Autodiff

Consider the product of three matrices w/ dimensions:

$$\underset{1\times n}{A} \cdot \underset{n\times n}{B} \cdot \underset{n\times n}{C}$$

We can evaluate the product numerically from left to right or R to L. To understand computational load, lets review flop counts for basic matrix / vector operations (see supplement at end for more details.)

Let a, b vectors in $\mathbb{R}^n$, $A \in \mathbb{R}^{n\times n}$, $B \in \mathbb{R}^{n\times p}$

- $a^T b$: $2n$ flops (n multiplies, $n-1$ adds)
- Ab : $2mn$ flops (each row is $2n$ flops, m rows)
- AB : $2mnp$ flops ($AB = A[b_1, .., b_p]$; Ab_1 is $2mn$ flops times p colums of B.)

Returning to our matrix product ABC, here are the flop counts for evaluating in different directions.

L to R

$$\underbrace{\underbrace{A \cdot B}_{\substack{\text{result: } 1\times n \\ (1)}} \cdot C}_{(2)}$$

(1) : $2n^2$ flops

(2) : $2n^2$ flops

R to L

$$\underbrace{A \cdot \underbrace{B \cdot C}_{\substack{\text{Result: } n\times n \\ (1)}}}_{(2)}$$

(1) $2n^3$ flops

(2) $2n^2$ flops

More generally, consider product $A \prod_{i=1}^{k} B_i$, $A \in \mathbb{R}^{1\times n}$, $B \in \mathbb{R}^{n\times n}$

L to R

flops for each product:

$$\underbrace{\underbrace{A \cdot B_1}_{(1)\ \text{result: } 1\times n} \cdot B_2}_{(2)\ \text{result: } 1\times n} \cdots$$

(1) $2n^2$

(2) $2n^2$

⋮

(k) $2n^2$

$$\boxed{\text{Total} = 2kn^2}$$

R to L

$$A \cdots \underbrace{B_{k-2} \cdot \underbrace{B_{k-1} \cdot B_k}_{(1)\ \text{result: } n\times n}}_{(2)\ \text{result: } n\times n}$$

(1) $2n^3$

(2) $2n^3$

⋮

(k-1) $2n^3$

(k) $2n^2$

$$\boxed{\text{Total} \approx 2kn^3}$$

Consider now, computing the derivative of a composition of functions.

$$F(x) = f(g(h(x)))$$

We have

$$D_x F = \underbrace{Df(y)\Big|_{y=g(h(x))}}_{1\times n} \underbrace{Dg(y)\Big|_{y=h(x)}}_{n\times n} \underbrace{Dh(y)\Big|_{y=x}}_{n\times n}$$

Suppose they have dimensions:

<u>Forward mode (R to L)</u>

Flops : n^3

Memory : Small. $h(x)$ is evaluated to compute the 2^{nd} Jacobian, but it can be forgotten afterwords. For longer function compositions, this means you evaluate a "forward pass" in tandem w/ the Jacobian multiplies to get the base points for the derivatives. But you can discard it right after you use it.

<u>Reverse mode</u> (L to R)

Flops : n^2

Memory: Have to do a "forward pass" first, then remember all the intermediate results to use as base points for the intermediate derivatives

Example:

Consider a simple neural net w/ 1 hidden layer.

$$F(x) = f_3(f_2(f_1(x)))$$

$$f_1(z) = W_1 z + b_1$$
$$f_2(z) = \sigma(z) \quad (\text{elementwise})$$
$$f_3(z) = W_2 z + b_2$$

Lets compute $D_x F$. To do this, first compute closed form of intermediate derivatives:

$$D_x f_1 = W_1$$

To see this, let $g = f_1$ (so we dont triple index)

$$D_x g = \begin{pmatrix} \frac{\partial g_1}{\partial x_1} & \cdots & \frac{\partial g_1}{\partial x_n} \\ \vdots & & \\ & \cdots & \frac{\partial g_n}{\partial x_n} \end{pmatrix}$$

$$g_i(x) = \underbrace{(W_1)_i}_{i\text{-th row}} x = \sum_k W_{ik} x_k$$

$$\frac{\partial g_i}{\partial x_j} = W_{ij} \implies D_x g = W.$$

$$D_x f_2 = D_x \sigma(x) = (\sigma_1(x) \dots \sigma_n(x)), \text{ where } \sigma \text{ is applied elementwise}$$

$$\frac{\partial \sigma_i}{\partial x_j} = \begin{cases} \sigma'(x_i) & j = i \\ 0 & \text{else} \end{cases}$$

$$\Rightarrow \quad D_x \sigma(x) = \operatorname{diag}(\sigma'(x_1) \dots \sigma'(x_n))$$

where

$$\sigma'(z) = \sigma(z)(1 - \sigma(z)) \leftarrow \text{Just compute derivative of } \sigma \text{ \& massage}$$

We can now write straightforward code computing the derivative in forward & reverse mode & compare execution times. (See accompanying code files.)

10-725/36-725: Convex Optimization	**Fall 2015**

Lecture 19: November 5

Lecturer: Ryan Tibshirani *Scribes: Bohan Li, Donghan Yu, Ge Huang*

Note: *LaTeX template courtesy of UC Berkeley EECS dept.*

Disclaimer: *These notes have not been subjected to the usual scrutiny reserved for formal publications. They may be distributed outside this class only with the permission of the Instructor.*

19.1 Flops for basic operations

Complexity can be expressed in terms of *floating point operations* or *flops* required to find the solution. A flop serves as a basic unit of computation, which could denote one addition, subtraction, multiplication or division of floating point numbers. Note that, the flop count is just a rough measure of how expensive an algorithm can be. Many more aspects need to be taken into account to accurately estimate practical runtime. And in practical situations, we're interested in rough, not exact flop counts to measure the complexity of operations.

In the following sections, we'll show the flop count of some basic operations.

19.1.1 Vector-vector opertaions

Given vector $a, b \in \mathbb{R}^n$:

- Addition $a + b$: requires n flops for n element-wise additions.
- Scalar multiplication $c \cdot a$: requires n flops for n element-wise multiplications.
- Inner product $a^T b$: requires approximately $2n$ flops for n multiplications and $n - 1$ additions.

However, as said above, the flop count is just a rough measure of how expensive an algorithm can be. For example, setting every element of vector a to 1 costs 0 flops.

19.1.2 Matrix-vector opertaions

Given $A \in \mathbb{R}^{m\times n}, b \in \mathbb{R}^n$, consider Ab:

- In general, $Ab = (a_1^T, a_2^T, \cdots, a_m^T)^T b = (a_1^T b, a_2^T b, \cdots, a_m^T b)^T$, each row takes $2n$ flops, then m rows take $2mn$ flops in total.
- If A is s-sparse, then the i'th element of Ab is $Ab(i) = \sum_j a_{ij} b_j$, $(i, j) \in S$, where S is the index set of non-zero elements in A. Since $|S| = s$, the total flop count is $2s$. (The worst case is that all the non-zero elements are in the same row)

- If $A \in \mathbb{R}^{n\times n}$ is k-banded, the non-zero elements of each row is $2k$, then the total flop count of n row is $2nk$.
- If $A = \sum_{i=1}^{r} u_i v_i^T \in R^{m\times n}$, $Ab = \sum_{i=1}^{r} u_i(v_i^T b)$. Calculate $m_i = v_i^T b,\ i = 1, \cdots, r$ costs $2nr$ flops. Then scalar multiplication takes mr flops, finally the summation takes mr flops. The total flop count is $2r(m+n)$.
- If $A \in \mathbb{R}^{n\times n}$ is a permutation matrix, it takes 0 flops to reorder elements in b.

19.1.3 Matrix-matrix product

Given $A \in \mathbb{R}^{m\times n}, B \in \mathbb{R}^{n\times p}$, consider AB:

- In general, $AB = A(b_1, b_2, \cdots, b_p) = (Ab_1, \cdots, Ab_p)$. For each b_i, the product cost $2mn$ flops. Then the total flop count is $2mnp$.
- If A is s-sparse, it costs $2sp$ flops. The cost can be further reduced if B is also sparse.

19.1.4 Matrix-matrix-vector product

Given $A \in \mathbb{R}^{m\times n}, B \in \mathbb{R}^{n\times p}, c \in \mathbb{R}^p$, consider ABc:

- If product is done properly, that is, $ABc = A(Bc)$, the total cost is $2np + 2mn$. Else if done improperly, i.e., $ABc = (AB)c$, the cost is $2mnp + 2mp$!

19.2 Solving linear systems

Given a non-singular square matrix $A \in \mathbb{R}^{n\times n}$ and a vector $b \in \mathbb{R}^n$, consider solving the linear equation, $Ax = b$. In others words, we intend to determine the cost of computing $x = A^{-1}b$. Note that in Newton's method, we need to solve $\nabla^2 f(x)v = -\nabla f(x)$, which is exactly this form.

- In general, it cost n^3 flops. This can be a very expensive cost when n is a large number. However, the complexity of solving linear systems can be reduced for some matrices having special properties.
- If A is diagnal, it just costs n flops, one each for element-wise divisions. $x = (b_1/a_1, \cdots, b_n/a_n)$.
- If A is lower triangular $(A_{ij} = 0, j > i)$, it costs about n^2 flops by forward substitution.

$$
\begin{aligned}
x_1 &= b_1/A_{11} \\
x_2 &= (b_2 - A_{21}x_1)/A_{22} \\
&\cdots \\
x_n &= (b_n - A_{n,n-1}x_{n-1} - \cdots - A_{n,1}x_1)/A_{nn}
\end{aligned}
$$

- If A is upper triangular $(A_{ij} = 0, j > i)$, it costs about n^2 flops by back substitution.
- If A is s-sparse, it often costs $\ll n^3$. However, it is hard to determine the exact order of flops. It heavily depends on the sparsity structure of the matrix.