dual spaces in classical algebra tends to be ignored, since linear functionals can be identified with row vectors. However, the result is general - for example, all linear functionals over matrices (when endowed with the Frobenius inner product) can be written as:

$$l[\mathbf{A}] = \mathrm{Tr}(\mathbf{B}^\top \mathbf{A}) \tag{E.1.31}$$

for some matrix $\mathbf{B}$. We can also define generic linear functions from a vector space $\mathscr{E}$ to another vector space $\mathscr{F}$ – we call these **linear maps**.

[3] This is true for finite-dimensional spaces, but only true for *complete* inner product spaces (Hilbert spaces) in general.

Table T.1.1: *A summary of the key concepts we need for a generic (real-valued) inner product space.*

Notation	Description
x	Vector (element of the vector space)
$\langle x, y \rangle$	Inner product with associated norm $\lVert x \rVert = \sqrt{\langle x, x \rangle}$.
$l[x] : \mathscr{X} \rightarrow \mathbb{R}$	Linear functional – can be written as inner product $l[x] = \langle y, x \rangle$.
$l[x] : \mathscr{E} \rightarrow \mathscr{F}$	Linear map – functionals and maps satisfy $l[ax + by] = al[x] + bl[y]$.

Characterizing linear maps

Can we also characterize the space of *all* possible linear maps? The answer is yes, but it requires more concepts from tensor algebra [Lim21]. For now, note that a pair of vector and covector $(y, l[\bullet])$ defines a linear map as:

$$f[x] = l[x]y \tag{E.1.32}$$

This is called a **rank-1** map, and the space of all linear maps can be characterized by considering all possible (finite in our case) linear sums of rank-1 terms. For inner product spaces we thus have:

$$f[x] = \sum_i \langle v_i, x \rangle y_i \tag{E.1.33}$$

for some vectors v_i and y_i. This is not immediately apparent: as an example, can you express (E.1.20) in this form?

1.2.2 Defining the gradient (again)

Do not miss

We are now ready to (re)define the concept of derivative for the general case of two vector spaces as in (E.1.23). In order to be consistent, from now on we use $x, u \in \mathcal{E}$ to denote *input* vectors and $y, v \in \mathcal{F}$ to denote *output* vectors (which are now potentially very complex objects).

Recall that we want to take small displacements $f(x+u) - f(x)$ and compare them to a *linear function* $l_x[u]$, which is in general different from point to point. Because we are considering inner product spaces, we can use the input norm $\|u\|$ to define a “small" displacement, and the output norm instead to compare the approximation. We obtain the following definition.

Important

Definition D.1.6 (Derivative, generic vector space)

We say a function $f : \mathcal{E} \rightarrow \mathcal{F}$ is (Fréchet) ***differentiable*** *if there exists a linear function $\partial f(x)$ for every input x such that:*

$$\lim_{\|u\| \to 0} \frac{\|f(x+u) - f(x) - \partial f(x)[u]\|}{\|u\|} = 0 \quad \text{(E.1.34)}$$

Keep in mind the important (and subtle) change in perspective. Before, we defined the matrix of partial derivatives (gradient or Jacobian) $\partial f(x)$ as the essential object, and only as a by-product we obtained the partial derivative (for the scalar case) or the JVP (for the vector case) $\partial f(x)[u]$, which is interpreted as “the matrix $\partial f(x)$ multiplied by u". We now consider the *map*, i.e., the directional derivative as the defining characteristic of differentiable functions, fully justifying our use of the notation $\partial f(x)[u]$, which becomes “the function $\partial f(x)$

applied on u".[4]

In functional programming notation we have:

$$\partial f : \mathscr{E} \rightarrow (\mathscr{E} \rightarrow \mathscr{F}) \tag{E.1.35}$$

Can we make the reverse step and go back to a matrix representation if we have access to a generic linear map? This is equivalent to materializing the Jacobian, and we have already discussed the solution in the context of the directional derivative and in the introduction to Section 1.2: since $u = \sum_i u_i e_i$ over the basis vectors e_i we have:

$$\partial f(x)[u] = \sum_i \partial f(x)[e_i] \cdot u_i \tag{E.1.36}$$

Thus, we can materialize the map as an array by running it in sequence over all basis vectors e_i, and this provides a full description of the map in that coordinate system – see Figure F.1.2 for a visualization. Once again, in most cases this would be hugely impractical, e.g., for a function mapping matrices to matrices we would have n^2 basis vectors and the previous equation would look like (E.1.18).

Consider now the special case of a function $f : \mathscr{E} \rightarrow \mathbb{R}$. If $\partial f(x)[u]$ exists, by the Riesz representation theorem we know it takes the form of an inner product with another vector v_x. In that case, we *define* such vector to be the gradient of the function.

[4]We will refer to $\partial f(x)[u]$ as the JVP for consistency, although its implementation rarely involves either Jacobians or products.

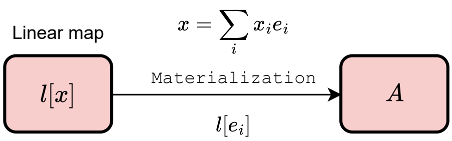

Figure F.1.2: *Materializing a generic linear map in a given set of coordinates can be done by calling the map over the basis vectors.*

Important

Definition D.1.7 (Gradient, generic vector space)

For a differentiable function $f : \mathcal{E} \to \mathbb{R}$, the gradient $\nabla f(x) \in \mathcal{E}$ is the vector such that:

$$\partial f(x)[u] = \langle \nabla f(x), u \rangle \tag{E.1.37}$$

We reuse the same symbol $\nabla f(x)$ for the gradient, although for a generic vector space this is *defined* as the vector paired with the derivative. Observe there is now a very strict (and non-trivial) pairing between our choice of inner product and the gradient we obtain, i.e., the inner product endows the space with a precise geometry which is reflected into our computations. Consider the following example taken from [BEJ25]:

$$f(\mathbf{x}) = \mathbf{x}^\top \mathbf{A}\mathbf{x} \tag{E.1.38}$$

Under the weighted inner product (E.1.25) we obtain:

$$\nabla f(\mathbf{x}) = \mathbf{W}^{-1}(\mathbf{A} + \mathbf{A}^\top)\mathbf{x} \tag{E.1.39}$$

which has a contribution from the inverse metric used in the inner product.

Inner products and optimization

As another application of the same ideas, consider optimizing a function $f(\mathbf{x})$. Most optimization algorithms can be framed as creating a surrogate function $\hat{f}(\mathbf{x};\mathbf{x}_0)$ around the current estimate $\mathbf{x}_0$ and optimizing for $\mathbf{x}$ [BR24]:

$$\mathbf{x} = \arg\min_{\mathbf{x}} \hat{f}(\mathbf{x};\mathbf{x}_0) \tag{E.1.40}$$

In the simplest case we can consider a (regularized) linear approximation:

$$\hat{f}(\mathbf{x};\mathbf{x}_0) = f(\mathbf{x}_0) + \nabla f(\mathbf{x}_0)^\top(\mathbf{x}-\mathbf{x}_0) + \eta\|\mathbf{x}-\mathbf{x}_0\|^2 \tag{E.1.41}$$

where η is known as the *sharpness* parameter and penalizes large deviations from the point of approximation. If we take the standard Euclidean norm $\|\mathbf{x}\| = \mathbf{x}^\top\mathbf{x}$, equating the gradient to zero we obtain one iteration of gradient descent:

$$\mathbf{x} = \mathbf{x}_0 - \frac{1}{2\eta}\nabla f(\mathbf{x}_0) \tag{E.1.42}$$

But if we consider a weighted inner product (for some symmetric $\mathbf{W}$) we obtain a *preconditioned* gradient descent iteration:

$$\mathbf{x} = -\frac{1}{2\eta}\mathbf{W}^{-1}\nabla f(\mathbf{x}_0) \tag{E.1.43}$$

The interplay between geometry of the data and optimization is fundamental in many sub-fields, from natural gradient descent [Mar20] to recent work on modular optimization [BN24].

Examples of computing the JVP

Let us make a brief pause to see how we would compute JVPs in practice. Consider for example a sum operation in NumPy:

```
y = x.sum(axis=i)
```

For 1D or 2D arrays, we can represent this operation in standard form as, e.g., $\mathbf{X}\mathbf{1}$ to sum over columns, with $\mathbf{1}$ a column vector of ones of the appropriate shape. For a generic array, however, it is better to use a simpler index notation, e.g., for a 3D array where we sum over the second dimension we have:

$$y_{o_1 o_2} = \sum_{i_2} x_{o_1 i_2 o_3} \tag{E.1.44}$$

where from here onwards we use i, i_2, i_3 to index the input dimensions and o_1, o_2 to index the output dimensions.[5] We see that materializing the Jacobian here requires a 5D matrix (2 dimensions for the output, 3 for the input), satisfying:

$$J_{o_1 o_2 i_1 i_2 i_3} = \frac{\partial y_{o_1 o_2}}{\partial x_{i_1 i_2 i_3}} = \begin{cases} 1 & \text{if } o_1 = i_1 \text{ and } o_2 = i_3 \\ 0 & \text{otherwise} \end{cases}$$

Clearly, building this tensor in memory would be a colossal waste of space! However, since most of the values of J are

[5] Index notation should be familiar if you remember `einsum` from Volume 1. Scientists from different disciplines all have their own index notations, such as upper or lower indices to distinguish vector components from covector ones (https://en.wikipedia.org/wiki/Index_notation).

zero, applying the tensor on an input vector u drastically simplifies:

$$(\partial f(x)[u])_{o_1 o_2} = \sum_{i_1, i_2, i_3} J_{o_1 o_2 i_1 i_2 i_3} u_{i_1 i_2 i_3} = \sum_{i_2} u_{o_1 i_2 o_2} \quad \text{(E.1.45)}$$

where we use colors to show all the indices that have to match in order for J to be different than 0. The result shows that the JVP in this case is a sum over u on the same index as the original operation.

Working with many indices may seem confusing at first, but it is the natural generalization of standard matrix operations to higher-dimensional arrays. We can see this by noting that for one input and one output dimensions, applying the Jacobian simplifies to:

$$(\partial f(x)[u])_{o_1} = \sum_{i_1} J_{o_1 i_1} u_{i_1} \quad \text{(E.1.46)}$$

which is just a matrix-vector product. Formally, operations like (E.1.45) are called *contractions* of the (tensor) Jacobian over the input dimensions.

In some cases, we can also compute JVPs by cleverly applying the definition of derivative. For example, consider again the function $f(\mathbf{X}) = \mathbf{X}^2$, and apply the definition of perturbation:[6]

$$(\mathbf{X} + \mathbf{U})^2 = \mathbf{X}^2 + \mathbf{U}^2 + \mathbf{XU} + \mathbf{UX} \quad \text{(E.1.47)}$$

[6]This is slightly different from the square $(a + b)^2$ of two scalars since matrix multiplication does not commute.

Since we are considering infinitesimal perturbations we can ignore “second-order effects" like $\mathbf{U}^2$ (if you are interested, we can formalize this intuition with the notion of infinitesimals [BEJ25]), and we obtain:

$$\partial f(\mathbf{X})[\mathbf{U}] = f(\mathbf{X}+\mathbf{U}) - f(\mathbf{X}) = \mathbf{XU} + \mathbf{UX} \tag{E.1.48}$$

recovering (E.1.20).

Adjoint maps and vector-Jacobian products

Now that we are familiar with the JVP, we switch to considering the VJP. In the matrix case, we define the VJP in terms of the transposed Jacobian matrix (ignore the notation on the left for now):

$$\partial f(\mathbf{x})^*[\mathbf{v}] = \mathbf{J}^\top \mathbf{v} \tag{E.1.49}$$

where $\mathbf{J}$ is the Jacobian around $\mathbf{x}$, $\mathbf{v}$ is now a vector from the output space, and the VJP returns a vector from the input space. Thus, the signature of the VJP has the last two dimensions reversed:

$$\partial f(\bullet)^* : \mathscr{E} \to (\mathscr{F} \to \mathscr{E}) \tag{E.1.50}$$

In the general case we cannot trivially define a transpose matrix, but we can do something similar by introducing the so-called **adjoint** of a map. First, note that in (E.1.45) we can define an operation satisfying this signature by simply contracting over the *output* dimensions:

$$\sum_{o_1,o_2} J_{o_1 o_2 i_1 i_2 i_3} v_{o_1 o_2} = v_{i_1 i_3} \tag{E.1.51}$$

The output is now a 3D matrix with the v vector broadcasted over the second axis. This turns out to be the actual VJP of the function, and it is equivalent to the **adjoint** itself.

Definition D.1.8 (Adjoint map)

Given a linear map $l[u] : \mathcal{E} \rightarrow \mathcal{F}$, its adjoint is defined as the map $l^*[v] : \mathcal{F} \rightarrow \mathcal{E}$ (with inverted domain and codomain) such that:

$$\langle l[u], v \rangle = \langle u, l^*[v] \rangle \tag{E.1.52}$$

The adjoint map is guaranteed to exist (and to be unique) for all finite-dimensional spaces, and for Hilbert spaces more in general.[a]

[a] The existence can be seen by considering the linear functional $m[u] = \langle l[u], v \rangle$, and then applying the Riesz representation theorem to deduce this must be equivalent to $\langle u, l^*[v] \rangle$ for some $l^*[v]$.

If the map is represented as a 2D matrix, the adjoint is exactly its transpose:[7]

$$\langle \mathbf{Au}, \mathbf{v} \rangle = \langle \mathbf{u}, \mathbf{A}^\top \mathbf{v} \rangle \tag{E.1.53}$$

In general, however, the adjoint of a map is another map

[7] This holds if we consider the standard inner product. For a weighted inner product we need to have $(\mathbf{Au})^\top \mathbf{Wv} = \mathbf{u}^\top \mathbf{W}(\mathbf{A}^* \mathbf{v})$, and re-arranging we obtain $\mathbf{A}^* = \mathbf{W}^{-1} \mathbf{A}^\top \mathbf{W}$.

having inverted domain and co-domain. This motivates the following definition.

Important

Definition D.1.9 (VJP, general vector space)

Given a differentiable function $f(x)$ (in the sense of Definition D.1.6), we define the ***vector-Jacobian product*** *(VJP) $\partial f(x)^*[v]$ as the adjoint of the JVP.*

In many senses, the VJP is the symmetric parent of the JVP. As an example, we can materialize the corresponding map by calling it over the basis elements $\partial f(x)^*[e_i]$, except that e_i is now a basis for the *output* vector space. Thus, for a function with n input dimensions and m output dimensions, we can materialize its Jacobian equivalently with n calls to the JVP or m calls to the VJP. Since in practice we work with programs having a single output (the scalar to be optimized), the VJP will play a fundamental role in the sections to come.

As another example of symmetry, the VJP can also be defined from first principles by considering infinitesimals variations of the output of the function [BR24]:

$$\partial f(x)^*[v] = \nabla \langle f(x), v \rangle \tag{E.1.54}$$

and then proceeding to define the JVP as the adjoint of the VJP itself. As a final note, both the JVP and the VJP admit a formulation of the chain rule. Considering the former as example, we have:

$$\partial (f \circ g)(x)[u] = \partial f(g(x))[\partial g(x)[u]] \tag{E.1.55}$$

and we can compute the JVP of the composition $F(x) =$

$(f \circ g)(x)$ with only access to the JVPs of f and g.[8]

1.3 Building the autodiff framework

We now proceed to sketch how an autodiff framework can be designed by building on the theory of JVPs/VJPs. The content re-elaborates on the material from Volume 1, although in a more general form and with some additional content.

We assume we have available some NumPy-like library giving access to a set of *primitive* functions $f_1, f_2, \ldots f_n$ operating on multi-dimensional arrays. What we consider as a primitive depends on the context. For example, convolution may be a primitive in one implementation, and implemented via a standard matrix multiplication on another. From our point of view, primitives are functions for which JVPs and/or VJPs are hard-coded and from which we can build more complex computational graphs by function composition.

The first task in implementing the system is to augment (*overload*) the execution of each primitive with a mechanism to recover the corresponding JVP or VJP:[9]

[8] In linear algebra, algorithms that can be run without materializing the corresponding matrices are known as *matrix-free*. Lanczos algorithm for computing eigenvalues is a classical example.

[9] Operator overloading is only one way to implement an autodiff framework, and other techniques (e.g., source-to-source transformations) are possible. In this chapter we focus on operator overloading because it is the basic mechanism underlying both PyTorch and JAX. See the first autodiff lab for a guided implementation: http://tinyurl.com/guided-labs.

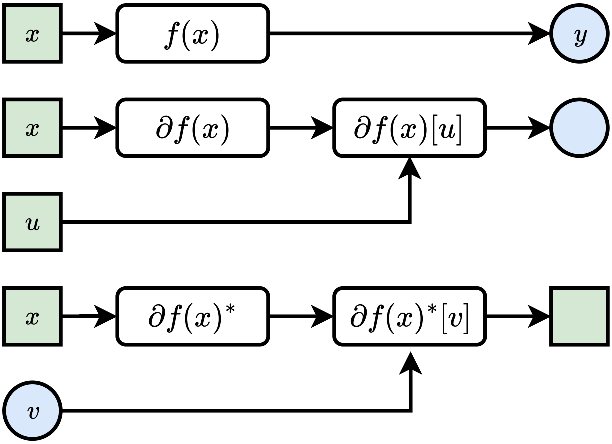

Figure F.1.3: *The three basic objects of an autodiff system: the original function f, its JVP $\partial f(x)[u]$, and the corresponding adjoint VJP $\partial f(x)^*[v]$. For visualization we show elements of the input space with green squares, and elements of the output space with blue circles.*

$$(x, \overbrace{u, v}^{\text{Additional inputs}}) \to (f(x), \underbrace{\partial f(x)[u], \partial f(x)^*[v]}_{\text{Additional outputs}}) \tag{E.1.56}$$

As we will see, depending on the application we may not need to have both implemented.[10] The overall setup is summarized in Figure F.1.3.

Forward-mode automatic differentiation

Suppose we take two primitives out of our set, and we chain them in sequence (the discussion trivially extends to

[10] In fact, since the derivative is a linear map, it is possible to recover the JVP [VJP] by repeated application of the JVP [VJP], by a mechanism known as **automatic linear transposition** [RPF+23, BR24].

longer sequences due to the locality of the chain rule):

$$F(x) = (f_2 \circ f_1)(x) \tag{E.1.57}$$

From a matrix point-of-view, each operation has an associated Jacobian, and we can compute the complete derivative by matrix multiplication. From our new perspective, instead, there are only two choices: we can use the JVP maps to propagate an input direction u forward, or we can use the VJP maps to propagate an output direction v backward.

Considering the former case, we select an input direction u before execution, and we simply call in sequence the JVPs of f_1 and f_2:

$$\begin{aligned} u &\leftarrow \partial f_1(x)[u] \\ u &\leftarrow \partial f_2(h)[u] \end{aligned} \tag{E.1.58}$$

where we defined $h = f_1(x)$, and we use the symbol $\leftarrow$ to highlight that u is overwritten at each step. As a result, we can execute both the original set of instructions (E.1.57) and the JVP calls (E.1.58) in parallel by a simple overloading of all primitives:

$$(x, u) \rightarrow (f(x), \partial f(x)[u]) \tag{E.1.59}$$

Substituting each call of a primitive with the overloaded variant returns both $F(x)$ and the complete JVP, as shown schematically in Figure F.1.4:

$$(x, u) \rightarrow (F(x), \partial F(x)[u]) \tag{E.1.60}$$

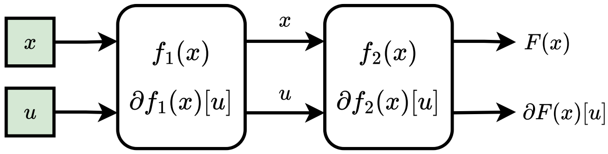

Figure F.1.4: *Forward-mode automatic differentiation with operator overloading: each call is replaced by the overloaded variant in* (E.1.59)*. As a result, the new program returns both $F(x)$ and its directional derivative $\partial F(x)[u]$ for a chosen u.*

This is of course **forward-mode** autodiff.[11] It can be shown that under reasonable assumptions on the primitives, the cost of executing a JVP is upper-bounded by a small multiple of the cost of the primitives. As a result, computing $\partial F(x)[u]$ is also upper-bounded in the same way. Memory also has a (roughly) overhead of 2x given by the need to store the vector u at each point.

In most cases we do not need a directional derivative but the full Jacobian of F, which we can materialize by repeated application of (E.1.60) on the input basis vectors, thus at a cost that grows linearly in the number of input dimensions (i.e., Jacobian columns in the standard matrix case).

Reverse-mode automatic differentiation

The reverse case is only slightly more involved. We first choose an *output* direction v, and we propagate it *in reverse*

[11]Paired objects as in (E.1.59) recall complex numbers. In fact, forward-mode autodiff can be formulated in terms of a different hypercomplex algebra known as **dual numbers** having an imaginary unit ε with $\varepsilon^2 = 0$ [BPRS18]. Rules for derivative compositions can be mapped to the rules for manipulating dual numbers, and as a result executing $F(x+\varepsilon u)$ returns $F(x)+\varepsilon\partial F(x)[u]$.

over the sequence of VJPs:

$$v \leftarrow \partial f_2(h)^*[v] \tag{E.1.61}$$
$$v \leftarrow \partial f_1(x)^*[v] \tag{E.1.62}$$

This operation, known as **reverse-mode** autodiff, returns $F(x)^*[v]$. Because the operations are now run in the reverse order, we cannot trivially augment the original primitives. What we can do instead is overload each primitive to return an additional VJP *function*:

$$x \rightarrow (f(x), \partial f(x)^*) \tag{E.1.63}$$

These functions are collected during the program execution (or *forward pass* in neural networks terminology), and afterwards they can be chained to compute the output (the *reverse pass*) – see Figure F.1.5. In programming terms, the collection of VJP functions is a **closure** associated to the execution of $F(x)$. The cost of a single reverse-mode call is also upper-bounded by a factor of $3-4$ with respect to the original cost. The memory cost, however, grows approximately linearly in the number of operations due to the need to store everything that is required to run the VJPs $\partial f(x)^*$ in the reverse pass.

The advantage of the reverse-mode is that materializing the Jacobian is achieved by running the VJP $F(x)^*[e_i]$ on all *output* basis vectors. Since in practice we tend to have a single output, we have $F(x)^*[a] = a\nabla F(x)$. Thus, running $\partial F(x)^*[1.0]$ just *once* returns the full gradient vector, at a cost of a potentially very large memory footprint.[12]

[12]The cost is only approximately linear because how much data we need to define a single VJP $\partial f(x)^*$ depends on the primitive. As an

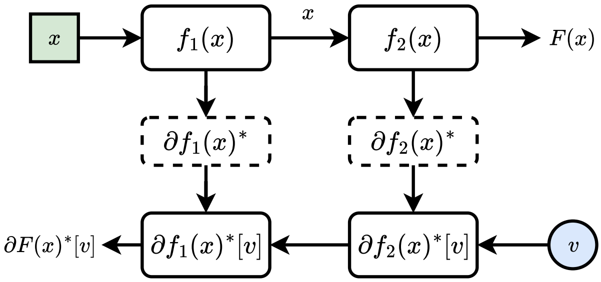

Figure F.1.5: *Reverse-mode automatic differentiation with operator overloading: each call returns the prepared VJP functions (dashed lines). These are collected and subsequently applied in reverse over an output direction v to return $\partial F(x)^*[v]$.*

1.3.1 Extending to computational graphs

The previous discussion is enough to implement a working AD system for computational chains (sequences), which is where we stopped in Volume 1. In order to handle generic directed acyclic graphs, thanks to the locality of the chain rule we only need to discuss two modifications: firstly, how to handle nodes having multiple *inputs*, and secondly, how to handle nodes with multiple *outputs*. We consider the reverse-mode scenario for simplicity, with the forward-mode case being completely symmetric.

First of all, consider a primitive function having two (or more) inputs:

$$y = f(x_1, x_2) \tag{E.1.64}$$

---

example, the ReLU function $f(x) = \max(0, x)$ has as Jacobian matrix $\text{diag}(\mathbb{I}_{x\geq 0})$, and as a result $\partial f(x)^*[v] = v \odot \mathbb{I}_{x\geq 0}$. This can be computed by directly storing the output of $f(x)$ (which explains the `in_place` flag in PyTorch's implementation) or the corresponding binary mask.

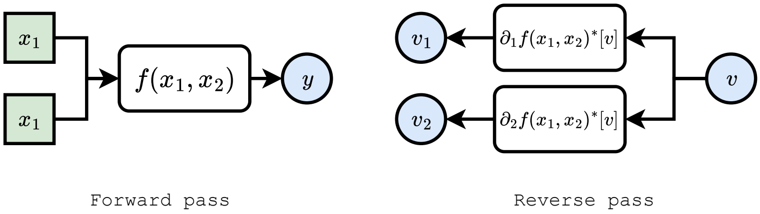

Figure F.1.6: *Reverse-mode AD on a node with two inputs. The adjoint value is propagated on the two paths separately after being updated through the corresponding VJP operation.*

This can represent, for example, matrix-vector multiplication $f(\mathbf{x}, \mathbf{W}) = \mathbf{W}\mathbf{x}$ where both terms are now being differentiated. The two inputs can be leaf nodes in the graph (constants, trainable parameters), or functions of previous computations.

We can associate two VJPs to the function by fixing one argument and differentiating the other, which we denote by a subscript $\partial_1 f(x_1, \bullet)^*[v]$, $\partial_2 f(x_2, \bullet)^*[v]$. During the backward pass we receive an adjoint value v and we call both VJPs:

$$v_1 = \partial_1 f(x_1, x_2)^*[v] \tag{E.1.65}$$

$$v_2 = \partial_2 f(x_1, x_2)^*[v] \tag{E.1.66}$$

We then continue reverse-mode AD on each branch using the corresponding value v_1 or v_2 – see Figure F.1.6. In Volume 1 we discussed the case of a *parametric* layer $f(x, w)$, where w represents the trainable parameter ($f(\mathbf{x}, \mathbf{W}) = \mathbf{W}\mathbf{x}$ being a simple example), and we referred to the matrices $\partial_1 f(x_1, x_2)$ and $\partial_2 f(x_1, x_2)$ as the *input Jacobian* and *weight Jacobian*, respectively. In that case, v_1 would continue on the reverse path, while v_2 is the necessary gradient for w, w being directly a leaf node.

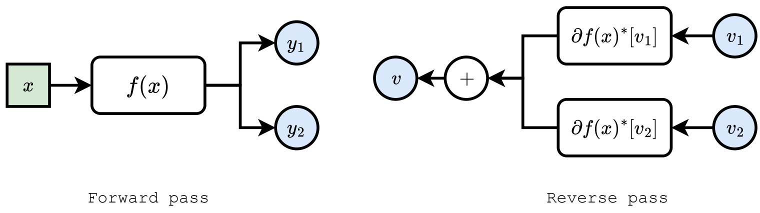

Figure F.1.7: *Reverse-mode AD on a node with two outputs (or the same output reused twice). In this case we receive two adjoint values (from the two children nodes). Both are updated with the VJP of the node and the result is accumulated.*

Consider now the symmetric case of a primitive function with two outputs:

$$y_1, y_2 = f(x) \tag{E.1.67}$$

As an edge case, y_1 and y_2 can be the same output reused twice in the graph (e.g., a residual connection). In that case, with a slight abuse of notation we can also make $f(x)$ a leaf node that we use twice (e.g., weight sharing).[13] In this case, during the backward pass we *receive* two adjoint values v_1 and v_2, and by linearity the final output is the *sum* of the two contributions (Figure F.1.7):

$$v = \partial f(x)^*[v_1] + \partial f(x)^*[v_2] \tag{E.1.68}$$

As a result, for an efficient implementation of reverse-mode AD we require that v is computed in full before being propagated to its parents in the computational graph - stated differently, the VJPs must be called in a **reverse topological ordering** of the nodes in the computational graph.

[13] See [BR24] for a more precise formalization.

1.4 Advanced topics

The previous section concludes the core part of the chapter. We now consider a selection of two advanced topics that are useful in practice: implicit differentiation (Section 1.4.1) and second-order differentiation (Section 1.4.2). We close with a general digression on tensor notation.

1.4.1 Implicit differentiation

Implicit differentiation is a powerful example of how an autodiff system can be extended in a non-trivial way [BBC+22]. In particular, in this section we sketch in good detail the implementation of a **deep equilibrium** (DEQ) layer [BKK19], a family of neural network layers that are equivalent to *infinite* recursion over the same core layer. We stress that implicit differentiation is a huge field with many applications, and we provide only a basic outline - see [BR24, Chapter 11] for a more complete treatment.

All the primitive functions we discussed up to this point are *explicit*, in the sense that we can write their output $y = f(x)$ in an explicit form. Implicit primitives, by contrast, define the output only indirectly as the vector satisfying a given constraint.

Important

Definition D.1.10 (Implicit function)

*Given an input $x \in \mathcal{E}$, an **implicit function** outputs a vector $y \in \mathcal{F}$ satisfying an equation:*

$$f(x, y) = 0 \tag{E.1.69}$$

> *where we assume f is differentiable w.r.t. x (i.e., we know how to compute its VJP or JVP), but we cannot provide an explicit function $y(x)$ solving the previous equation.*

This is an extremely general setup that subsumes many applications. For example, any operation defined as the minimization of a given optimization problem can be reframed as (E.1.69) by considering its stationarity conditions, including convex optimization layers [AK17, AAB+19], or bilevel optimization for hyper-parameter optimization [LVD20].

As a running example, consider a recurrent layer (such as those we covered in Chapter 13 in Volume 1):

$$y_{i+1} = g(x_i, y_i) \tag{E.1.70}$$

where y is the state vector and x the input vector. Suppose we fix the input $x_i = x$ at all time-steps. Absent the input, this is equivalent to iterating the *same* layer multiple times, which can be used to, e.g., increase inference-time compute of a pre-existing model [BKB+25]. If we run the layer *infinite* times, assuming the dynamics of the layer is contractive we reach a **fixed-point** of the recursion:

$$y^* = g(x, y^*) \tag{E.1.71}$$

A layer parameterized implicitly as in (E.1.71) is known as a DEQ layer [BKK19]. We can easily rewrite it in the form of (E.1.69) as $f(x, y) = g(x, y) - y = 0$.

Solving fixed-point equations of this form is not inherently difficult. For example, the code in Box C.1.1 finds a solution

```python
while norm(y - f(x, y)) > eps:
    y = f(x, y)
```

Box C.1.1: *Basic fixed-point solver, where* `eps` *represents a tolerance factor.*

up to a pre-specified tolerance by simply iterating the fixed-point equation.

Running the code in Box C.1.1 produces a perfectly valid computational graph that we can differentiate, at the cost (in reverse-mode) of having a memory overhead that grows linearly in the number of iterations. Worse, advanced fixed-point solvers may not be so easily differentiable (e.g., if implemented in standard NumPy). Luckily, we now show how implicit differentiation provides a more elegant (and scalable) solution.

Informally, if we can find a fixed-point y, the **implicit function theorem** asserts that under loose conditions on $f(x,y)$ there exists a function $y(x)$ such that $y = y(x)$ and $y(x)$ in differentiable in the neighborhood of y. Practically, it means that we can differentiate the implicit equation to obtain:

$$\overbrace{\partial_2 f(x,y)}^{\text{Jacobian of } f \text{ w.r.t. } y}\;\overbrace{\partial y(x)}^{\text{Unknown}} + \underbrace{\partial_1 f(x,y)}_{\text{Jacobian of } f \text{ w.r.t. } x} = 0 \tag{E.1.72}$$

If we find a fixed-point y (irrespective of how), $\partial_1 f(x,y)$ and $\partial_1 f(x,y)$ can be materialized by standard autodiff, and the unknown gradient in (E.1.72) can be found by solving a system of linear equations as $\partial y(x) = -(\partial_2 f(x,y))^{-1}\,\partial_1 f(x,y)$. However, in practice we do not want to materialize these arrays. Instead, what we need is a way to compute the VJP $\partial y(x)^*[v]$, which would allows us to integrate the implicit equation as a primitive inside any existing autodiff framework.

To simplify the notation, we define $A = \partial_2 f(x,y)$ and $B = -\partial_1 f(x,y)$, from which $\partial y(x) = A^{-1}B$. Recall that for two linear operators l_1 and l_2, $(l_1 l_2)^* = l_2^* l_1^*$ (you can also consider the equivalent statement in terms of matrix transposes if you feel more comfortable). Hence:

$$\partial y(x)^*[v] = B^*(A^{-1})^*[v] \tag{E.1.73}$$

This is still not optimal because it requires knowledge of the adjoint of the inverse of A, but practical solutions can be obtained on a case-by-case basis.

For example, consider again the case of the DEQ layer and focus on the first operation $t = (A^{-1})^*[v]$. Substituting $g(x,y) = f(x,y) - y$ we obtain:

$$t = (I - A^*)^{-1}[v] \tag{E.1.74}$$

Left-multiplying again by $(I - A^*)$ we obtain *a second* fixed-point equation:

$$t = A^*[t] + v \tag{E.1.75}$$

This provides an interesting algorithm for implementing DEQ layers:

1. Solve the original fixed-point equation to find a solution y, using whatever fixed-point solver is available.

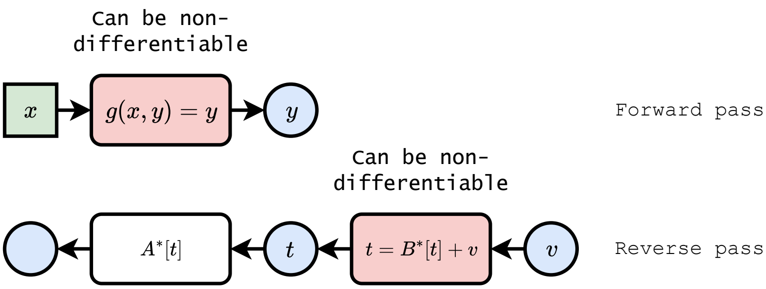

Figure F.1.8: *Forward pass and reverse pass for a DEQ layer, where the two (possibly non-differentiable) fixed-point layers are shown in red.*

2. During the backward pass, take the adjoint v and solve equation (E.1.75) for t.

3. Compute the (very standard) VJP $\partial_1 g(x, y)^*[t]$.

This procedure provides the required VJP $y(x)^*[v]$. Interestingly, steps (1) and (2) are decoupled and can be executed outside the computational graph, with no need to store intermediate quantities – see Figure F.1.8. In PyTorch, for example, step (2) can be implemented by attaching a so-called *backward hook* to the operation, pre-processing the adjoint value.[14]

1.4.2 Second-order differentiation

As a second advanced topic, we consider how the notation and concepts extend to second-order (and higher) derivatives. For a more complete introduction we refer to [BR24, Chapter 9].

[14]For a toy implementation, see lab PT02 online, or Chapter 4 from the Deep Implicit Layers tutorial of NeurIPS 2020: https://implicit-layers-tutorial.org.

Assuming that the derivative $\partial f(x)[u]$ exists, we can consider small perturbations of the form $\partial f(x+u')[u]$. Iterating our previous discussion, we can look for a *second* linear map, which we denote by $\partial^2 f(x)[u][u']$, to approximate infinitesimal perturbations. Since this is linear in both u and u', we can slightly simplify the notation and denote it as $\partial^2 f(x)[u,u']$ – this is an example of a **bilinear map**. We formalize this idea below.

Definition D.1.11 (Second-order derivative)

A function $f(x) : \mathscr{E} \rightarrow \mathscr{F}$ is said to be twice-differentiable if we can find a bilinear map $\partial^2 f(x)[u,u']$ such that:

$$\lim_{\|u'\|\rightarrow 0} \frac{\|\partial f(x+u')[u] - \partial f(x)[u] - \partial^2 f(x)[u,u']\|}{\|u'\|} = 0 \tag{E.1.76}$$

In programming terms, the signature of the second-order derivative is:

$$\partial^2 f : \mathscr{E} \rightarrow (\mathscr{E} \times \mathscr{E} \rightarrow \mathscr{F}) \tag{E.1.77}$$

This might look abstract, but it mirrors nicely our previous discussion. In particular, since $\partial^2 f(x)$ is a bilinear map, for finite-dimensional vector spaces we can write its materialization T as:

$$(\partial^2 f(x)[u,u'])_k = \sum_i \sum_j T_{kij} u_i u'_j \tag{E.1.78}$$

i.e., in general it will be a 3D (hyper)matrix. If we have access to the map, we can obtain the materialization by calling it n^2 times over all possible pairs of input basis vectors $\partial^2 f(x)[e_i, e_j]$. For the special case of $f(x) : \mathscr{E} \to \mathbb{R}$, by the Riesz representation theorem, if we fix u we can write the second-order derivative as the inner product of u' with a *second* linear map that we denote as $\nabla^2 f(x)[u]$:

$$\partial^2 f(x)[u, u'] = \langle u', \nabla^2 f(x)[u] \rangle \tag{E.1.79}$$

where $\nabla^2 f(x)[u]$ is the second-order generalization of the gradient. For $\mathscr{E} = \mathbb{R}^n$ we can write it as a matrix-vector product $\nabla^2 f(x)[u] = Hu$, with the matrix H called the **Hessian** of the function. By extension, we call $\nabla^2 f(x)[u]$ the **Hessian-vector product** (HVP) of the function.

Interestingly, since the Hessian is a symmetric matrix, we can implement the HVP map with any paired combination of forward- and reverse-mode, with varying computational complexities [BR24].[15] Intuitively, since the computational graph of reverse-mode is roughly twice the original one, the cost of a single HVP will be a constant overhead over the original graph [Pea94].[16]

It is possible in principle to extend the derivation to higher-order derivatives. For example, a third-order derivative becomes a *trilinear* map whose materialization is a 4D matrix, that can be materialized by calling the map on all possible *triplets* of basis vectors (e_i, e_j, e_z). For a scalar function, its contraction with a vector can be computed by composing forward- or reverse-mode *thrice*,

[15] Not all frameworks support chained application of the AD functions. To allow it, all operations specific to forward- or reverse-mode must be implemented using *the same* set of primitives as the original computational graph, trivially making the extended computational graph differentiable.

[16] If interested, see: https://iclr-blogposts.github.io/2024/blog/bench-hvp/.

and so forth. However, the growth in complexity of the underlying graph makes this approach unfeasible and generally less supported in most modern frameworks. For higher-order derivatives, a more efficient approach is to work on truncated Taylor series, a technique known as **Taylor-mode automatic differentiation** which is gaining traction lately [BJD19].

1.4.3 Tensors and multilinear algebra

Discursive

We mentioned in Volume 1 that the use of *tensors* to refer to multidimensional arrays (like in PyTorch) is a misnomer, but we did not pursue this discussion further, as tensors are generally considered a difficult concept to grasp. Nonetheless, we ended up encountering *a lot* of “real" tensors in this chapter - which is not surprising, since tensors are the mathematical object representing linear, multidimensional maps [Lim21]. We now introduce some interesting concepts from tensor algebra using the examples from this chapter as motivating material, referring the interested readers to [Lim21] for a broader overview. This section is a digression and can be safely skipped if you are not interested in the topic.

Throughout the chapter we have introduced many linear maps, which are characterized by the amount of indices appearing either on the input or the output vector. Listing a few of them with all indices made explicit:

1. Standard linear functionals on vectors $l[\mathbf{x}] = \sum_{i_1} v_{i_1} x_{i_1}$ with a single input index.

2. A bit abstractly, we can interpret vectors as functions from the set $\{1, \ldots, n\}$, so that the o_1-th output u_{o_1} is

the corresponding coordinate.[17]

3. Linear maps between vector spaces $(l[\mathbf{x}])_{o_1} = \sum_{i_1} A_{o_1 i_1} x_{i_1}$ have two indices, one on the input and one on the output.

4. Inner products over vectors $l[\mathbf{x}, \mathbf{y}] = \sum_{i_1, i_2} W_{i_1 i_2} x_{i_1} y_{i_2}$ also have two indices, both over the corresponding inputs.

5. Linear functionals over matrices $l[\mathbf{X}] = \sum_{i_1, i_2} W_{i_1 i_2} X_{i_1 i_2}$ have the same index structure as inner products.

6. By extension, inner products over matrices $l[\mathbf{A}, \mathbf{B}] = \sum_{i_1 i_2 i_3 i_4} W_{i_1 i_2 i_3 i_4} A_{i_1 i_2} B_{i_3 i_4}$ can be built out of 4 input indices.

All these objects are valid tensors in the mathematical sense, with the number of indices describing the **order** of the tensor. Case (1) is a $(1,0)$ tensor (one input index and no output indices), case (2) is a $(0,1)$ tensor, case (3) is a $(1,1)$ tensor, cases (4) and (5) are both $(2,0)$ tensors, and so on. This is summarized in Table T.1.2, where we also show how the index structure maps to an einsum-like format.

In fact, *both* sides of these equations represent valid tensors, either by considering the abstract linear maps (with no reference to a basis) or their multidimensional representation once all bases are selected (think of the gradient represented as either a linear map or a Jacobian – these are the *same* object).

[17]The set of functionals extracting the i-th coordinate of a vector form a basis of the dual space.

Description	Index Pattern	Type	Order
Linear functional on vectors	`i->`	$(1,0)$	1
Coordinate projection (dual basis)	`->j`	$(0,1)$	1
Linear map between vectors	`ij->j`	$(1,1)$	2
Inner product on vectors	`i,j->`	$(2,0)$	2
Linear functional on matrices	`ij->`	$(2,0)$	2
Inner product on matrices	`ij,kl->`	$(4,0)$	4

Table T.1.2: *Examples of linear maps summarized by index structure (einsum-style notation), tensor type, and order.*

While the maps are unique, their representation is not – it changes whenever we select a different basis. Thus, the same linear map can have infinite representations, one for each basis. Pair of bases are in general related by (possibly location-dependent) matrices, which are known as **change-of-basis** (CoB) matrices. These in turn define how the materialized tensors change whenever we modify the basis. For example, suppose the old basis vectors e_i and the new basis vectors e'_j are related by:

$$e'_i = C_{ij}e_j \tag{E.1.80}$$

Then, just to list a few examples, it is straightforward to show that a vector $\mathbf{u}$ will transform as $\mathbf{C}^{-1}\mathbf{u}$, a linear functional (identified as a row vector) $\mathbf{v}$ will transform as $\mathbf{C}^\top\mathbf{v}$, a matrix $\mathbf{A}$ will transform as $\mathbf{C}^\top\mathbf{A}\mathbf{C}^{-1}$,[18] an inner product matrix $\mathbf{W}$ will transform as $\mathbf{C}^\top\mathbf{W}\mathbf{C}$, and so on. You can see there is a correspondence between the order of a

[18]Assuming the same input and output space *and* the same CoB matrix on both sides.

tensor and the way its corresponding matrices transform: for a generic (m, p) tensor, the indices will transform p times with $\mathbf{C}$ and m times with its inverse $\mathbf{C}^{-1}$. In physics, these are known respectively as *covariant* and *contravariant* indices, which are expressed in a variety of notation including upper and lower indices (in relativity) and bra and ket notation (in quantum mechanics).

We can now amend our previous statement: a generic array is not a tensor, *unless* we specify the way its indices change under a CoB. To see this, note that $\mathbf{W}$ and $\mathbf{A}$ have the same number of indices, but they represent different types of tensors. As another example, consider a mini-batch $\mathbf{X}$ of input vectors organized row-wise (the standard convention in Volume 1). Under a CoB the matrix will modify as $\mathbf{X}\mathbf{C}^{-\top}$, which shows this is a $(1, 0)$ tensor despite having two indices!

Why haven't I see this before?

It turns out that most of these concepts do not matter if we restrict our attention to Euclidean spaces, where we generally use either the standard basis or a rotation of it. Since both of these are orthonormal bases, their corresponding CoB matrices are also orthogonal, and thus $\mathbf{C}^{-1} = \mathbf{C}^{\top}$. As a direct consequence, there is no practical distinction between row or column vectors, or between $\mathbf{W}$ and $\mathbf{A}$ in our previous examples. However, all these distinctions immediately resurface if we consider, for example, curvilinear coordinates.[a]

[a] https://en.wikipedia.org/wiki/Curvilinear_coordinates

Confusingly, different operations can be represented by the same tensor. For example, a matrix $\mathbf{W}$ can represent a map between two vector spaces, or it can represent a bilinear functional taking as input a row and a column vector (or in abstract linear algebra terminology, a vector and a covector). Practically, this means that the two corresponding spaces are isomorphic. While this is trivial to handle for two dimensions, the number of possibilities (and isomorphisms) grows combinatorially with the increase in the order of the tensors. This can be handled by introducing a third, more general definition of tensors in terms of **tensor products** [Lim21]. Very informally, a tensor product $\otimes$ is an operation that turns a generic multilinear operation into a linear operation over a different object, providing a way to quickly categorize all possible tensor isomorphisms. Just to sketch an example, the bilinear functional:

$$l[\mathbf{x},\mathbf{y}] = \mathbf{x}^\top \mathbf{W}\mathbf{y} \tag{E.1.81}$$

can be turned into a *linear* functional over the tensor product $\mathbf{x} \otimes \mathbf{y}$ (assuming $\mathbf{x}$ and $\mathbf{y}$ have the same dimensionality, so that $\mathbf{W}$ is square):

$$l[\mathbf{x} \otimes \mathbf{y}] = \langle \mathbf{W}, \mathbf{x} \otimes \mathbf{y} \rangle = \mathrm{Tr}(\mathbf{W}^\top \left(\mathbf{x}\mathbf{y}^\top\right)) \tag{E.1.82}$$

where the tensor product for two vectors is implemented as $\mathbf{x} \otimes \mathbf{y} = \mathbf{x}\mathbf{y}^\top$. Vector products have appeared "under the hood" in many places over the chapter,[19] but sadly covering it in detail would go (way) beyond the scope of this section.

[19] As an example, the space of linear maps as in (E.1.32) is isomorphic to the tensor product space $\mathcal{E} \otimes \mathcal{F}^*$.

Bibliography

[AAB+19] A. Agrawal, B. Amos, S. Barratt, S. Boyd, S. Diamond, and J. Z. Kolter. Differentiable convex optimization layers. *Advances in neural information processing systems*, 32, 2019. 34

[AK17] B. Amos and J. Z. Kolter. Optnet: Differentiable optimization as a layer in neural networks. In *International conference on machine learning*, pages 136–145. PMLR, 2017. 34

[Axl15] S. Axler. *Linear algebra done right*. Springer, 2015. 9

[BBC+22] M. Blondel, Q. Berthet, M. Cuturi, R. Frostig, S. Hoyer, F. Llinares-López, F. Pedregosa, and J.-P. Vert. Efficient and modular implicit differentiation. *Advances in neural information processing systems*, 35:5230–5242, 2022. 33

[BEJ25] P. Bright, A. Edelman, and S. G. Johnson. Matrix calculus (for machine learning and beyond). *arXiv preprint arXiv:2501.14787*, 2025. 1, 2, 8, 18, 22

[BJD19] J. Bettencourt, M. J. Johnson, and D. Duvenaud. Taylor-mode automatic differentiation for higher-order derivatives in jax. In *Program Transformations for ML Workshop at NeurIPS 2019*, 2019. 40

[BKB+25] S. Bae, Y. Kim, R. Bayat, S. Kim, J. Ha, T. Schuster, A. Fisch, H. Harutyunyan, Z. Ji, A. Courville, et al. Mixture-of-recursions: Learning dynamic recursive depths for adaptive token-level computation. *arXiv preprint arXiv:2507.10524*, 2025. 34

[BKK19] S. Bai, J. Z. Kolter, and V. Koltun. Deep equilibrium models. *Advances in neural information processing systems*, 32, 2019. 33, 34

[BN24] J. Bernstein and L. Newhouse. Modular duality in deep learning. *arXiv preprint arXiv:2410.21265*, 2024. 19

[BPRS18] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind. Automatic differentiation in machine learning: a survey. *Journal of machine learning research*, 18(153):1–43, 2018. 28

[BR24] M. Blondel and V. Roulet. The elements of differentiable programming. *arXiv preprint arXiv:2403.14606*, 2024. 1, 2, 11, 19, 24, 26, 32, 33, 37, 39

[Lim21] L.-H. Lim. Tensors in computations. *Acta Numerica*, 30:555–764, 2021. 15, 40, 44

[LVD20] J. Lorraine, P. Vicol, and D. Duvenaud. Optimizing millions of hyperparameters by implicit differentiation. In *International conference on artificial intelligence and statistics*, pages 1540–1552. PMLR, 2020. 34

[Mar20] J. Martens. New insights and perspectives on the natural gradient method. *Journal of Machine Learning Research*, 21(146):1–76, 2020. 19

[Pea94] B. A. Pearlmutter. Fast exact multiplication by the hessian. *Neural computation*, 6(1):147–160, 1994. 39

[RPF+23] A. Radul, A. Paszke, R. Frostig, M. J. Johnson, and D. Maclaurin. You only linearize once: Tangents transpose to gradients. *Proceedings of the ACM on Programming Languages*, 7(POPL):1246–1274, 2023. 26

[Rud87] W. Rudin. *Real and complex analysis*. McGraw-Hill, Inc., 1987. 14