11

Differentiation and Adjoints

In this chapter, we describe efficient calculation of gradients for certain structured functions, particularly those that arise in the training of deep neural networks (DNNs). Such functions share some features with objective functions that arise in such applications as data assimilation and control, in both of which the optimization problem is integrated with a model of a dynamic process, one that evolves in time or proceeds by stages. In deep learning, the progressive transformation of each item of data as it moves through the layers of the network is akin to a dynamic process.

11.1 The Chain Rule for a Nested Composition of Vector Functions

We start by introducing some notational conventions for derivatives of vector-valued functions. For a function $h: \mathbb{R}^p \times \mathbb{R}^q \to \mathbb{R}^r$, we denote the partial gradient with respect to w at a point $(w, y) \in \mathbb{R}^p \times \mathbb{R}^q$ by $\nabla_w h(w, y)$. This is the $p \times r$ matrix whose ith column is the gradient of h_i with respect to w, for $i = 1, 2, \ldots, r$. Note that this matrix is the transpose of the Jacobian, which is the $r \times p$ matrix whose rows are $(\nabla_w h_i)^T$.

We now consider the chain rule for differentiation of a function that is a nested composition of vector functions. Given a vector $x \in \mathbb{R}^n$ of variables, suppose that the objective $f: \mathbb{R}^n \to \mathbb{R}$ has the following nested form:

$$f(x) = (\phi \circ \phi_l \circ \phi_{l-1} \circ \ldots \circ \phi_1)(x) = \phi(\phi_l(\phi_{l-1}(\ldots(\phi_1(x))\ldots), \quad (11.1)$$

where

$$\phi_1: \mathbb{R}^n \to \mathbb{R}^{m_1}, \quad \phi_i: \mathbb{R}^{m_{i-1}} \to \mathbb{R}^{m_i} \quad (i = 2, 3, \ldots, l), \quad \text{and } \phi: \mathbb{R}^{m_l} \to \mathbb{R}.$$

https://doi.org/10.1017/9781009004282.012 Published online by Cambridge University Press

The chain rule for calculating $\nabla f(x)$ yields the following formula:

$$\nabla f(x) = (\nabla_x \phi_1)\left(\nabla_{\phi_1}\phi_2\right)\left(\nabla_{\phi_2}\phi_3\right)\ldots\left(\nabla_{\phi_{l-1}}\phi_l\right)\left(\nabla_{\phi_l}\phi\right), \tag{11.2}$$

where all partial derivatives are evaluated at the current point x and all consistent values of $\phi_1, \phi_2, \ldots, \phi_l$. Since $f\colon \mathbb{R}^n \to \mathbb{R}$, the left-hand side $\nabla f(x)$ of (11.2) is a (column) vector in $\mathbb{R}^n$. Following the convention on derivative notation, we have the following shapes for the terms on the right-hand side of (11.2):

$\nabla_x \phi_1$ is a matrix of dimensions $n \times m_1$;

$\nabla_{\phi_i}\phi_{i+1}$ is a matrix of dimensions $m_i \times m_{i+1}$, for $i = 1, 2, \ldots, l-1$;

$\nabla_{\phi_l}\phi$ is a column vector of length m_l.

The matrix multiplications on the right-hand side of the formula (11.2) are all valid, and the product is a vector in $\mathbb{R}^n$.

The function evaluation formula (11.1) and the derivative formula (11.2) suggest the following scheme for evaluating the function f and its gradient.

Algorithm 11.1 Evaluation of a nested function and its gradient using the chain rule.

Given $x \in \mathbb{R}^n$;
Define $x_1 := \phi_1(x)$ and $A_1 := \nabla\phi_1(x)$;
for $i = 1, 2, \ldots, l-1$ **do**
  Evaluate

$$x_{i+1} := \phi_{i+1}(x_i) \quad \text{and} \quad A_{i+1} := \nabla_{\phi_i}\phi_{i+1}(x_i);$$

end for
Evaluate $f := \phi(x_l)$ and $p_l := \nabla_{\phi_l}\phi(x_l)$;
for $i = l, l-1, \ldots, 2$ **do**
  Define $p_{i-1} := A_i p_i$;
end for
Define $g := A_1 p_1$;
Output $f = f(x)$ and $g = \nabla f(x)$.

This scheme consists of a function evaluation loop that makes a *forward pass* through the succession of functions, and the derivative calculation loop that implements a *reverse pass*. During the forward pass, we store partial derivative information in the matrices A_i, $i = 1, 2, \ldots, l$, that are subsequently applied during the reverse pass to accumulate the product in (11.2). The scheme is efficient because it requires only matrix-vector multiplications.

https://doi.org/10.1017/9781009004282.012 Published online by Cambridge University Press

The total cost of the algorithm is approximately $nm_1 + \sum_{i=1}^{l-1} m_i m_{i+1}$ multiplications and additions, plus the cost of evaluating the functions and gradients. (A more naive scheme for evaluating $\nabla f(x)$ from the formula (11.2) might involve numerous matrix-matrix multiplications, not just the matrix-vector multiplications required by this scheme.)

11.2 The Method of Adjoints

We now consider a more general function evaluation model that captures those seen in simple DNNs and in other applications such as data assimilation. In this model, the variables do not all appear at the innermost level of nesting, as in (11.1). Rather, they are introduced progressively, at each stage of the function evaluation. We use the term *progressive functions* to denote functions with this structure. Despite the greater generality of this model, the process of evaluating the function and its gradient still contains a forward pass and a reverse pass and requires only a slight modification of Algorithm 11.1.

We consider a partition of the variable vector x as follows:

$$x = (x_1, x_2, \ldots, x_l), \quad \text{where } x_i \in \mathbb{R}^{n_i},\ i = 1, 2, \ldots, l, \tag{11.3}$$

so that $x \in \mathbb{R}^n$ with $n = n_1 + n_2 + \cdots + n_l$. A progressive function has the following form

$$f(x) = \phi(\phi_l(x_l, \phi_{l-1}(x_{l-1}, \phi_{l-2}(x_{l-2} \ldots (x_2, \phi_2(x_2, \phi_1(x_1)) \ldots), \tag{11.4}$$

where

$$\phi_1 : \mathbb{R}^{n_1} \to \mathbb{R}^{m_1}, \quad \phi_i : \mathbb{R}^{n_i} \times \mathbb{R}^{m_{i-1}} \to \mathbb{R}^{m_i}\ (i = 2, 3, \ldots, l), \quad \text{and } \phi : \mathbb{R}^{m_l} \to \mathbb{R}.$$

Stage i of the evaluation requires the variable subvector x_i together with the value of the function ϕ_{i-1}, which depends on the previous variables $x_{i-1}, x_{i-2}, \ldots, x_1$.

The dependence of f on the final subvector x_l is straightforward, but the chain rule is needed to recover derivatives with respect to other subvectors x_i, with more and more factors in the product as i decreases toward 1. Writing the gradients with respect to the last few subvectors x_l, x_{l-1}, x_{l-2}, x_{l-3}, we obtain

$$\begin{aligned}
\nabla_{x_l} f(x) &= \left(\nabla_{x_l} \phi_l\right) \left(\nabla_{\phi_l} \phi\right) \\
\nabla_{x_{l-1}} f(x) &= \left(\nabla_{x_{l-1}} \phi_{l-1}\right) \left(\nabla_{\phi_{l-1}} \phi_l\right) \left(\nabla_{\phi_l} \phi\right) \\
\nabla_{x_{l-2}} f(x) &= \left(\nabla_{x_{l-2}} \phi_{l-2}\right) \left(\nabla_{\phi_{l-2}} \phi_{l-1}\right) \left(\nabla_{\phi_{l-1}} \phi_l\right) \left(\nabla_{\phi_l} \phi\right) \\
\nabla_{x_{l-3}} f(x) &= \left(\nabla_{x_{l-3}} \phi_{l-3}\right) \left(\nabla_{x_{l-3}} \phi_{l-2}\right) \left(\nabla_{\phi_{l-2}} \phi_{l-1}\right) \left(\nabla_{\phi_{l-1}} \phi_l\right) \left(\nabla_{\phi_l} \phi\right).
\end{aligned}$$

https://doi.org/10.1017/9781009004282.012 Published online by Cambridge University Press

A pattern emerges. We see that in the expression for each $\nabla_{x_i} f$, $i = l, l-1$, $l-2, \ldots,$ the last factor is $\nabla_{\phi_l}\phi$ and the first factor is the partial derivative of ϕ_i with respect x_i. The intermediate terms are partial derivatives of one of the nested functions with respect to the next function in the sequence. The general formula is as follows:

$$\nabla_{x_i} f(x) = \left(\nabla_{x_i}\phi_i\right)\left(\nabla_{\phi_i}\phi_{i+1}\right)\left(\nabla_{\phi_{i+1}}\phi_{i+2}\right)\ldots\left(\nabla_{\phi_{l-2}}\phi_{l-1}\right)\left(\nabla_{\phi_{l-1}}\phi_l\right)\left(\nabla_{\phi_l}\phi\right). \tag{11.5}$$

(Note the similarity in the middle terms for different i – the same derivative matrices appear repeatedly in multiple expressions.) By extending Algorithm 11.1, we derive the efficient procedure shown in Algorithm 11.2 for computing $\nabla f(x)$. Algorithm 11.2 is efficient because it requires only matrix-vector multiplications and exploits fully the repeated structures seen in the middle terms of (11.5).

Algorithm 11.2 Efficient evaluation of a progressive function and its gradient using the chain rule.

Given $x = (x_1, x_2, \ldots, x_l) \in \mathbb{R}^{n_1+n_2+\ldots+n_l}$;
Evaluate $s_1 := \phi_1(x_1)$ and $B_1 := \nabla\phi_1(x_1)$;
for $i = 1, 2, \ldots, l-1$ **do**
  Evaluate

$$s_{i+1} := \phi_{i+1}(x_{i+1}, s_i), \quad A_{i+1} := \nabla_{\phi_i}\phi_{i+1}(x_{i+1}, s_i),$$
$$B_{i+1} := \nabla_{x_{i+1}}\phi_{i+1}(x_{i+1}, s_i);$$

end for
Evaluate $f := \phi(s_l)$ and $p_l := \nabla_{\phi_l}\phi(s_l)$;
for $i = l, l-1, \ldots, 2$ **do**
  Define $p_{i-1} := A_i p_i$, $g_i := B_i p_i$;
end for
Define $g_1 := B_1 p_1$;
Output $f = f(x)$ and $g = (g_1, g_2, \ldots, g_l) = \nabla f(x)$.

11.3 Adjoints in Deep Learning

The objective functions that arise in the training of the neural networks described in Section 1.6 have the progressive form (11.4) (albeit with different notation). Consider the supervised multiclass classification problem

https://doi.org/10.1017/9781009004282.012 Published online by Cambridge University Press

of Section 1.6, and suppose we are given m training examples (a_j, y_j), $j = 1, 2, \ldots, m$, where each a_j is a feature vector and $y_j \in \mathbb{R}^M$ is a label vector that indicates membership of a_j in one of M classes (see (1.20)). The loss function for training the neural network is a finite summation of m functions of the form (11.4), one for each training input. To be precise, given feature vector a_j, we can define $s_1^{(j)} = \phi_1(x_1; a_j)$ in (11.4) to be the output of the first layer of the DNN with a_j denoting the input and x_1 denoting the parameters in the first layer. We can define $s_{i+1}^{(j)} = \phi_{i+1}(x_{i+1}, s_i^{(j)})$, $i = 1, 2, \ldots, l-1$ as in Algorithm 11.2, ending with the outputs of the final layer, which is the vector $s_l^{(j)}$. Note that $m_l = M$; that is, the number of outputs from the final layer equals the number of classes M. To define the loss function for this example, we set

$$\phi^{(j)}(s_l^{(j)}) = -\left[\sum_{c=1}^{M} (y_j)_c (s_l^{(j)})_c - \log \sum_{c=1}^{M} e^{(s_l^{(j)})_c}\right], \tag{11.6}$$

while the overall loss function is

$$f(x) = \frac{1}{m} \sum_{j=1}^{m} \phi^{(j)}(s_l^{(j)}). \tag{11.7}$$

This is a slight generalization of our framework (11.4) in that this f is defined as the average of the loss functions over m training examples (not as a function based on a single training example), but note that the variables x are the same in each of these m terms, as are the functions $\phi_l, \phi_{l-1}, \ldots, \phi_2$. However, ϕ_1 is different for each of the m terms, because the feature vector a_j that is input to the first layer differs between terms. The function ϕ also differs between terms, because it is based on the label vector y_j for training example j. Algorithm 11.2 can, in principle, be applied to each function $\phi^{(j)}$, $j = 1, 2, \ldots, m$ to obtain $\nabla f(x)$. In practice, training is usually done with some variant of a stochastic gradient algorithm from Chapter 5 and we obtain an approximate gradient needed by these methods by taking a single term or a minibatch of terms from the summation in (11.7) and apply Algorithm 11.2 only to these terms.

11.4 Automatic Differentiation

Consider now a generalization of the approach in Section 11.2 in which the variables are not necessarily introduced progressively, stage by stage, and in which each stage may depend not just on the previous stage but on many prior stages. We use only the observation that the computation of the function f can

https://doi.org/10.1017/9781009004282.012 Published online by Cambridge University Press

be organized as a directed acyclic graph (DAG), called the *computation graph*, in which there exists an enumeration of the nodes in which each node depends only on lower-numbered nodes. (Any procedure for function evaluation that can be implemented computationally must necessarily admit a DAG structure.) For a function $f\colon \mathbb{R}^n \to \mathbb{R}$, we denote the quantity evaluated at node i by x_i, where the first n nodes are the n components of the variable vector x, and the final node (x_N, say) is the function value. Each step of the computation has the form

$$x_i = \phi_i(x_{\mathcal{P}(i)}), \quad i = n+1, n+2, \ldots, N, \tag{11.8}$$

where $\mathcal{P}(i)$ denotes the parents of node i in the computation graph – that is, the elements x_j, $j \in \mathcal{P}(i)$ that are required to evaluate x_i. The "evaluation" is usually an elementary operation; for example, a node could simply multiply its two inputs together or sum them, or it could take an exponential or sine of its single input. The DAG representation of this computation has N nodes, with directed arcs from each node in $\mathcal{P}(i)$ to node i, for all $i = n+1, n+2, \ldots, N$. The nodes are numbered such that $\mathcal{P}(i) \subset \{1, 2, \ldots, i-1\}$.

A example computation graph for $n = 3$ and $N = 10$ is shown in Figure 11.1. Here, x_1, x_2, and x_3 are the three independent variables, and we have

$$\mathcal{P}(4) = \{1,2\}, \quad \mathcal{P}(5) = \{1,2,3\}, \quad \mathcal{P}(6) = \{1,4\}, \quad \mathcal{P}(7) = \{2,4,5\},$$
$$\mathcal{P}(8) = \{3,5\}, \quad \mathcal{P}(9) = \{6,7,8\}, \quad \mathcal{P}(10) = \{6,9\}.$$

It is clear from (11.8) how the function f can be evaluated – by moving from left to right through the graph, evaluating the nodes in sequence. But how do we recover the gradient $\nabla f(x)$? As suggested in earlier sections, the key is to store additional information during the evaluation (11.8). As well as evaluating x_i, we store partial derivatives of x_i with respect to each of its arguments x_j, $j \in \mathcal{P}(i)$. Specifically, we label the arc from j to i by $\partial x_i / \partial x_j = \partial \phi_i / \partial x_j$, for

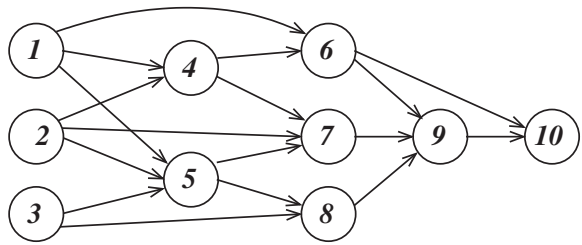

Figure 11.1 Computation graph for a function of three variables, with $N = 10$ nodes.

https://doi.org/10.1017/9781009004282.012 Published online by Cambridge University Press

each $j \in \mathcal{P}(i)$. The extra computation required for these partial derivatives is often minimal, no more than a few floating-point operations per arc.

Equipped with the partial derivative information, we can find $\nabla f(x)$ by performing a *reverse sweep* through the computation graph. At the conclusion of this sweep, node i of the graph will contain the partial gradient $\partial f/\partial x_i$, so that, in particular, the first n nodes will contain $\partial f/\partial x_i$, $i = 1, 2, \ldots, n$, which are the components of the gradient $\nabla f(x)$.

To do the reverse sweep, we introduce variables z_i at each node $i = 1, 2, \ldots, N$, initializing them to $z_i = 0$ for $i = 1, 2, \ldots, N-1$, and $z_N = 1$. At the end of the computation, each z_i will contain the partial derivative of f with respect to x_i. Since $f(x) = x_N$, we have, in fact, that $\partial f/\partial x_N = 1$, so $z_N = 1$ already contains the correct value. The sweep now proceeds through nodes $i = N, N-1, N-2, \ldots, 1$, as follows: At the start of step i, we have that $z_i = \partial f/\partial x_i$. We then update the variables z_j in the nodes j in the parent set $\mathcal{P}(i)$ as follows:

$$z_j \leftarrow z_j + z_i \frac{\partial \phi_i}{\partial x_j}, \quad \text{for all } j \in \mathcal{P}(i). \tag{11.9}$$

When the time comes to process a node i, the variable z_i contains the sum of the contributions from all of its of its children, since nodes $i+1, i+2, \ldots, N$ have been processed already. We thus have

$$z_i = \sum_{l\,:\,i \in \mathcal{P}(l)} z_l \frac{\partial \phi_l}{\partial x_i} = \sum_{l\,:\,i \in \mathcal{P}(l)} \frac{\partial f}{\partial \phi_l} \frac{\partial \phi_l}{\partial x_i}, \tag{11.10}$$

the second equality being due to the fact that z_l contains the value $\partial f/\partial x_l$ at the time that z_i is updated, since $i \in \mathcal{P}(l)$. Since the formula (11.10) captures the total dependence of f on x_j, and since $j \in \mathcal{P}(i)$ only when $i > j$, we have by an inductive argument that when z_j has gathered all the contributions from its child nodes i, it too contains the partial derivative $\partial f/\partial x_j$. The formula (11.10) is essentially the chain rule for $\partial f/\partial x_j$.

Returning to the example in Figure 11.1, we see that the partial function evaluations at nodes $4, 5, \ldots, 10$ can be carried out in numerical order, and the partial derivatives can be evaluated in the reverse order $10, 9, 8, \ldots, 4, 3, 2, 1$.

What we have described in this section is the *reverse mode* of automatic differentiation (also known as "computational differentiation" and "algorithmic differentiation"). (The technique is often called "back-propagation" in the machine learning community.) This technique and many other issues in automatic differentiation are explored in the monograph of Griewank and Walther (2008). The reverse mode has the remarkable property that the computational cost of obtaining the gradient ∇f is bounded by a small

https://doi.org/10.1017/9781009004282.012 Published online by Cambridge University Press

multiple of the cost of evaluating f. This fact can be deduced readily from the facts that (a) each arc in the computation graph corresponds to just one or a few floating-point operations during the evaluation of f; (b) labeling the arc from j to i with the partial derivative $\partial x_i / \partial x_j$ requires just one or a few additional floating-point operations; (c) the reverse sweep visits each arc exactly once, and the update formula (11.9) shows that in general two operations (one addition and one multiplication) are associated with each arc.

The chief drawback of the reverse mode is its space complexity. The procedure, as previously described, requires storage of the complete computation graph, including storage for x_i and z_i, $i = 1, 2, \ldots, N$ and the arc labels $\partial \phi_i / \partial x_j$, so that the storage requirements grow linearly with the time required to evaluate f. Such requirements can be prohibitive for some functions. This issue can be solved with the use of "checkpointing," which is essentially a process of trading storage for extra computation. At a checkpoint, we save only those nodes of the computation graph that will be needed in subsequent evaluations and discard the rest. When the reverse sweep reaches this point, we recalculate the discarded nodes, allowing the reverse sweep to continue to an earlier checkpoint. Details are given in Griewank and Walther (2008).

11.5 Derivations via the Lagrangian and Implicit Function Theorem

We examine here an alternative viewpoint on the progressive function (11.4) based on a reformulation as an equality constrained optimization problem. We also discuss algorithmic consequences of this reformulation, and several extensions.

11.5.1 A Constrained Optimization Formulation of the Progressive Function

Returning to the progressive functions defined in (11.4), suppose that our task is to find a stationary point for this function – that is, a point where $\nabla f(x) = 0$. By introducing variables s_i to store intermediate evaluation results, we can formulate this problem as the following equality-constrained optimization problem:

$$\min_{x,s} f(x,s) := \phi(s_l) \quad \text{s.t. } s_1 = \phi_1(x_1), \; s_i = \phi_i(x_i, s_{i-1}), \; i = 2, 3, \ldots, l, \tag{11.11}$$

https://doi.org/10.1017/9781009004282.012 Published online by Cambridge University Press

where $s = (s_1, s_2, \ldots, s_l)$ and $x = (x_1, x_2, \ldots, x_l)$. By introducing Lagrange multiplier vectors $p_1, p_2, \ldots, p_l$ for the constraints in (11.11), we can write the Lagrangian for this problem as

$$\mathcal{L}(x, s, p) = \phi(s_l) - \sum_{i=2}^{l} p_i^T (s_i - \phi_i(x_i, s_{i-1})) - p_1^T (s_1 - \phi_1(x_1)). \quad (11.12)$$

First-order conditions for (x, s) to be a solution of this problem are obtained by taking partial derivatives of the Lagrangian with respect to x, s, and p and setting them all to zero. These partial derivatives are as follows:

$$\nabla_{x_1} \mathcal{L} = B_1 p_1, \quad \text{where } B_1 = \nabla \phi_1(x_1), \quad (11.13a)$$

$$\nabla_{x_i} \mathcal{L} = B_i p_i, \quad \text{where } B_i := \nabla_{x_i} \phi_i(x_i, s_{i-1}), \ i = 2, 3, \ldots, l, \quad (11.13b)$$

$$\nabla_{p_i} \mathcal{L} = -s_i + \phi_i(x_i, s_{i-1}), \ i = 2, 3, \ldots, l, \quad (11.13c)$$

$$\nabla_{p_1} \mathcal{L} = -s_1 + \phi_1(x_1), \quad (11.13d)$$

$$\nabla_{s_i} \mathcal{L} = -p_i + A_i p_{i+1}, \quad \text{where } A_i := \nabla_{s_i} \phi_{i+1}(x_{i+1}, s_i), \ i = 1, 2, \ldots, l-1, \quad (11.13e)$$

$$\nabla_{s_l} \mathcal{L} = -p_l + \nabla \phi(s_l). \quad (11.13f)$$

Note the close relationship between this nonlinear system of equations and Algorithm 11.2. By setting the partial derivatives w.r.t. p_i to zero, we obtain the equality constraints in (11.11), which are satisfied when the s_i are defined by the forward pass in Algorithm 11.2. By setting the partial derivatives w.r.t. s_i to zero, we obtain the so-called adjoint equation that defines the p_i, which are identical to those obtained from the reverse sweep in Algorithm 11.2. Finally, the partial derivatives of $\mathcal{L}$ w.r.t. x_i yield the same formulas as for the gradient expressions in Algorithm 11.2. Thus, all terms in (11.13) are zero when, in the notation of (11.4), we have $\nabla f(x) = 0$ – that is, x is a stationary point for f.

The constrained optimization perspective can have an advantage when the formulation is complicated by the presence of constraints (equalities and inequalities) involving s as well as x, or by slightly more general structure than is present in (11.4). In such situations, the first-order conditions (11.13) contain complementarity conditions, which can be handled by an interior-point framework while still retaining the advantages of sparsity and structure in the Jacobian of the nonlinear equations (11.13) that lead to efficient calculation of steps. In the unconstrained formulation, reduction of the constraints to formulas involving x alone, even when this is possible, can lead to loss of structure in the constraints and thus loss of efficiency in algorithms based on (11.4).

https://doi.org/10.1017/9781009004282.012 Published online by Cambridge University Press

11.5.2 A General Perspective on Unconstrained and Constrained Formulations

We generalize the technique of the previous subsection by considering an unconstrained problem

$$\min\ f(x) \tag{11.14}$$

(for $x \in \mathbb{R}^n$) that can be rewritten equivalently as the following constrained formulation, as follows:

$$\min_{x,s}\ F(x,s) \quad \text{s.t. } h(x,s) = 0, \tag{11.15}$$

where $s \in \mathbb{R}^p$, and $h\colon \mathbb{R}^n \times \mathbb{R}^p \to \mathbb{R}^p$ uniquely defines s in terms of x. Because of the latter property, we can write $s = s(x)$, where $h(x(s),s) = 0$ for all s, so that the objective in (11.15) becomes $F(x,s(x))$, and

$$f(x) = F(x,s(x)). \tag{11.16}$$

Under appropriate assumptions of smoothness and nonsingularity of the $p \times p$ matrix $\nabla_s h(x,s(x))$, we have from the implicit function theorem (see Theorem A.2 in the Appendix) that

$$\nabla_x s(x) = -\nabla_x h(x,s(x))[\nabla_s h(x,s(x))]^{-1}. \tag{11.17}$$

(This can be see by taking the total derivative of h with respect to x and setting it to zero; that is, $0 = \nabla_x h(x,s(x)) + \nabla_x s(x)\nabla_s h(x,s(x))$.) By substituting (11.17) into (11.16), we obtain

$$\begin{aligned}\nabla f(x) &= \nabla_x F(x,s) + \nabla_x s(x)\nabla_s F(x,s)\\ &= \nabla_x F(x,s) - \nabla_x h(x,s(x))[\nabla_s h(x,s(x))]^{-1}\nabla_s F(x,s),\end{aligned} \tag{11.18}$$

The problem (11.11) is a special case of (11.15), in which $\nabla_s h(x,s)$ is a block-bidiagonal matrix with identity matrices on the diagonal. Thus, the inverse $[\nabla_s h(x,s(x))]^{-1}$ is guaranteed to exist. We can show that (11.18) leads to the same formula for $\nabla f(x)$ as was obtained by Algorithm 11.2 for (11.4). Details are left as an Exercise.

11.5.3 Extension: Control

By a slight extension to the framework (11.11), we can define *discrete-time optimal control*, an important class of problems in engineering and, more recently, in machine learning. The only essential difference is that the objective

https://doi.org/10.1017/9781009004282.012 Published online by Cambridge University Press

depends not just on the s_l but possibly on all variables x_i, $i = 1, 2, \ldots, l$ and all intermediate variables s_i, $i = 1, 2, \ldots, l$, so we have

$$\min_{x,s} f(x,s) := \phi(x,s) \tag{11.19a}$$

$$\text{subject to } s_1 = \phi_1(x_1), \quad s_i = \phi_i(x_i, s_{i-1}), \; i = 2, 3, \ldots, l. \tag{11.19b}$$

In the language of control, the variables x_i are referred to as *controls* or *inputs*, whose purpose is to influence the evolution of a dynamical system that is described by the functions ϕ_i. The variables s_i are called the *states* of this system. This is usually some known initial state s_0 (not included in the formulation above because it is fixed), and other states are fully determined by the equations in (11.19).

The problem (11.19) has the form (11.15), where again the Jacobian $\nabla_s h(x,s)$ has block-bidiagonal structure with identity matrices on the diagonal, so that it is structurally nonsingular. Algorithms for solving (11.19) can thus make use either of the unconstrained perspective or the constrained perspective. The latter is often more useful in the case of control, as many problems have constraints on the states s_i as well as the controls x_i, and these can be handled more efficiently in the constrained formulations. (Even bound constraints on s_i would convert to complicated constraints on the controls x_i, and elimination of the s_i, as is done in the unconstrained formulation, would cause the stagewise structure to be lost.)

Notes and References

The monograph of Griewank and Walther (2008) is the standard reference on computational differentiation. The widespread use of ReLU activations in neural networks introduces some complications into the derivative computation, as the functions are not smooth! The same ideas as described in Sections 11.2 and 11.3 obtain, but the concept of "derivative" needs to be generalized considerably. Generalizations are discussed in Griewank and Walther (2008, chapter 14), focusing on the Clarke subdifferential. The latter generalization, used also by David et al. (2020), analyzes the convergence of a stochastic subgradient method based on this generalization in a framework that can be applied to neural networks with ReLU activations. Later work (Bolte and Pauwels, 2020) makes use of "conservative fields" as generalizations of derivatives (the Clarke subdifferential is a "minimal conservative field," a kind of special case). The latter paper gives details of the generalization of

https://doi.org/10.1017/9781009004282.012 Published online by Cambridge University Press

the reverse mode of automatic differentiation and of the convergence of a minibatch variant of the stochastic gradient method.

Efficient solution of optimal control problems that exploit the stagewise structure are discussed in (Rao et al., 1998), including variants in which additional constraints are present at each stage of the problem.

Exercises

1. By expressing (11.11) in the form (11.15), show that the formula (11.18) leads to the same gradient of f as is calculated in Algorithm 11.2. Explain in particular why the matrix $\nabla_s h(x, s)$ is nonsingular for the particular function h from (11.11).
2. Sketch the computation graphs for the nested function (11.1) and the progressive function (11.4), in a format similar to Figure 11.1.
3. By expressing (11.19) in the form (11.15), derive an expression for the gradients with respect to $x_1, x_2, \ldots, x_l$ of the version of this problem in which the states s_i are eliminated.
4. Consider a DNN with ResNet structure, in which there are connections not just between adjacent layers but also connections that skip one layer, connecting the transformed output of the neurons at layer i to the input at layer $i + 2$. Write down the extension of the constrained formulation (11.11) to this case. By working with the implicit function theorem techniques of Section 11.5.2, derive expressions for the total derivative of the objective function with respect to the parameters of the DNN.

https://doi.org/10.1017/9781009004282.012 Published online by Cambridge University Press