B Derivatives and Automatic Differentiation

B.1 Introduction

The concept of the *derivative* from calculus is fundamental to forming an intuitive and practical understanding of local optimization techniques, which are the workhorse of machine learning. Because of its critical importance this appendix chapter provides a self-contained overview of derivatives, practical computation of derivatives via automatic differentiation (including an outline of the *backpropagation algorithm*), and Taylor series approximations.

B.2 The Derivative

The derivative is a simple tool for understanding a (continuous) mathematical function *locally*, meaning at and around a single point. In this section we review the notion of the derivative at a point, and how it naturally defines the best *linear approximation* (i.e., a line in two dimensions, a hyperplane in higher dimensions) that matches the given function at that point as well as a line/hyperplane can.

B.2.1 The secant and tangent lines

In Figure B.1 we illustrate a single-input function $g(w)$ along with the tangent line to this function at various points in its input domain. At any input point the *slope* of the tangent line to this function is referred to as its *derivative*. Notice at different input points how this quantity (i.e., the slope of the tangent line) seems to match the general local steepness of the underlying function itself. The derivative naturally encodes this information.

To formally define the derivative, we start by examining a *secant line* formed by taking any two points on a (single-input) function and connecting them with a straight line. The equation of any secant line constructed in this way is easy to find since its slope is given by the "rise over run" difference between the two points, and the equation of any line can be formed using its slope and any point on said line. For a generic single-input function g and the input points w^0 and w^1, the secant line h passes through the points $\left(w^0,\ g\left(w^0\right)\right)$ and $\left(w^1,\ g\left(w^1\right)\right)$, with its slope defined as

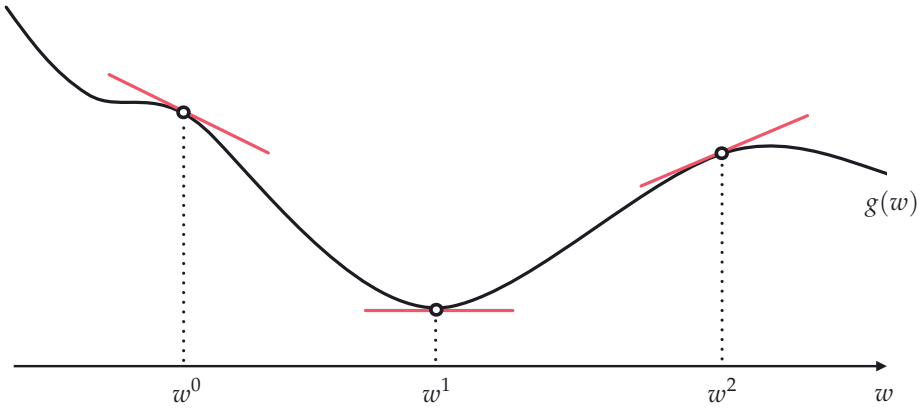

Figure B.1 A generic function with the tangent line drawn at various points of its input domain. See text for further details.

$$\frac{g(w^1) - g(w^0)}{w^1 - w^0} \tag{B.1}$$

and its equation defined as

$$h(w) = g(w^0) + \frac{g(w^1) - g(w^0)}{w^1 - w^0}(w - w^0). \tag{B.2}$$

Now imagine we fix the point w^0 and begin slowly pushing the point w^1 towards it. As w^1 gets closer and closer to w^0 the secant line resembles the tangent line at w^0 more and more. In other words, if we denote the difference between w^1 and w^0 by ϵ

$$w^1 = w^0 + \epsilon \tag{B.3}$$

then the slope of the secant line connecting $\left(w^0, g\left(w^0\right)\right)$ to $\left(w^1, g\left(w^1\right)\right)$, given as

$$\frac{g(w^1) - g(w^0)}{w^1 - w^0} = \frac{g(w^0 + \epsilon) - g(w^0)}{w^0 + \epsilon - w^0} = \frac{g(w^0 + \epsilon) - g(w^0)}{\epsilon} \tag{B.4}$$

matches the slope of the tangent line at w^0 (i.e., the derivative at w^0) as $|\epsilon|$ gets infinitesimally small. Notice that ϵ may be positive or negative here, depending on whether w^1 lies to the right or left of w^0. For the derivative to be defined at w^0 the quantity in Equation (B.4) needs to converge to the same value as ϵ approaches zero, regardless of the mathematical sign of ϵ (or equivalently, the direction in which w^1 approaches w^0). For example, the derivative at $w^0 = 0$ *is* defined for the function $g\,(w) = \max(0, w^2)$ since

$$\frac{g(w^0+\epsilon)-g(w^0)}{\epsilon} = \frac{\max(0,\epsilon^2)-\max(0,0)}{\epsilon} = \frac{\epsilon^2}{\epsilon} = \epsilon \tag{B.5}$$

which converges to zero as ϵ approaches zero. On the other hand, the derivative at $w^0 = 0$ is *not* defined for the function $g(w) = \max(0, w)$ since

$$\frac{g(w^0+\epsilon)-g(w^0)}{\epsilon} = \frac{\max(0,\epsilon)}{\epsilon} \tag{B.6}$$

equals zero when a negative ϵ approaches zero, and equals one when a positive ϵ approaches zero.

One common notation used to denote the derivative of g at the point w^0 is

$$\frac{\mathrm{d}g\left(w^0\right)}{\mathrm{d}w} \tag{B.7}$$

where the symbol d means "infinitely small change in the value of." Notice this is precisely what the fraction in Equation (B.4) expresses when $|\epsilon|$ is infinitesimally small. A common variation on this notation puts $g\left(w^0\right)$ out front, as

$$\frac{\mathrm{d}}{\mathrm{d}w}g\left(w^0\right). \tag{B.8}$$

There are also other notations commonly used in practice to denote the derivative of g at w^0, e.g., $g'\left(w^0\right)$. Finally, note that with this notation the equation of the tangent line to g at the point w^0 can be written as

$$h(w) = g(w^0) + \frac{\mathrm{d}}{\mathrm{d}w}g(w^0)(w-w^0). \tag{B.9}$$

B.2.2 Numerical differentiation

To avoid having to find the derivative expression in Equation (B.4) analytically, especially when the function g is rather complicated, we can create a derivative calculator that *estimates* its derivative at a user-defined point w^0 by simply setting ϵ to some small number (in absolute value) like $\epsilon = 0.0001$. To be more robust in our approximation of the derivative we can replace the "one-way" secant slope in Equation (B.8) with the average slope of the right and left secant lines, as

$$\frac{\mathrm{d}}{\mathrm{d}w}g\left(w^0\right) \approx \frac{g(w^0+\epsilon)-g(w^0-\epsilon)}{2\epsilon}. \tag{B.10}$$

In either case the smaller we set ϵ the better our estimation of the actual derivative value becomes. However, setting ϵ too small can create *round-off errors* due to the fact that numerical values (whether or not they are produced from a mathematical function) can be represented only up to a certain precision

on a computer, and that both the numerator and denominator in Equations (B.4) and (B.10) shrink to zero rapidly. This numerical stability issue does not completely invalidate numerical differentiation, but is worth being aware of.

B.3 Derivative Rules for Elementary Functions and Operations

For certain elementary functions and operations we need not resort to numerical differentiation since finding their exact derivative value is quite straightforward. We organize derivative formulae for popular *elementary functions* and *operations* in Tables B.1 and B.2, respectively. We provide formal proof for one elementary function from Table B.1 (monomial of degree d) and one elementary operation from Table B.2 (multiplication) in Examples B.1 and B.2, respectively. One can easily confirm the remainder of the derivative rules by following a similar argument, or consulting any standard calculus reference.

Table B.1 Derivative formulae for elementary functions.

Elementary function	Equation	Derivative
Sine	$\sin(w)$	$\cos(w)$
Cosine	$\cos(w)$	$-\sin(w)$
Exponential	e^w	e^w
Logarithm	$\log(w)$	$\frac{1}{w}$
Hyperbolic tangent	$\tanh(w)$	$1 - \tanh^2(w)$
Rectified Linear Unit (ReLU)	$\max(0, w)$	$\begin{cases} 0 & \text{if } w < 0 \\ 1 & \text{if } w > 0 \end{cases}$

Table B.2 Derivative formulae for elementary operations.

Elementary operation	Equation	Derivative
Addition of a constant	$c + g(w)$	$\frac{\mathrm{d}}{\mathrm{d}w} g(w)$
Multiplication by a constant	$c\, g(w)$	$c \frac{\mathrm{d}}{\mathrm{d}w} g(w)$
Addition of functions	$f(w) + g(w)$	$\frac{\mathrm{d}}{\mathrm{d}w} f(w) + \frac{\mathrm{d}}{\mathrm{d}w} g(w)$
Multiplication of functions	$f(w)\ g(w)$	$\left[\frac{\mathrm{d}}{\mathrm{d}w} f(w)\right] g(w) + f(w) \left[\frac{\mathrm{d}}{\mathrm{d}w} g(w)\right]$
Composition of functions	$f(g(w))$	$\frac{\mathrm{d}}{\mathrm{d}g} f(g) \frac{\mathrm{d}}{\mathrm{d}w} g(w)$

Example B.1 Derivative of general monomial terms
Starting with a degree-two monomial $g(w) = w^2$, we can write for general w and small ϵ

$$\frac{g(w+\epsilon) - g(w)}{\epsilon} = \frac{(w+\epsilon)^2 - w^2}{\epsilon} = \frac{(w^2 + 2w\epsilon + \epsilon^2) - w^2}{\epsilon} = \frac{2w\epsilon + \epsilon^2}{\epsilon} = 2w + \epsilon. \tag{B.11}$$

Sending ϵ to zero we get

$$\frac{\mathrm{d}}{\mathrm{d}w} g(w) = 2w. \tag{B.12}$$

Now let us examine a general degree-d monomial $g(w) = w^d$. All we need to do here is expand $(w+\epsilon)^d$ and rearrange its terms appropriately, as

$$(w+\epsilon)^d = \sum_{j=0}^{d} \binom{d}{j} \epsilon^j \, w^{d-j} = w^d + d\,\epsilon\, w^{d-1} + \epsilon^2 \sum_{j=2}^{d} \binom{d}{j} \epsilon^{j-2} \, w^{d-j} \tag{B.13}$$

where

$$\binom{d}{j} = \frac{d!}{j!(d-j)!}. \tag{B.14}$$

Plugging this expansion into the definition of the derivative

$$\frac{(w+\epsilon)^d - w^d}{\epsilon} = d w^{d-1} + \epsilon \sum_{j=2}^{d} \binom{d}{j} \epsilon^{j-2} \, w^{d-j} \tag{B.15}$$

we can see that the second term on the right-hand side vanishes as $\epsilon \longrightarrow 0$.

Example B.2 The product rule
With two functions $f(w)$ and $g(w)$ we want to evaluate

$$\frac{f(w+\epsilon)g(w+\epsilon) - f(w)g(w)}{\epsilon} \tag{B.16}$$

as ϵ approaches zero. Adding and subtracting $f(w+\epsilon)g(w)$ in the numerator gives

$$\frac{f(w+\epsilon)g(w+\epsilon) - f(w+\epsilon)g(w) + f(w+\epsilon)g(w) - f(w)g(w)}{\epsilon} \tag{B.17}$$

which then simplifies to

$$\frac{f(w+\epsilon) - f(w)}{\epsilon}\, g(w) \;+\; f(w+\epsilon)\, \frac{g(w+\epsilon) - g(w)}{\epsilon}. \tag{B.18}$$

Notice as $\epsilon \longrightarrow 0$, the first and second term in Equation (B.18) goes to $\left[\frac{\mathrm{d}}{\mathrm{d}w}f(w)\right]g(w)$ and $f(w)\left[\frac{\mathrm{d}}{\mathrm{d}w}g(w)\right]$, respectively, together giving

$$\frac{\mathrm{d}}{\mathrm{d}w}\left[f\left(w\right)\ g\left(w\right)\right] = \left[\frac{\mathrm{d}}{\mathrm{d}w}f\left(w\right)\right]g\left(w\right) + f\left(w\right)\left[\frac{\mathrm{d}}{\mathrm{d}w}g\left(w\right)\right]. \tag{B.19}$$

B.4 The Gradient

The *gradient* is a straightforward generalization of the notion of derivative for a multi-input function $g\left(w_1, w_2, \ldots, w_N\right)$. Treating all inputs but the first one (i.e., w_1) as fixed values (not *variables*), the function g momentarily reduces to a single-input function for which we have seen how to define the derivative (with respect to its only input variable w_1). This *partial* derivative, written as $\frac{\partial}{\partial w_1}g\left(w_1, w_2, \ldots, w_N\right)$, determines the slope of the hyperplane tangent to g at a given point, along the first input dimension. Repeating this for all inputs to g gives N partial derivatives (one along each input dimension) that collectively define the set of slopes of the *tangent hyperplane*. This is completely analogous to the single-input case where the derivative provides the slope of the *tangent line*.

For notational convenience these partial derivatives are typically collected into a vector, called the *gradient* and denoted by $\nabla g\left(w_1, w_2, \ldots, w_N\right)$, as

$$\nabla g\left(w_1, w_2, \ldots, w_N\right) = \begin{bmatrix} \frac{\partial}{\partial w_1}g\left(w_1, w_2, \ldots, w_N\right) \\ \frac{\partial}{\partial w_2}g\left(w_1, w_2, \ldots, w_N\right) \\ \vdots \\ \frac{\partial}{\partial w_N}g\left(w_1, w_2, \ldots, w_N\right) \end{bmatrix}. \tag{B.20}$$

Stacking all N inputs (w_1 through w_N) similarly into a column vector

$$\mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_N \end{bmatrix}, \tag{B.21}$$

the hyperplane tangent to the function $g\left(\mathbf{w}\right)$ at the point $\left(\mathbf{w}^0, g\left(\mathbf{w}^0\right)\right)$ can be compactly characterized as

$$h(\mathbf{w}) = g(\mathbf{w}^0) + \nabla g(\mathbf{w}^0)^T(\mathbf{w} - \mathbf{w}^0), \tag{B.22}$$

which is the direct generalization in higher dimensions of the formula for a tangent line defined by the derivative of a single-input function given in Equation (B.9), as illustrated in Figure B.2.

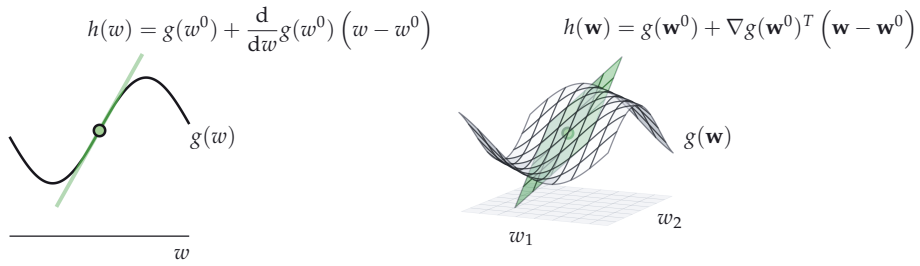

Figure B.2 (left panel) The derivative of a single-input function defines the slope of the tangent line at a point. (right panel) The gradient of a multi-input function analogously defines the set of slopes of the tangent hyperplane at a point. Here $N = 2$. In each panel the point of tangency is highlighted on the function as a green circle.

B.5 The Computation Graph

Virtually any function g expressed via an algebraic formula can be broken down (akin to the way physical substances may be broken down into their atomic parts) into a combination of elementary functions (e.g., $\sin(\cdot)$, $e^{(\cdot)}$, $\log(\cdot)$, etc.) and operations (e.g., addition, multiplication, composition, etc.). One very useful way of organizing the elementary decomposition of a generic function is via a so-called *computation graph*. The computation graph of a function g not only allows us to more easily understand its anatomy, as a combination of elementary functions and operations, but it also allows us to evaluate a function in a programmatic way. Here we describe the computation graph by studying two simple examples employing a single-input and a multi-input function.

Example B.3 The computation graph of a single-input function
Take the single-input function

$$g(w) = \tanh(w)\cos(w) + \log(w). \tag{B.23}$$

We can represent this function by decomposing it into its simplest parts, as shown in the top panel of Figure B.3. This graphical depiction is like a blueprint, showing us precisely how g is constructed from elementary functions *and* operations. We read this computation graph from left to right starting with the input node representing w, and ending with the full computation of $g(w)$ on the right. Each yellow node in the graph (aside from the input node that is colored differently in gray) represents a single elementary function or operation, and is marked as such. The directed arrows or *edges* connecting pairs of nodes then show how computation flows when evaluating $g(w)$.

The terms *parent* and *child* are often used to describe the local topology of computation graphs for any pair of nodes in the graph connected via a directed

edge. The parent node is where the edge/arrow originates from and the child node is where it points to. Because these parent–child relationships are defined locally, a particular node can be both a parent and a child with respect to other nodes in the graph. For instance, in the computation graph shown in the top panel of Figure B.3, the input node w (colored gray) is parent to the nodes a, b, and c, while a and b themselves are parents to the node d.

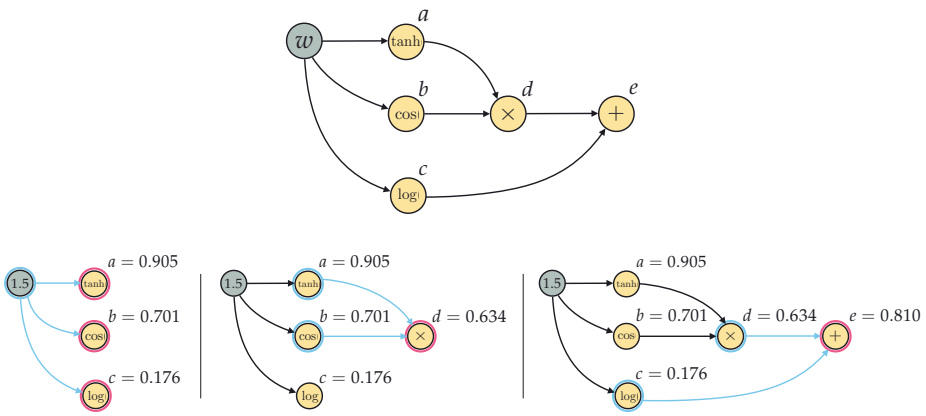

Figure B.3 Figure associated with Example B.3. (top panel) The computation graph for the single-input function defined in Equation (B.23). (bottom panels) Visualizing the flow of computation through this graph. See text for further details.

If we were to write out the formula of each child node in terms of its parent(s) we would have the following list of formulae

$$\begin{aligned} a &= \tanh(w) \\ b &= \cos(w) \\ c &= \log(w) \\ d &= a \times b \\ e &= c + d \end{aligned} \tag{B.24}$$

where the final node evaluates the whole function, i.e., $e = g(w)$. Note, once again, how each child node is a function of its parent(s). Therefore we can, for example, write a as $a(w)$ since w is the (only) parent node of a, or likewise write d as $d(a, b)$ since both a and b are parents to d. At the same time, if we unravel the definition of each node, every node in the end is really a function of the input w alone. In other words, we can also write each node as a function of their common *ancestor* w, e.g., $d(w) = \tanh(w) \times \cos(w)$.

The computation flows *forward* through the graph (from left to right) in sets of parent–child nodes. In the bottom panels of Figure B.3 we illustrate how $g(w)$ is evaluated for the particular input value $w = 1.5$ using the computation graph shown in the top panel of the figure. Beginning on the left we first substitute

the value $w = 1.5$ in the input node, and evaluate each of the input node's children, here computing $a(1.5) = \tanh(1.5) = 0.905$, $b(1.5) = \cos(1.5) = 0.701$, and $c(1.5) = \log(1.5) = 0.176$, as illustrated in the bottom left panel with the parent node highlighted in blue and the children in red. Computation next flows to any child whose parents have all been evaluated, here the node d, as illustrated in the middle panel of the figure where we have used the same coloring to denote the parent–child relationship. Note how in computing $d(1.5) = a(1.5) \times b(1.5) = 0.634$ we only need access to its evaluated parents, i.e., $a(1.5)$ and $b(1.5)$, which we have indeed already computed. We then evaluate the final child node in the graph, e, at our desired input value. Once again, to compute $e(1.5) = c(1.5) + d(1.5) = 0.810$ we only need access to the evaluations made at its parents, here $c(1.5)$ and $d(1.5)$, which have already been computed.

Example B.4 The computation graph of a multi-input function

Computation graphs can similarly be constructed for multi-input functions as well. For example, in the top row of Figure B.4 we show the computation graph for the simple multi-input quadratic

$$g(w_1, w_2) = w_1^2 + w_2^2. \tag{B.25}$$

The two inputs w_1 and w_2 here are each represented by a distinct node. Just as with the single-input case in Example B.3, the computation flows from left to right or *forward* through the graph. Also, as with single-input case, one forward sweep through the graph is sufficient to calculate any value $g(w_1, w_2)$.

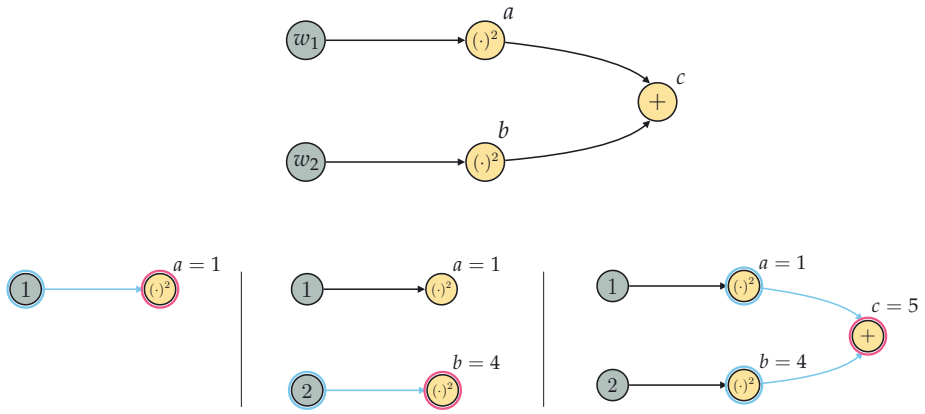

Figure B.4 Figure associated with Example B.4. (top panel) The computation graph for the multi-input quadratic function defined in Equation (B.25). (bottom panels) Visualizing the flow of computation through this graph. See text for further details.

In the bottom panels of Figure B.4 we illustrate how $g(w_1, w_2)$ is evaluated for the particular input values of $w_1 = 1$ and $w_2 = 2$. Beginning on the left we first

substitute in the value $w_1 = 1$ and evaluate its only child $a(1) = 1^2$, as illustrated in the bottom-left panel of the figure. Next, we do the same for the second input node, substituting in the value $w_2 = 2$ and evaluating its only child $b(2) = 2^2$, as illustrated in the bottom-middle panel of the figure. Finally, we move to the last child node, computing it as $c = a(1) + b(2) = 5$ where the evaluations $a(1)$ and $b(2)$ have already been computed in the previous steps of the process.

The notion of a computation graph is quite flexible, as functions can be decomposed in various ways. Here we have broken two example functions down into their simplest, most elementary parts. However, it is more useful to decompose more sophisticated functions (e.g., fully connected neural networks, as detailed in Section 13.2) into computation graphs consisting of more sophisticated elementary building blocks such as matrix multiplication, vector-wise functions, etc.

B.6 The Forward Mode of Automatic Differentiation

In the previous section we saw how representing a function via its computation graph allows us to evaluate it at any input point by traversing the graph in a forward direction, from left to right, recursively evaluating each node in the graph with respect to the function's original input. The computation graph of a function can similarly be used to form and evaluate a function's *gradient* by similarly sweeping forward through the function's computation graph from left to right, forming and evaluating the gradient of each node with respect to the function's original input. In doing this we also naturally evaluate the original function at each node along with the gradient evaluation. This recursive algorithm, called the *forward mode of automatic differentiation*, is easily programmable and allows for the transfer of the tedious chore of gradient computation to a computer program, which makes gradient computation faster and more reliable (than when performed manually and then hard-coded into a computer program). Moreover, unlike numerical differentiation (see Section B.2.2), automatic differentiation provides the *exact* derivative or gradient evaluation, not just an approximation. Here we describe the forward mode of automatic differentiation by studying two examples, employing the algorithm to differentiate a single-input and a multi-input function.

Example B.5 Forward-mode differentiation of a single-input function
Take the function $g(w) = \tanh(w)\cos(w) + \log(w)$ whose computation graph we previously illustrated in Figure B.3. To evaluate the derivative $\frac{\mathrm{d}}{\mathrm{d}w}g(w)$ using the forward mode of automatic differentiation, we traverse this computation graph

much in the same way we did to compute its evaluation $g(w)$ in Example B.3. We begin at the input node all the way to the left where w and its derivative $\frac{\mathrm{d}}{\mathrm{d}w}w$ both have known values: the value of w is chosen by the user and $\frac{\mathrm{d}}{\mathrm{d}w}w$ is always, trivially, equal to 1. With the form of both the node and its derivative in hand we then move to the children of the input node, i.e., nodes a, b, and c. For each child node, we form both the node and its derivative with respect to the input, i.e., a and $\frac{\mathrm{d}}{\mathrm{d}w}a$ for the first child node, b and $\frac{\mathrm{d}}{\mathrm{d}w}b$ for the second child node, and c and $\frac{\mathrm{d}}{\mathrm{d}w}c$ for the third. These first steps are illustrated on the computation graph of the function in the top panel of Figure B.5, where the parent node (the input) is highlighted in blue and the children (nodes a, b, and c) are colored in red.

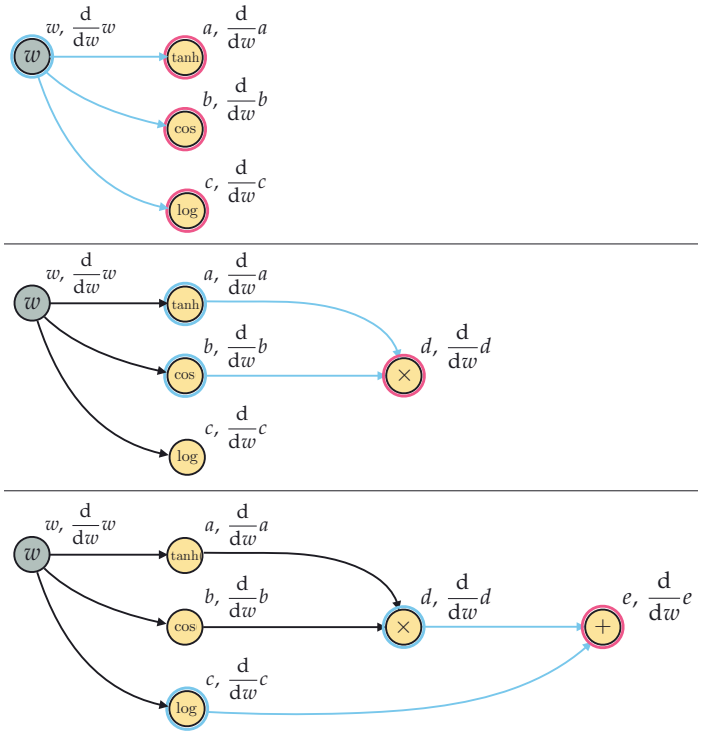

Figure B.5 Figure associated with Example B.5 illustrating forward-mode derivative computation of the single-input function defined in Equation (B.23). At each step of the process a child node and its derivative are both formed with respect to w, which are recursively constructed using the node/derivative evaluations of its parent(s). See text for further details.

Forming each child node and its derivative requires only the values computed at its parent(s), along with the derivative rules for elementary functions and operations described in Section B.3, which tell us how to combine the derivatives computed at the parent(s) to compute the child derivative. For each child node here we can form their derivatives with respect to the input w as

$$\begin{aligned}\frac{\mathrm{d}}{\mathrm{d}w}a(w) &= 1 - \tanh^2(w) \\ \frac{\mathrm{d}}{\mathrm{d}w}b(w) &= -\sin(w) \\ \frac{\mathrm{d}}{\mathrm{d}w}c(w) &= \frac{1}{w}\end{aligned} \tag{B.26}$$

using Table B.1 as a look-up table.

With the current function values/derivatives computed with respect to w we move forward to the next child nodes in the graph where we will see the same pattern emerge, seeking to form the nodes and their derivatives with respect to w. Examining the computation graph in the top panel of Figure B.5 we can see that we have already formed all parents of d (i.e., nodes a and b) as well as their derivatives, and thus we move to node d next and form $d(w) = a(w) \times b(w)$ using the values of $a(w)$ and $b(w)$ just computed. To compute the derivative $\frac{\mathrm{d}}{\mathrm{d}w}d(a,b)$ we employ the *chain rule* and write it, in terms of the derivatives of its parents/inputs, as

$$\frac{\mathrm{d}}{\mathrm{d}w}d(a,b) = \frac{\partial}{\partial a}d(a,b) \times \frac{\mathrm{d}}{\mathrm{d}w}a(w) + \frac{\partial}{\partial b}d(a,b) \times \frac{\mathrm{d}}{\mathrm{d}w}b(w). \tag{B.27}$$

Notice, because we have already formed the derivatives of a and b with respect to w, we need only compute the parent–child derivatives $\frac{\partial}{\partial a}d(a,b)$ and $\frac{\partial}{\partial b}d(a,b)$. Since the parent–child relationship here is multiplicative both of these derivatives can be found, using the *product rule* from Table B.2, as

$$\begin{aligned}\frac{\partial}{\partial a}d(a,b) &= b \\ \frac{\partial}{\partial b}d(a,b) &= a.\end{aligned} \tag{B.28}$$

All together we have the entire form of the derivative at node d as

$$\frac{\mathrm{d}}{\mathrm{d}w}d(a,b) = (1 - \tanh^2(w))\cos(w) - \tanh(w)\sin(w), \tag{B.29}$$

which we illustrate pictorially in the middle panel of Figure B.5.

Now that we have resolved d and its derivative, we can work on the final node e, which is a child of nodes d and c, defined in terms of them as $e(d,c) = d + c$. Once again, using the chain rule the derivative of e with respect to w is written as

$$\frac{\mathrm{d}}{\mathrm{d}w}e(d,c) = \frac{\partial}{\partial d}e(d,c) \times \frac{\mathrm{d}}{\mathrm{d}w}d(w) + \frac{\partial}{\partial c}e(d,c) \times \frac{\mathrm{d}}{\mathrm{d}w}c(w). \tag{B.30}$$

We have already computed the derivatives of d and c with respect to w, so plugging 1 for both $\frac{\partial}{\partial d}e(d,c)$ and $\frac{\partial}{\partial c}e(d,c)$ we have our desired derivative

$$\frac{\mathrm{d}}{\mathrm{d}w}g(w) = \frac{\mathrm{d}}{\mathrm{d}w}e(d,c) = (1-\tanh^2(w))\cos(w) - \tanh(w)\sin(w) + \frac{1}{w}. \tag{B.31}$$

With the form of the derivative computed at each node of the graph we can imagine appending each derivative to its respective node, leaving us with an upgraded computation graph for our function $g(w)$ that can now evaluate both function and derivative values. This is done by plugging in a value for w at the start of the graph, and propagating the function and derivative forward through the graph from left to right. This is why the method is called the *forward mode* of automatic differentiation, as all computation is done moving forward through the computation graph. For example, in Figure B.6 we illustrate how $g(1.5)$ and $\frac{\mathrm{d}}{\mathrm{d}w}g(1.5)$ are computed together traversing forward through the computation graph. Note that in evaluating the derivative in this manner we naturally evaluate the function as well.

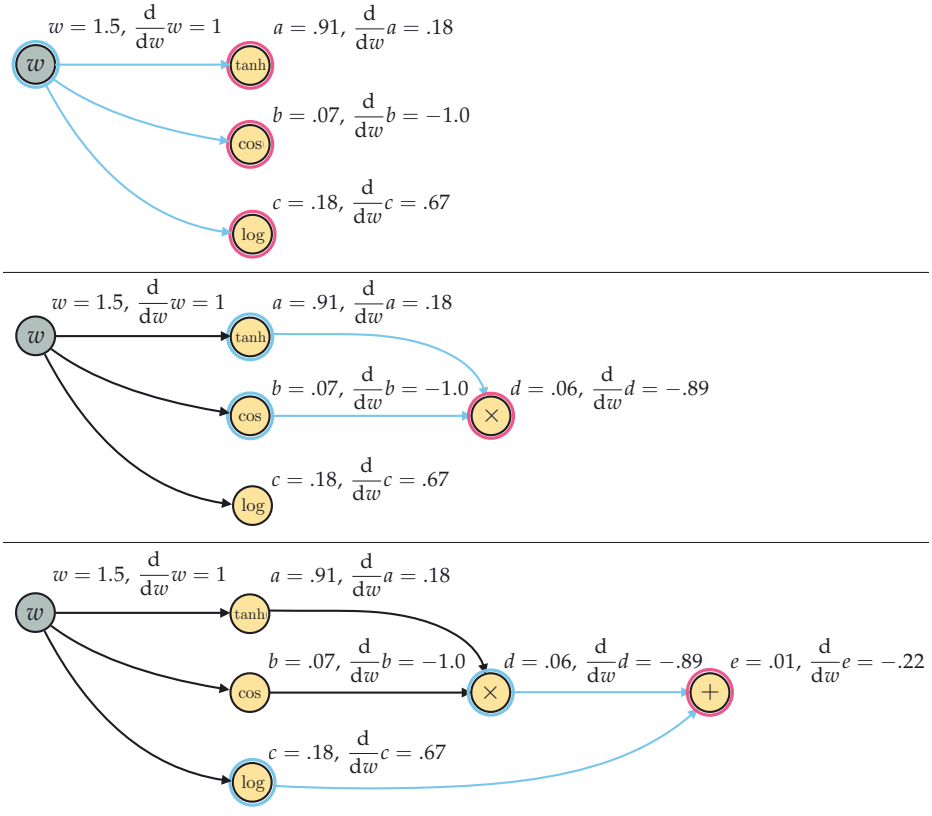

Figure B.6 Figure associated with Example B.5 illustrating evaluation of the function $g(w)$ given in Equation (B.23), as well as its derivative, at the input point $w = 1.5$ using the forward mode of automatic differentiation. See text for further details.

Example B.6 Forward-mode differentiation of a multi-input function
The forward mode of automatic differentiation works similarly for multi-input functions as well, only now we must compute the form of the gradient at each node in the graph (instead of a single derivative). Here we illustrate how this is done using the multi-input quadratic $g(w_1, w_2) = w_1^2 + w_2^2$ whose computation graph was first shown in Figure B.4.

Following the pattern set forth with the single-input function in Example B.5, we begin by computing the gradient at each input node, which are trivially

$$\nabla w_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \qquad \text{and} \qquad \nabla w_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}. \tag{B.32}$$

We then move to the children of our input nodes beginning at node a, where we compute the *gradient* of a or, in other words, the partial derivatives of $a(w_1) = w_1^2$ with respect to both w_1 and w_2, as

$$\frac{\partial}{\partial w_1} a = 2w_1 \qquad \text{and} \qquad \frac{\partial}{\partial w_2} a = 0. \tag{B.33}$$

Similarly, we can compute the partial derivatives of $b(w_2) = w_2^2$ with respect to w_1 and w_2, as

$$\frac{\partial}{\partial w_1} b = 0 \qquad \text{and} \qquad \frac{\partial}{\partial w_2} b = 2w_2. \tag{B.34}$$

These two steps are illustrated in the left panel of Figure B.7. With the form of the gradient computed at nodes a and b we can finally compute the gradient at their common child node, c. Employing the chain rule we have

$$\begin{aligned} \frac{\partial}{\partial w_1} c &= \frac{\partial}{\partial a} c \, \frac{\partial}{\partial w_1} a + \frac{\partial}{\partial b} c \, \frac{\partial}{\partial w_1} b = 1 \times 2w_1 + 1 \times 0 = 2w_1 \\ \frac{\partial}{\partial w_2} c &= \frac{\partial}{\partial a} c \, \frac{\partial}{\partial w_2} a + \frac{\partial}{\partial b} c \, \frac{\partial}{\partial w_2} b = 1 \times 0 + 1 \times 2w_2 = 2w_2. \end{aligned} \tag{B.35}$$

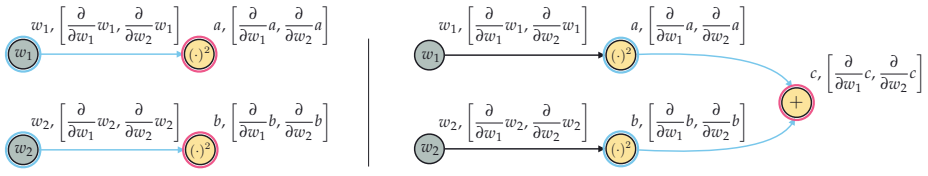

Figure B.7 Figure associated with Example B.6 illustrating forward-mode gradient computation of the multi-input function $g(w_1, w_2)$ defined in Equation (B.25). See text for further details.

The forward-mode differentiator discussed here does not provide an algebraic description of a function's derivative, but a programmatic function (a computation graph) that can be used to evaluate the function and its derivative at any set of input points. Conversely, one can build an algorithm that employs the basic derivative rules to provide an algebraic derivative, but this requires the implementation of a computer algebra system. Such a derivative calculator that deals with derivatives using symbolic computation (i.e., algebra on the computer) is called a *symbolic differentiator*. However, expressing equations algebraically can be quite unwieldy. For example, the rather complicated-looking function

$$g(w) = \frac{w^2 \sin(w^2 + w) \cos(w^2 + 1)}{\log(w + 1)} \tag{B.36}$$

has an expansive algebraic derivative

$$\begin{aligned} \frac{\mathrm{d}}{\mathrm{d}w} g(w) = & \frac{(2w + 1)\, w^2 \cos(w^2 + 1) \cos(w^2 + w)}{\log(w + 1)} \\ & - \frac{w^2 \sin(w^2 + w) \cos(w^2 + 1)}{(w + 1) \log^2(w + 1)} + \frac{2w \sin(w^2 + w) \cos(w^2 + 1)}{\log(w + 1)} \\ & - \frac{2w^3 \sin(w^2 + 1) \sin(w^2 + w)}{\log(w + 1)}. \end{aligned} \tag{B.37}$$

This problem is exponentially worse, to the point of being a considerable computational burden, when dealing with multi-input functions (which we commonly deal with in machine learning). The (forward-mode) automatic differentiator, which produces a computation graph of the derivative instead of an algebraic form, does not have this problem.

Finally, note that because calculations involved in the forward mode of automatic differentiation are made using the computation graph of a given function g, one engineering choice to be made is to decide how the graph will be constructed and manipulated. Essentially we have two choices: we can either construct the computation graph *implicitly* by carefully implementing the elementary derivative rules, or we can parse the input function g and construct its computation graph *explicitly* as we did in explaining the method in this section. The advantage of implicitly constructing the graph is that the corresponding calculator is light-weight (as there is no need to store a graph) and easy to construct. On the other hand, implementing a calculator that explicitly constructs computation graphs requires additional tools (e.g., a parser), but allows for easier computation of higher-order derivatives. This is because in the latter case the differentiator takes in a function to differentiate and treats it as a computation graph, and often outputs a computation graph of its derivative which can then be plugged back into the same algorithm to create second-order derivatives, and so forth.

B.7 The Reverse Mode of Automatic Differentiation

While the forward mode of automatic differentiation introduced in the previous section provides a wonderful programmatic way of computing derivatives, it can be inefficient for many kinds multi-input functions (particularly those in machine learning involving *fully connected networks*). This is because while most of the nodes in the computation graph of a multi-input function may only take in just a few inputs, we compute the *complete gradient* with respect to *all the inputs* at each and every node. This leads to considerable computation waste since we know that the partial derivative of any node with respect to an original input of the function that it does *not* take in will always be equal to zero.

This obvious waste is the motivation for what is called the *reverse mode of automatic differentiation* or, in the jargon of machine learning, the *backpropagation algorithm*. With the reverse mode we begin by traversing a function's computation graph in the forward direction (starting with the input nodes and moving from left to right) computing *only* the form of the partial derivatives needed at each node of the computation graph (ignoring partial derivatives that will always be zero). Once this forward sweep is complete we then (starting with the final node(s) in the graph) move in the *reverse* direction and sweep *backwards* through the graph, collecting and combining the previously computed partial derivatives to appropriately construct the gradient. While this means that we must construct the computation graph explicitly (and store it), the trade-off with wasted computation is often well worth it when dealing with machine learning functions. This makes the *reverse mode* a more popular choice (than the forward mode detailed in the prior section) in machine learning applications, and is in particular the brand of automatic differentiator implemented in `autograd` (the `Python`-based automatic differentiator we recommend one uses with this text).

Example B.7 Reverse-mode differentiation of a multi-input function

In Example B.6 of the previous section we described how to compute the gradient of the function $g(w_1, w_2) = w_1^2 + w_2^2$ using forward-mode automatic differentiation. Notice how, in calculating the full gradient at each node of this function's computation graph, we performed several wasteful computations: whenever partial derivatives were taken with respect to input not taken in by a node (through its complex web of parent–child relations leading back to the original input of the function) we know by default this partial derivative will always equal zero. For example, the partial derivative $\frac{\partial}{\partial w_2} a = 0$ since a is not a function of the original input w_2.

In Figure B.8 we redraw the computation graph of this quadratic with the gradient expressed in terms of the partial derivatives at each node and with all zero partials marked accordingly. Examining this graph we can see that a good deal of the partials trivially equal zero. Trivial zeros such as these waste computation as we form them traversing *forward* through the graph.

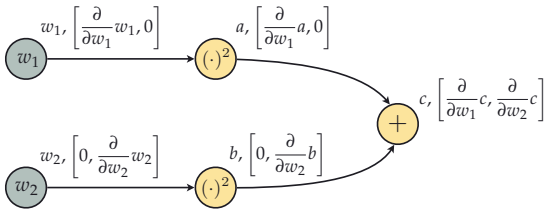

Figure B.8 Figure associated with Example B.7. The computation graph of the quadratic function described in the text, with zero partials marked. See text for further details.

This issue becomes much more severe with multi-input functions that take in larger numbers of input variables. For example, in Figure B.9 we illustrate the computational graph of the analogous quadratic function taking four inputs $g(w_1, w_2, w_3, w_4) = w_1^2 + w_2^2 + w_3^2 + w_4^2$. In this case more than *half* of all the gradient entries at the nodes in this graph are zero due to the fact that certain nodes are not functions of certain inputs, and hence their partial derivatives are always zero.

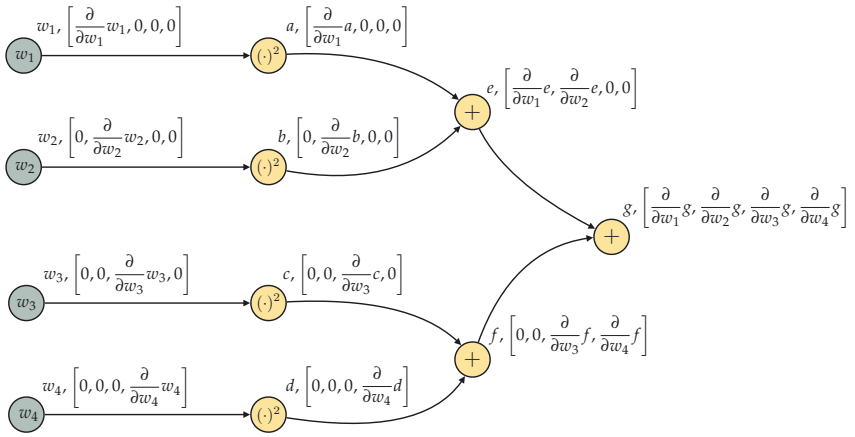

Figure B.9 Figure associated with Example B.7. The computation graph of a simple four-input quadratic with the gradient expressed at each node in terms of partial derivatives. Here over half of the partial derivatives computed are trivial zeros. See text for further details.

To remedy this inefficiency, automatic differentiation can also be performed in a *reverse mode*, which consists of a forward and reverse (or backward) sweep through the computation graph of a function. In the forward sweep of the reverse mode we traverse the computation graph in forward direction (from left to right) recursively just as with the forward mode, only at each node we compute the partial derivatives of each child node with respect to its parent(s) *only*, and *not* the full gradient with respect to the function input.

This is illustrated for the quadratic function $g(w_1, w_2) = w_1^2 + w_2^2$ in the top panels of Figure B.10. In the top-left panel we show the computing of partial derivatives for child nodes a and b (colored red), which are taken only with respect to their parents (which here are w_1 and w_2, respectively, colored blue). In the top-right panel we illustrate the next computation in the forward sweep, the partial derivatives computed at the child node c (colored red) with respect to its parents a and b (colored blue).

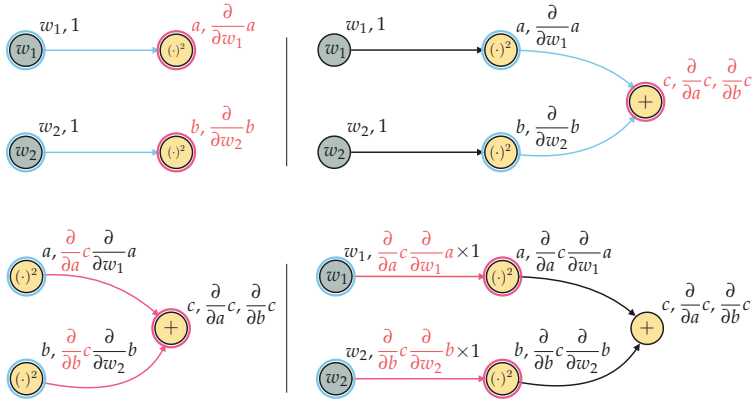

Figure B.10 Figure associated with Example B.7. Forward (top panels) and backward sweep (bottom panels) illustrated for a simple two-input quadratic function. See text for further details.

Once the forward sweep is complete, we change course and traverse the computation graph in the *reverse* direction, starting at the end node and traversing backwards recursively from right to left until we reach the input nodes. At every step of the process we update the partial derivative of each parent by multiplying it by the partial derivative of its child node with respect to that parent (when the parent node has multiple children the accumulated partials should be added, not multiplied, since this is what the chain rule requires). When the backward sweep is completed we will have recursively constructed the gradient of the function with respect to all of its inputs.

The backward sweep is illustrated for the two-input quadratic function in the bottom panels of Figure B.10. Starting from end node in our forward sweep, i.e., node c, we observe that c has two parents: a and b. Therefore we update the derivative at a by (left) multiplying it by $\frac{\partial}{\partial a} c$ giving $\frac{\partial}{\partial a} c \frac{\partial}{\partial w_1} a$, and similarly update the derivative at b by (left) multiplying it by $\frac{\partial}{\partial b} c$ giving $\frac{\partial}{\partial b} c \frac{\partial}{\partial w_2} b$. We then repeat this procedure recursively with the children of a and b, ending with the partial derivative $\frac{\partial}{\partial a} c \frac{\partial}{\partial w_1} a \frac{\partial}{\partial w_1} w_1 = \frac{\partial}{\partial a} c \frac{\partial}{\partial w_1} a$ and $\frac{\partial}{\partial b} c \frac{\partial}{\partial w_2} b \frac{\partial}{\partial w_2} w_2 = \frac{\partial}{\partial b} c \frac{\partial}{\partial w_2} b$. These are precisely the two partial derivatives of the complete gradient of the quadratic with respect to its input w_1 and w_2.

B.8 Higher-Order Derivatives

In previous sections we have seen how we can efficiently compute the derivative of functions composed of elementary building blocks, and that these derivatives are themselves functions built from elementary building blocks. Because of this we can likewise compute derivatives of derivatives, commonly referred to as *higher-order derivatives*, which are the subject of this section.

B.8.1 Higher-order derivatives of single-input functions

Here we explore the concept of higher-order derivatives of single-input functions by looking at a simple example.

Example B.8 Higher-order derivatives
To compute the second-order derivative of the function

$$g(w) = w^4 \tag{B.38}$$

we first find its first-order derivative as

$$\frac{\mathrm{d}}{\mathrm{d}w}g(w) = 4w^3 \tag{B.39}$$

and then differentiate it one more time to get

$$\frac{\mathrm{d}}{\mathrm{d}w}\left(\frac{\mathrm{d}}{\mathrm{d}w}g(w)\right) = 12w^2. \tag{B.40}$$

Taking the derivative of the resulting function one more time gives the third-order derivative of g as

$$\frac{\mathrm{d}}{\mathrm{d}w}\left(\frac{\mathrm{d}}{\mathrm{d}w}\left(\frac{\mathrm{d}}{\mathrm{d}w}g(w)\right)\right) = 24w. \tag{B.41}$$

Similarly, we can compute the first three derivatives of the function

$$g(w) = \cos(3w) + w^2 + w^3 \tag{B.42}$$

explicitly as

$$\begin{aligned}\frac{\mathrm{d}}{\mathrm{d}w}g(w) &= -3\sin(3w) + 2w + 3w^2 \\ \frac{\mathrm{d}}{\mathrm{d}w}\left(\frac{\mathrm{d}}{\mathrm{d}w}g(w)\right) &= -9\cos(3w) + 2 + 6w \\ \frac{\mathrm{d}}{\mathrm{d}w}\left(\frac{\mathrm{d}}{\mathrm{d}w}\left(\frac{\mathrm{d}}{\mathrm{d}w}g(w)\right)\right) &= 27\sin(3w) + 6.\end{aligned} \tag{B.43}$$

Higher-order derivatives are also often expressed using more compact notation than given in Example B.8. For instance, the second derivative is very often denoted more compactly using the following notation

$$\frac{\mathrm{d}^2}{\mathrm{d}w^2}g(w) = \frac{\mathrm{d}}{\mathrm{d}w}\left(\frac{\mathrm{d}}{\mathrm{d}w}g(w)\right). \tag{B.44}$$

Likewise, the third derivative is often denoted more compactly as

$$\frac{\mathrm{d}^3}{\mathrm{d}w^3}g(w) = \frac{\mathrm{d}}{\mathrm{d}w}\left(\frac{\mathrm{d}}{\mathrm{d}w}\left(\frac{\mathrm{d}}{\mathrm{d}w}g(w)\right)\right) \tag{B.45}$$

and, in general, the nth derivative is written as

$$\frac{\mathrm{d}^n}{\mathrm{d}w^n}g(w). \tag{B.46}$$

B.8.2 Higher-order derivatives of multi-input functions

We have seen how the gradient of a multi-input function is a collection of partial derivatives

$$\nabla g(w_1, w_2, \ldots, w_N) = \begin{bmatrix} \frac{\partial}{\partial w_1}g(w_1, w_2, \ldots, w_N) \\ \frac{\partial}{\partial w_2}g(w_1, w_2, \ldots, w_N) \\ \vdots \\ \frac{\partial}{\partial w_N}g(w_1, w_2, \ldots, w_N) \end{bmatrix} \tag{B.47}$$

where the gradient contains the nth partial derivative $\frac{\partial}{\partial w_n}g(w_1, w_2, \ldots, w_N)$ as its nth entry. This partial derivative (like the original function itself) is a function, taking in the N inputs abbreviated $\mathbf{w}$, which we can differentiate along each input axis. For instance, we can take the mth partial derivative of $\frac{\partial}{\partial w_n}g(w_1, w_2, .., w_N)$ as

$$\frac{\partial}{\partial w_m}\frac{\partial}{\partial w_n}g(w_1, w_2, \ldots, w_N). \tag{B.48}$$

This is a *second-order* derivative. How many of these does the function g have? Since every one of g's N first-order derivatives (each being a function of N inputs) has N partial derivatives, $g(\mathbf{w})$ has a total of N^2 second-order derivatives.

As with the notion of the gradient, this large set of second-order derivatives are typically organized in a particular way so that they can be more easily communicated and computed with. The *Hessian* – which is written notationally as $\nabla^2 g(\mathbf{w})$ – is the $N \times N$ matrix of second-order derivatives whose (m, n)th element is $\frac{\partial}{\partial w_m}\frac{\partial}{\partial w_n}g(\mathbf{w})$, or $\frac{\partial}{\partial w_m}\frac{\partial}{\partial w_n}g$ for short. The full Hessian matrix is written as

$$\nabla^2 g(\mathbf{w}) = \begin{bmatrix} \frac{\partial}{\partial w_1}\frac{\partial}{\partial w_1}g & \frac{\partial}{\partial w_1}\frac{\partial}{\partial w_2}g & \cdots & \frac{\partial}{\partial w_1}\frac{\partial}{\partial w_N}g \\ \frac{\partial}{\partial w_2}\frac{\partial}{\partial w_1}g & \frac{\partial}{\partial w_2}\frac{\partial}{\partial w_2}g & \cdots & \frac{\partial}{\partial w_2}\frac{\partial}{\partial w_N}g \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial}{\partial w_N}\frac{\partial}{\partial w_1}g & \frac{\partial}{\partial w_N}\frac{\partial}{\partial w_2}g & \cdots & \frac{\partial}{\partial w_N}\frac{\partial}{\partial w_N}g \end{bmatrix}. \tag{B.49}$$

Moreover, since it is virtually always the case that $\frac{\partial}{\partial w_m}\frac{\partial}{\partial w_n}g = \frac{\partial}{\partial w_n}\frac{\partial}{\partial w_m}g$, particularly with the sort of functions used in machine learning, the Hessian is always a *symmetric* matrix.

The number of partial derivatives of a multi-input function grows *exponentially* with the order. We have just seen that a function taking in N inputs has N^2 second-order derivatives. In general such a function has N^D partial derivatives of order D.

B.9 Taylor Series

In this section we describe the *Taylor series* of a function, a fundamental tool of calculus that is critically important for first- and second-order local optimization (as detailed in Chapters 3 and 4 of this text). We begin by deriving this crucial concept for single-input functions, and follow by generalizing to the case of multi-input functions.

B.9.1 Linear approximation is only the beginning

We began our discussion of derivatives in Section B.2 by defining the derivative at a point as the slope of the tangent line to a given input function. For a function $g(w)$ we then formally described the tangent line at a point w^0 as

$$h(w) = g(w^0) + \frac{\mathrm{d}}{\mathrm{d}w}g(w^0)(w - w^0) \tag{B.50}$$

with the slope here given by the derivative $\frac{\mathrm{d}}{\mathrm{d}w}g(w^0)$. The justification for examining the tangent line to begin with is fairly straightforward: locally (close to the point w^0) the tangent line looks awfully similar to the function, and so if we want to better understand g near w^0 we can just as well look at the tangent line there. This makes our lives a lot easier because a line (compared to a generic g) is always a comparatively simple object, and therefore understanding the tangent line is always a simple affair.

If we study the form of our tangent line $h(w)$ closely, we can define in precise mathematical terms how it matches the function g. Notice, first of all, that the tangent line takes on the same value as the function g at the point w^0, i.e.,

$$h(w^0) = g(w^0) + \frac{\mathrm{d}}{\mathrm{d}w}g(w^0)(w^0 - w^0) = g(w^0). \tag{B.51}$$

Next, notice that the first derivative value of these two functions match as well. That is, if we take the first derivative of h with respect to w we can see that

$$\frac{\mathrm{d}}{\mathrm{d}w}h(w^0) = \frac{\mathrm{d}}{\mathrm{d}w}g(w^0). \tag{B.52}$$

In short, when the tangent line h matches g so that at w^0 both the function *and* derivative values are equal, we can write

$$\begin{aligned} h(w^0) &= g(w^0) \\ \frac{\mathrm{d}}{\mathrm{d}w}h(w^0) &= \frac{\mathrm{d}}{\mathrm{d}w}g(w^0). \end{aligned} \tag{B.53}$$

What if we turned this idea around and tried to find a line that satisfies these two properties. In other words, we start with a general line

$$h(w) = a_0 + a_1(w - w^0) \tag{B.54}$$

with unknown coefficients a_0 and a_1, and we want to determine the right value for these coefficients so that the line satisfies the two criteria in Equation (B.53). These two criteria constitute a *system of equations* we can solve for the correct values of a_0 and a_1. Computing the left-hand side of each, where h is our general line in Equation (B.54), we end up with a trivial system of equations to solve for both unknowns simultaneously

$$\begin{aligned} h(w^0) &= a_0 = g(w^0) \\ \frac{\mathrm{d}}{\mathrm{d}w}h(w^0) &= a_1 = \frac{\mathrm{d}}{\mathrm{d}w}g(w^0) \end{aligned} \tag{B.55}$$

and behold, the coefficients are precisely those of the tangent line.

B.9.2 From tangent line to tangent quadratic

Given that the function and derivative values of the tangent line match those of its underlying function, can we do better? Can we find a simple function that matches the function value, first derivative, *and* the second derivative value at a point w_0? In other words, is it possible to determine a simple function h that satisfies the following *three* conditions?

$$\begin{aligned} h(w^0) &= g(w^0) \\ \frac{\mathrm{d}}{\mathrm{d}w}h(w^0) &= \frac{\mathrm{d}}{\mathrm{d}w}g(w^0) \\ \frac{\mathrm{d}^2}{\mathrm{d}w^2}h(w^0) &= \frac{\mathrm{d}^2}{\mathrm{d}w^2}g(w^0) \end{aligned} \tag{B.56}$$

Notice how a (tangent) line h can only satisfy the first two of these properties

and never the third, since it is a degree-one polynomial and $\frac{d^2}{dw^2}h(w) = 0$ for all w. This fact implies that we need *at least* a degree-two polynomial to satisfy all three criteria. Starting with a general degree-two polynomial

$$h(w) = a_0 + a_1(w - w^0) + a_2(w - w^0)^2 \tag{B.57}$$

with unknown coefficients a_0, a_1, and a_2, we can evaluate the left-hand side of each criterion in Equation (B.56), forming a system of three equations, and solve for these coefficients.

$$\begin{aligned} h(w^0) &= a_0 = g(w^0) \\ \frac{d}{dw}h(w^0) &= a_1 = \frac{d}{dw}g(w^0) \\ \frac{d^2}{dw^2}h(w^0) &= 2a_2 = \frac{d^2}{dw^2}g(w^0) \end{aligned} \tag{B.58}$$

With all of our coefficients determined (in terms of the derivatives of g at w^0) we have a degree-two polynomial that satisfies the three desired criteria

$$h(w) = g(w^0) + \frac{d}{dw}g(w^0)(w - w^0) + \frac{1}{2}\frac{d^2}{dw^2}g(w^0)(w - w^0)^2. \tag{B.59}$$

This is one step beyond the tangent line (an approximating quadratic function) but note that the first two terms are indeed the tangent line itself. Such a quadratic approximation matches a generic function g near the point w_0 far more closely than the tangent line, as illustrated in Figure B.11.

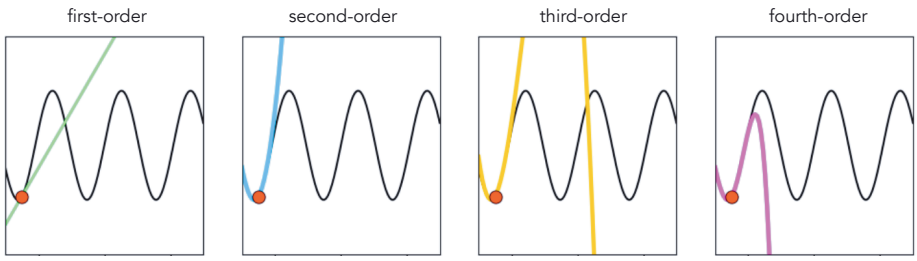

Figure B.11 From left to right, the first-, second-, third-, and fourth-order Taylor series approximation to the function $g(w) = \sin(w)$, drawn in different colors, evaluated at the same input point.

B.9.3 Building better and better local approximations

Having derived this quadratic based on our reflection on the tangent line, one can naturally think of going one step further. That is, finding a simple function h that satisfies even one more condition than the quadratic.

$$\begin{aligned} h(w^0) &= g(w^0) \\ \frac{\mathrm{d}}{\mathrm{d}w}h(w^0) &= \frac{\mathrm{d}}{\mathrm{d}w}g(w^0) \\ \frac{\mathrm{d}^2}{\mathrm{d}w^2}h(w^0) &= \frac{\mathrm{d}^2}{\mathrm{d}w^2}g(w^0) \\ \frac{\mathrm{d}^3}{\mathrm{d}w^3}h(w^0) &= \frac{\mathrm{d}^3}{\mathrm{d}w^3}g(w^0) \end{aligned} \tag{B.60}$$

Noting that no degree-two polynomial could satisfy this last condition, since its third derivative is always equal to zero, we could seek out a degree-three polynomial. Using the same analysis as we performed previously, setting up the corresponding system of equations based on a generic degree-three polynomial leads to the conclusion that the following does indeed satisfy all of the criteria in Equation (B.60)

$$h(w) = g(w^0) + \frac{\mathrm{d}}{\mathrm{d}w}g(w^0)(w - w^0) + \frac{1}{2}\frac{\mathrm{d}^2}{\mathrm{d}w^2}g(w^0)(w - w^0)^2 + \frac{1}{6}\frac{\mathrm{d}^3}{\mathrm{d}w^3}g(w^0)(w - w^0)^3. \tag{B.61}$$

This is an even better approximation of g near the point w^0 than the quadratic, as illustrated for a particular example in Figure B.11. Examining this figure we can clearly see that the approximation becomes better and better as we increase the order of the approximation. This makes sense, as each polynomial contains more of the underlying function's derivative information as we increase the degree. However, we can never expect it to match the entire function everywhere: we build each polynomial to match g at only a single point, so regardless of degree we can expect it only to match the underlying function near the point w_0.

Setting up a set of $N + 1$ criteria, the first demanding that $h(w^0) = g(w^0)$ and the remaining N demanding that the first N derivatives of h match those of g at w^0, leads to construction of the degree-N polynomial

$$h(w) = g(w^0) + \sum_{n=1}^{N} \frac{1}{n!}\frac{\mathrm{d}^n}{\mathrm{d}w^n}g(w^0)(w - w^0)^n. \tag{B.62}$$

This general degree-N polynomial is called the *Taylor series* of g at the point w^0.

B.9.4 Multi-input Taylor series

We have now seen, with single-input functions, how the general Taylor series approximation can be thought of as a natural extension of the tangent line for higher-degree polynomial approximations. The story with multi-input functions is precisely analogous.

If we asked what sort of degree-one polynomial $h(\mathbf{w})$ matched a function $g(\mathbf{w})$ at a point $\mathbf{w}^0$ in terms of both its function and gradient value there, i.e.,

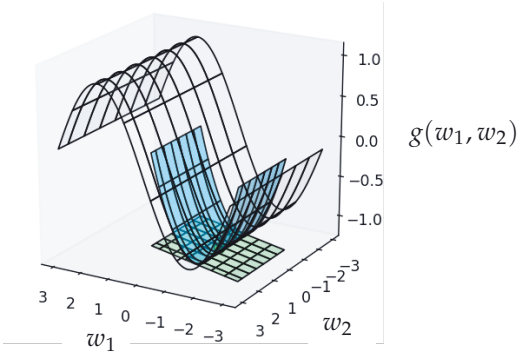

Figure B.12 The first-order (colored in lime green) and second-order (colored in turquoise) Taylor series approximations to the function $g(w_1, w_2) = \sin(w_1)$ at a point near the origin.

$$\begin{aligned} h\left(\mathbf{w}^0\right) &= g\left(\mathbf{w}^0\right) \\ \nabla h\left(\mathbf{w}^0\right) &= \nabla g\left(\mathbf{w}^0\right) \end{aligned} \tag{B.63}$$

we could set up a system of equations (mirroring the one we set up when asking the analogous question for single-input functions) and recover the tangent hyperplane (our first-order Taylor series approximation) we saw back in Section B.4

$$h(\mathbf{w}) = g(\mathbf{w}^0) + \nabla g(\mathbf{w}^0)^T(\mathbf{w} - \mathbf{w}^0). \tag{B.64}$$

Notice how this is the exact analog of the first-order approximation for single-input functions, and reduces to it (a tangent line) when $N = 1$.

Likewise, inquiring about what sort of degree-two (quadratic) function h could match g at a point $\mathbf{w}^0$ in terms of the value it takes, as well as the values its first and second derivatives take, i.e.,

$$\begin{aligned} h\left(\mathbf{w}^0\right) &= g\left(\mathbf{w}^0\right) \\ \nabla h\left(\mathbf{w}^0\right) &= \nabla g\left(\mathbf{w}^0\right) \\ \nabla^2 h\left(\mathbf{w}^0\right) &= \nabla^2 g\left(\mathbf{w}^0\right) \end{aligned} \tag{B.65}$$

we would likewise derive (as we did explicitly with the single-input case) the second-order Taylor series approximation

$$h(\mathbf{w}) = g(\mathbf{w}^0) + \nabla g(\mathbf{w}^0)^T(\mathbf{w} - \mathbf{w}^0) + \frac{1}{2}(\mathbf{w} - \mathbf{w}^0)^T\nabla^2 g\left(\mathbf{w}^0\right)(\mathbf{w} - \mathbf{w}^0). \tag{B.66}$$

Notice once again how this is the exact analog of the second-order approximation for single-input functions, and reduces to it when $N = 1$.

In Figure B.12 we plot the first- and second-order Taylor series approximation (shown in lime green and turquoise, respectively) of the function $g(w_1, w_2) =$

$\sin(w_1)$ at a point near the origin. As was the case with single-input functions the second-order approximation is a much better local approximator than the first, as it contains more derivative information there.

Higher-order Taylor series approximations can be defined precisely as in the single-input case. The main difference with multi-input functions is that higher-order approximations, starting with the third-order derivative, require serious manipulation of tensors of partial derivatives.

While this can be readily defined, only approximations up to the second order are ever used in practice. This is because (as we saw in Section B.8.2) the number of partial derivatives grows *exponentially* in the order of a derivative. Thus even though we get a better (local) approximator as we increase the order of a Taylor series, the serious pitfall of calculating/storing exponentially many partial derivatives nullifies the benefit.

B.10 Using the `autograd` Library

In this section we demonstrate how to use a simple yet powerful *automatic differentiator* written in `Python`, called `autograd` [10, 11] – a tool we make extensive use of throughout the text.

B.10.1 Installing `autograd`

`autograd` is an open-source professional-grade gradient calculator, or *automatic differentiator*, whose default is the *reverse mode* outlined in the previous section. It allows you to automatically compute the derivative of arbitrarily complex functions built using basic `Python` and `NumPy` functions.

It is also very easy to install: simply open a terminal and type

```
pip install autograd
```

to install the program. You can also visit the github repository for `autograd` at

```
https://github.com/HIPS/autograd
```

to download the same set of files to your machine. Another tool called JAX – built by the same community as an extension of `autograd` for use on GPUs and TPUs – can be used in a very similar manner as described here, and can be downloaded by visiting

```
https://github.com/google/jax
```

Along with `autograd` we also highly recommend the Anaconda `Python` 3 distribution, which can be installed by visiting

```
https://anaconda.com
```

This standard Python distribution includes a number of useful libraries, including NumPy, Matplotlib, and Jupyter notebooks.

B.10.2 Using autograd

Here we show a number of examples highlighting the basic usage of the autograd automatic differentiator. With simple modules we can easily compute derivatives of single-input functions, as well as partial derivatives and complete gradients of multi-input functions implemented in Python and NumPy.

Example B.9 Computing derivatives of single-input functions

Since autograd is specially designed to automatically compute the derivative(s) of NumPy code, it comes with its own wrapper on the basic NumPy library. This is where the differentiation rules (applied specifically to NumPy functionality) are defined. You can use autograd's version of NumPy exactly like you would the standard version, as virtually nothing about the user interface has been changed. To import this autograd wrapped version of NumPy, type

```
# import statement for autograd wrapped NumPy
import autograd.numpy as np
```

We begin by demonstrating the use of autograd with the simple function

$$g(w) = \tanh(w) \tag{B.67}$$

whose derivative, written algebraically, is

$$\frac{\mathrm{d}}{\mathrm{d}w} g(w) = 1 - \tanh^2(w). \tag{B.68}$$

There are two common ways of defining functions in Python. First is the standard Python function declaration (as shown below).

```
# a named Python function
def g(w):
    return np.tanh(w)
```

You can also create *anonymous* functions in Python (functions you can define in a single line of code) by using the lambda command.

```
# a function defined via lambda
g = lambda w: np.tanh(w)
```

Regardless of how we define our function in `Python` it still amounts to the same thing mathematically/computationally.

To compute the derivative of our function we must first import the gradient calculator, conveniently called `grad`.

```
# import autograd's basic automatic differentiator
from autograd import grad
```

To use the `grad` function we simply pass in the function we wish to differentiate. `grad` works by *explicitly* computing the computation graph of our input, returning its derivative that we can then evaluate wherever we want. It does not provide an *algebraic function*, but a `Python` *function*. Here we call the derivative function of our input `dgdw`.

```
# create the gradient of g
dgdw = grad(g)
```

We can compute higher-order derivatives of our input function by using the same `autograd` functionality recursively, i.e., by plugging in the derivative function `dgdw` into `autograd`'s `grad` function. Doing this once gives us the second derivative `Python` function, which we call `d2gdw2`.

```
# compute the second derivative of g
d2gdw2 = grad(dgdw)
```

We plot the input function as well as its first and second derivatives in Figure B.13.

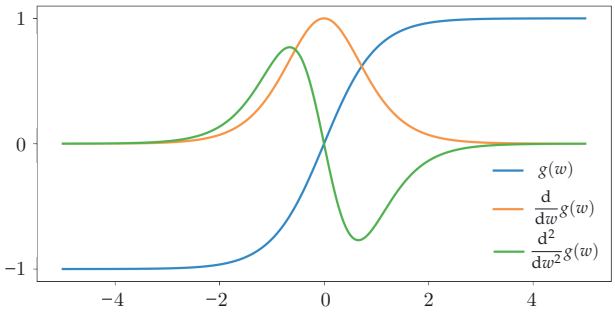

Figure B.13 Figure associated with Example B.9. See text for details.

Example B.10 Function and gradient evaluations

As mentioned in the previous section, when we use an automatic differentiator to evaluate the gradient of a function we evaluate *the function itself* in the process. In other words, whenever we evaluate the gradient, we get the function evaluation "for free."

However, the `grad` function in the previous example returned only a single value: the evaluation of the derivative. The function evaluation is indeed being computed "under the hood" of `grad`, and is simply not returned for ease of use.

There is another `autograd` method called `value_and_grad` that returns everything computed "under the hood" including both the derivative(s) and function evaluation(s). Below we use this `autograd` functionality to reproduce the previous example's first derivative calculations.

```
# import autograd's automatic differentiator
from autograd import value_and_grad

# create the gradient of g
dgdw = value_and_grad(g)

# evaluate g and its gradient at w=0
w = 0.0
g_val, grad_val = dgdw(w)
```

Example B.11 Computing Taylor series approximations

Using `autograd` we can easily compute Taylor series approximations (see Section B.9) of any single-input function. Take, for instance, the function $g(w) = \tanh(w)$ along with its first-order Taylor series approximation

$$h(w) = g(w^0) + \frac{\mathrm{d}}{\mathrm{d}w}g(w^0)(w - w^0) \tag{B.69}$$

centered at the point $w^0 = 1$. First, we produce this function and its first-order Taylor series approximation in `Python` as follows.

```
# create the function g and its first derivative
g = lambda w: np.tanh(w)
dgdw = grad(g)

# create first-order Taylor series approximation
first_order = lambda w0, w: g(w0) + dgdw(w0)*(w - w0)
```

Next, we evaluate and plot the function (in black) and its first-order approximation (in green) in Figure B.14. It is just as easy to compute the second-order Taylor series approximation as well, whose formula is given as

$$q(w) = g(w^0) + \frac{\mathrm{d}}{\mathrm{d}w}g(w^0)(w - w^0) + \frac{1}{2}\frac{\mathrm{d}^2}{\mathrm{d}w^2}g(w^0)(w - w^0)^2. \tag{B.70}$$

```
# create the second derivative of g
d2gdw2 = grad(dgdw)

# create second-order Taylor series approximation
second_order = lambda w0, w: g(w0) + dgdw(w0)*(w - w0) + 0.5*d2gdw2(w0
    )*(w - w0)**2
```

The second-order Taylor series approximation is shown (in blue) in Figure B.14, with the point of expansion/tangency shown as a red circle.

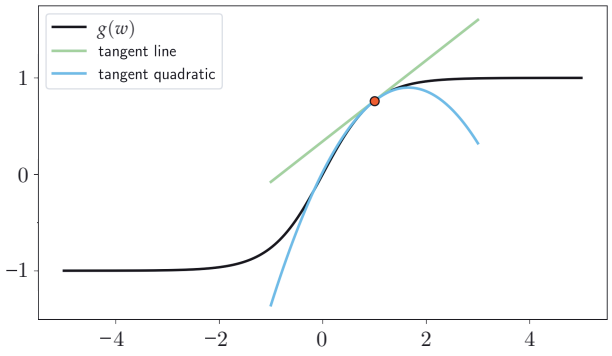

Figure B.14 Figure associated with Example B.11. See text for details.

Example B.12 Computing individual partial derivatives

There are a number of ways we can go about using `autograd` to compute the partial derivatives of a multi-input function. First, let us look at how to use `autograd` to compute partial derivatives individually or, one at a time, beginning with the function

$$g\left(w_1, w_2\right) = \tanh\left(w_1 w_2\right). \tag{B.71}$$

We translate this function into `Python` below.

```
# a simple multi-input function
def g(w_1, w_2):
    return np.tanh(w_1*w_2)
```

Taking in two inputs, this function lives in three dimensions, as plotted in the left panel of Figure B.15.

If we use the exact same call we used in the previous examples, and write

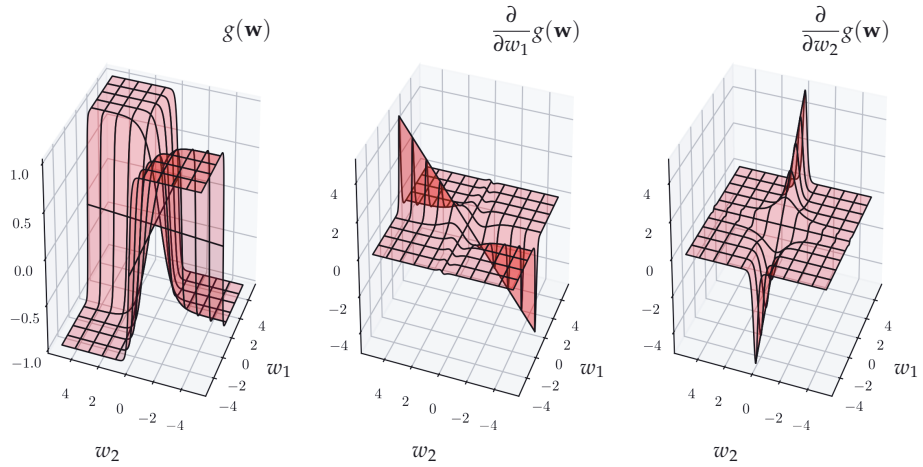

Figure B.15 Figure associated with Example B.12. A multi-input function (left panel) and its partial derivative in the first (middle panel) and second input (right panel). See text for further details.

```
grad(g)
```

since our function takes in two inputs, this will return the *first* partial derivative $\frac{\partial}{\partial w_1} g\,(w_1, w_2)$. This is the *default* setting of each automatic differentiation method in `autograd`.

Alternatively to compute this partial derivative function we can explicitly pass a second argument to `grad`, or any of the other `autograd` methods, which is a simple index denoting which partial derivative we want. To create the same (first) partial derivative this way, we pass in the index `0` (since `Python` indexing starts with `0`) as follows.

```
grad(g, 0)
```

Similarly, to construct the second partial derivative we pass in the index `1`.

```
grad(g, 1)
```

More generally, if g takes in N inputs (w_1 through w_N) we can construct its nth partial derivative using the same pattern as follows.

```
grad(g, n-1)
```

We plot these two partial derivative functions in the middle and right panels of Figure B.15.

Example B.13 Computing several derivatives or the full gradient

Building on the previous example, here we look at how to use `autograd` to construct several partial derivative functions at once or the entire gradient of a multi-input function. We do this via example, using the same function employed in the previous example.

There are two ways to do this using `autograd`. The first way is to simply index all the partial derivatives desired using the same sort of notation introduced previously. For instance, if we wish to construct the full gradient of the two-input function employed in the previous example, we tell `autograd` of this desire by feeding in the two indices `(0,1)` as shown below.

```
grad(g, (0,1))
```

More generally, for a function taking in N inputs, to construct any subset of partial derivatives at once we use the same sort of indexing notation. Note that this usage applies to all methods in the `autograd` automatic differentiation library.

The second way to construct several derivatives at once using `autograd` is by writing a function in `NumPy` where the desired variables we wish to differentiate with respect to are all input into the function as a single argument. For example, instead of writing out our function in `Python` as

```
# a simple multi-input function defined in Python
def g(w_1, w_2):
    return np.tanh(w_1*w_2)
```

where both `w_1` and `w_2` are fed in one at a time, if we write it equivalently using vector notation as

```
def g(w):
    return np.tanh(w[0]*w[1])
```

then the call

```
grad(g)
```

or equivalently

```
grad(g, 0)
```

will produce derivatives of g with respect to its first argument, which here will give us the complete gradient of g.

This indexing format holds more generally as well, that is, the statement

```
grad(g, n-1)
```

computes the derivatives of the function with respect to the nth input (whether it is a single variable or multiple variables).

B.10.3 Flattening mathematical functions using autograd

Mathematical functions come in all shapes and sizes, and moreover, we can often express individual functions algebraically in a variety of different ways. This short section discusses *function flattening*, which is a convenient way to *standardize* functions implemented in Python so that, for example, we can more quickly apply (in code) local optimization without the need to loop over weights of a particularly complicated function.

Example B.14 Flattening a multi-input function
Consider the function

$$g(a, \mathbf{b}, \mathbf{C}) = \left(a + \mathbf{r}^T\mathbf{b} + \mathbf{z}^T\mathbf{C}\mathbf{z}\right)^2 \tag{B.72}$$

where the input variable a is a scalar, $\mathbf{b}$ is a 2×1 vector, $\mathbf{C}$ is a 2×2 matrix, and the nonvariable vectors $\mathbf{r}$ and $\mathbf{z}$ are fixed at $\mathbf{r} = [1\ 2]^T$ and $\mathbf{z} = [1\ 3]^T$, respectively. This function is not written in the standard form $g(\mathbf{w})$ in which we discuss local optimization in Chapters 2 through 4 of this text. While, of course, all of the principles and algorithms described there still apply to this function, the implementation of any of those methods will be naturally more complicated as each step will need to be explicitly written in all three inputs a, $\mathbf{b}$, and $\mathbf{C}$, which is more cumbersome to write (and implement). This annoyance is greatly amplified when dealing with functions of many more explicit inputs variables, which we regularly encounter in machine learning. For such functions, in order to take a single descent step using some local method we must *loop* over their many different input variables.

Thankfully, every mathematical function can be expressed so that *all* of its input variables are represented as a single contiguous vector $\mathbf{w}$, which alleviates this irritation. For example, by defining a single vector

$$\mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \\ w_5 \\ w_6 \\ w_7 \end{bmatrix} = \begin{bmatrix} a \\ b_1 \\ b_2 \\ c_{11} \\ c_{12} \\ c_{21} \\ c_{22} \end{bmatrix} \tag{B.73}$$

the original function in Equation (B.72) can then be equivalently written as

$$g(\mathbf{w}) = \left(\mathbf{s}^T \mathbf{w}\right)^2 \tag{B.74}$$

where

$$\mathbf{s} = \begin{bmatrix} 1 \\ 1 \\ 2 \\ 1 \\ 3 \\ 3 \\ 9 \end{bmatrix}. \tag{B.75}$$

Again note that all we have really done here is *reindexed* the entries of the input vectors in a contiguous manner so that we can implement local optimization schemes in a less cumbersome way in a single line of algebra or `autograd` code, instead of requiring a loop over each input variable. This variable reindexing scheme is called *function flattening*, and is depicted visually in Figure B.16.

Figure B.16 Figure associated with Example B.14. A figurative illustration of flattening of the function given in Equation (B.72). See text for further details.

While performing the reindexing required to flatten a function is important, it (like derivative computation itself) is a repetitive and time-consuming operation for humans to perform themselves. Thankfully, the `autograd` library has a built-in module that flattens functions, which can be imported as shown below.

```
# import function flattening module from autograd
from autograd.misc.flatten import flatten_func
```

Below we define a Python version of the function in Equation (B.72).

```
# Python implementation of g
r = np.array([[1],[2]])
z = np.array([[1],[3]])
def g(input_weights):
    a = input_weights[0]
    b = input_weights[1]
    C = input_weights[2]
    return (((a + np.dot(r.T, b) + np.dot(np.dot(z.T, C), z)))**2)
        [0][0]
```

To flatten g we then simply call

```
# flatten an input function g
g_flat, unflatten_func, w = flatten_func(g, input_weights)
```

Here on the right-hand side `input_weights` is a list of initializations for input variables to the function g. The outputs `g_flat`, `unflatten_func`, and `w` are the flattened version of g, a module to unflatten the input weights, and a flattened version of the initial weights, respectively.