Practical aspects

Waldemar Hebisch

January 25, 2022

1 Automatic differentiation

Usual continuous optimization methods are based on computing gradient. Gradient requires extra code and sometimes straightforward code to compute gradient may be inefficient. However, given reasonable procedure for computing f (say as code in a programming language), there is method to transform such procedure into procedure computing gradient of f. This method is called "automatic differentiation".

Automatic differentiation has interesting property: number of operations needed to compute f and gradient of f together is about twice as large as number of operations needed to compute f alone. To present this more precisely we need notion of straight line programs.

Simple approach to computing derivatives is as follows: we write F as composition

$$F = G_n \circ G_{n-1} \circ \cdots \circ G_1.$$

Then

$$F' = G'_n \cdot G'_{n-1} \cdot \cdots \cdot G'_1.$$

This works, but in multivariate case implied matrix multiplication may be expensive.

1.1 Straight line programs

Straight line program is a sequence of nodes numbered from 1 to n. When executing program each node is assigned a value v_i. First m nodes are input nodes, values a those nodes are input data. For i bigger than m node number i have associated function f_i. f_i has k_i arguments. There are numbers $j_{i,1}, \ldots, j_{i,k_i}$ such that $j_{i,l} < i$ and

$$v_i = f_i(v_{j_{i,1}}, \ldots, v_{j_{i,k_i}}).$$

In other words, value at node i is computed from values at previous nodes using f_i. In general we designate some nodes (possibly several) as output nodes and consider vector formed from those values as value computed by straight line program. In important special case there is single (scalar) value, say value at last node.

We can also embed constants in straight line program, this means that corresponding f_i has no inputs.

We can visualize straight line program as a directed graph such that there is arc from j to i when value at node j is used directly to compute value at node i, that is $j \in \{j_{i,1}, \ldots, j_{i,k_i}\}$.

We have formulated definition of straight line program is rather abstract way, but usually functions f_i are relatively simple. In fact usually at most nodes we have multiplications and additions. But we may also have other functions like divisions by nonzero number or exponential. Our intention is that f_i are smooth easily computable functions. For later use we assume that gradients of f_i are easily computable.

In such setting size of graph corresponding to a straight line program measures cost of computations. That is we assume that cost is proportional to number of arcs (sometimes cost is measures using number of nodes, but in our setting number of arcs may by much larger than number of nodes).

At first glance definition of straight line program looks quite restrictive: there are no loops, no conditionals, no functions. However, it is natural and relevant for numerical computations. Namely, if function is given by exact formula, than data dependent branches just represent alternative definitions. When computing value in neighbourhood of single point we can just follow on branch. In fact, without data dependent branches we can follow single execution path and treat it as straight line program. So, for theoretical analysis we can treat large proportion of numerical programs as straight line programs.

We want to show that given straight line program computing single scalar function F we can compute ∇F with comparable effort, that is there exist straight line program computing ∇F with size comparable to straight line program computing F. Clearly, it would be false if f_i had many arguments and gradient of f_i required a lot of effort to compute. To avoid this difficulty we assume that gradients of f_i are easy to compute. This is true for typical functions of single variable like exponential. Also, gradients of functions like multiargument sum, linear combination or product are easy to compute. Actually, it is not obvious that gradient of multiargument product is easy to compute, but we can replace multiargument product by tree of two argument products: this increases number of nodes but preserves number of arcs.

We represent our assumption about gradient of f_i by introducing single node for each coordinate of gradient of f_i and *single* arc joining this node with node i. Gradient normally depends all arguments of f_i, so to properly represent dependence we would need arcs from each coordinate of gradient of f_i to each argument of f_i. However, single arc per coordinate of gradient adequately represent our assumption about cost of computing gradient. In practice, we frequently compute gradient of f_i together with f_i.

Lemma 1.1 *Assume that f can be computed by straight line program having N arcs. With assumption about gradients of f_i as above we can compute gradient of F using straight line program having at most $5N$ arcs.*

Proof: By assumption we have straight line program P computing F. Without loss of generality we may assume that output of F is from last node n (otherwise we can drop nodes after output node decreasing N). We build straight line program for computing ∇F is the following way: first we take all nodes and arcs from P. Next, for each i we add nodes corresponding to coordinates of gradient of g_i and add arc joining this node with i. Assuming that $j < i$ is among arguments of f_i we will denote by $g_{j,i}$ corresponding coordinate of gradient of f_i. Next, for each node i of original program P we introduce new node for value d_i (we will give formula later). For each $g_{j,i}$ we add extra node and two arcs: value $h_{j,i}$ at new node is product of $g_{j,i}$ and d_i (so one of new arcs joins $g_{j,i}$ with $h_{j,i}$, the other joins d_i with $h_{j,i}$. Value at d_n (where node n is output node of P) is 1. For $j \neq n$ value d_j is sum of $h_{j,i}$ over all i. This means that for each $h_{j,i}$ we have arc from $h_{j,i}$ to d_j. Finally, output of our program is vector $(d_1, \ldots, d_m)$ (recall that $1, \ldots, m$ correspond to input nodes).

To satisfy condition of our definition of straight line program we need to renumber nodes so that value at given node depends only on values at nodes with smaller node number. It is relatively easy, we keep numbering of nodes of P. After that we allocate (in arbitrary order) numbers for $g_{j,i}$. We need to put d_i in opposite order compared to P. We need to put $h_{j,i}$ after d_i and before d_j this is clearly possible.

Now, concerning number of arcs: we have arcs from P and added arcs. For each arc (j, i) from P we added 4 new arcs: one for coordinate of gradient joining i with $g_{j,i}$. Two for product $h_{j,i}$ and one from product to d_j. So together we have $5N$ arcs.

It remains to prove that our program computes gradient. Before this let us note that d_j is given by the following formulas: $d_n = 1$ and

$$d_j = \sum_{i>j} g_{j,i} d_i$$

for $j < n$.

Now, we prove our claim by induction over N. For $N = 0$ claim is obvious: if v_n is among input values than n-th coordinate of gradient is 1 and other are 0. But $d_n = 1$ and in formula above for $j < n$ there are no terms in sum, so sum is 0 so $d_j = 0$. So we have equality in this case. If v_n is not among input values, then it is a constant and gradient is 0. Again relevant d_j are 0 and we have equality. When $N > 0$ we consider first node j such that f_j is not a constant. Let G be function computed by straight line program Q where Q is like P but without f_j so all nodes from 1 to j are input nodes of Q. We have

$$F(v_1, \ldots, v_m) = G \circ H$$

where H is vector function such that $H_k = v_k$ for $k < j$ and $H_j = f_j$. We have

$$\partial_l F = \sum_{k=1}^{j} (\partial_k G)(\partial_l H_k) = \partial_l G + (\partial_j G)\partial_l f_j.$$

By inductive assumption $d_j = \partial_j G$. Also, by inductive assumption for $l < j$

$$\partial_l G = \sum_{i>j} g_{l,i} d_i.$$

Note that in Q there are no arcs from l to nodes with number smaller or equal to j so we can write sum in form above. Now

$$\partial_l F = \partial_l G + (\partial_j G)\partial_l f_j = \left(\sum_{i>j} g_{l,i} d_i\right) + g_{l,j} d_j$$

$$\sum_{i>l} g_{l,i} d_i = d_l$$

if f_j depends on l. Also, when f_j does not depend on l we get the same equality. So

$$\partial_l F = d_l$$

which means that P indeed computes gradient. □

Remark: Number $5N$ looks much larger then N. However, we allow complex f_j while added nodes are quite simple. For example, when f_j is exponential then computing it requires several (say 10) simple machine operations. And when nodes in original program are simple we can simplify program for derivative. For exponential we need no extra operations to compute derivative and just node (two arcs) to compute product with d_j and one arcs for sum. When f_j is a sum than coordinate of gradient all are 1 and we could skip corresponding arcs and multiplication. So for each term of sum we just add one arc to program computing derivative. For two argument product derivative with respect to first argument is just second argument, so we need no extra nodes and arcs for coordinates of gradient.

The lemma above is frequently formulated in terms of nodes and assuming special form of nodes. Then result looks better. But this is really matter of constant factors and in practice issue of constants is more tricky. Namely, in many cases it is possible to simplify resulting program computing derivative lowering number of operations and compute time. On the other hand, to compute derivative we may be forced to store more intermediate values than for computing F. Consequently, pattern of memory accesses may be less cache friendly. In principle, pattern of memory accesses may produce constant factor much larger than 5.

Remark: Lemma above is also called Telagen's transposition principle.

Example: Mutilayer perception (neural network). We have input x_0 and layers x_i, $i = 1, \ldots n$ where x_n is output. Each x_i is obtained from x_{i-1} by composition of linear transformation A_i with nonlinear function H_i acting separately on each coordinate:

$$x_i = H_i A_i x_{i-1},$$

$$H_i(y)_j = h_{i,j}(y_j).$$

Note: in many cases all $h_{i,j}$ are equal, say all are sigmoids. A_i usually are affine, that is contain also additive term. But for theoretical purposes we can assume that A_i are linear by enlarging A_i and adding to x_{i-1} extra coordinate equal to 1. In neural network training x_0 comes from training set and we try to minimize error

$$L(x_0, y, A_1, \ldots, A_n) = \|x_n - y\|^2$$

where y comes from training set. During training A_i form input to our function and we need to compute gradient with respect to A_i. Clearly

$$\nabla_{x_n} \|x_n - y\|^2 = 2(x_n - y),$$

$$\partial_{A_n} \|x_n - y\|^2(B) = \langle H'_n B x_{n-1}, 2(x_n - y)\rangle,$$

$$\partial_{x_{n-1}} \|x_n - y\|^2(v) = \langle H'_n A_n, 2(x_n - y)\rangle,$$

and we can compute derivatives with respect to other A_i using similar formulas and and chain rule. When done in naive way, this leads to product of large matrices which can be expensive to compute. Our lemma tells us that gradient can be computed at cost comparable to cost of computing L (assuming that derivatives of $h_{i,j}$ are no more expensive to compute than $h_{i,j}$). For gradient we can write:

$$\nabla_{A_n} \|x_n - y\|^2 = 2x_{n-1} \otimes (H'_n(x_n - y))$$

where $v \otimes w$ denotes matrix with entries $m_{i,j} = v_i w_j$ (in general above we would have transpose of H'_n but we used fact that H'_n is diagonal). Also

$$\nabla_{x_{n-1}} \|x_n - y\|^2 = A_n^T H'_n(2(x_n - y)).$$

Now

$$\nabla_{A_{n-1}} \|x_n - y\|^2 = 2x_{n-2} \otimes \left(H_{n-1} A_n^T H'_n(x_n - y)\right).$$

Note that $H_{n-1} A_n^T H'_n(x_n - y)$ is just sequence of applications of matrices to vector so the formula above requires number of operations proportional to sum of sizes of A_n and A_{n-1}. We have similar formulas for other A_i:

$$\nabla_{x_i} \|x_n - y\|^2 = A_{i+1}^T H'_{i+1} A_{i+2}^T H'_{i+2} \ldots A_n^T H'_n(2(x_n - y)).$$

$$\nabla_{A_i} \|x_n - y\|^2 = 2x_{i-1} \otimes \left(A_{i+1}^T H'_{i+1} A_{i+2}^T H'_{i+2} \ldots A_n^T H'_n(x_n - y)\right).$$

We can reuse products for $i+1$ computing gradient with respect to A_i, so cost is proportional to sum of sizes of all A_i.

Remark: This is backpropagation method which was originally introduced in somewhat different way. Later it was observed that backpropagation is actually equivalent to use formulas from automatic differentiation. However, vector formulation above allows use of standard optimized linear algebra routines.

Remark: Main gain is due to correct ordering of products: computing product of matrices first and then applying it to vector would lead to much larger execution time.

1.2 Sparse matrices

In several optimization problems significant part of cost is due to linear algebra. In practical problem in many cases there are sparse matrices. Algorithm that take advantage of sparse matrices in practice may by much faster than more general ones.

1.3 Remarks about derivative-free methods

There are methods which find optimum of f without using derivatives. However most of such methods implicitly assume existence of derivatives, namely they only work well for regular functions which have derivatives. For irregular functions in high dimension currently best methods are subgradient methods. Derivative-free methods were proposed for case when there is procedure to compute f, but procedure to compute gradient of f is not available. One class of "derivative-free" methods uses finite differences to approximate gradient. Such methods may work reasonably well, but usually are less efficient than alternatives. Namely, automatic differentiation usually is able to produce efficient procedure for computing gradient.

As explained number of operations needed to compute f and gradient of f together is a constant multiple of number of operations needed to compute f. Since gradient of f has n coordinates, computing f and finite difference approximation to gradient of f requires $n+1$ computations of f. For large n this is much bigger than number of operations needed by automatic differentiation. Other factors may affect execution time, but when automatic differentiation is available it is usually better than finite difference approximations.

There is another possible case where derivative-free methods could be useful. Namely, sometimes functions are computed using stochastic simulations. Values obtained via simulations have substantial (pseudo)random error and consequently finite differences tend to produce bad approximation to gradient. Similarly, automatic differentiation have trouble with random numbers. However, in such case stochastic gradient methods usually works well.