Data Sparse Matrix Computations - Lecture 3

Scribe: John Ryan

September 5, 2017

Contents

1 Application of Fast Fourier Transform

Discrete Convolution

The discrete convolution is a common technique in signal processing.

Suppose we have two signals, x_i and y_i for $i = 0, ..., N-1$ which are both periodic with respect to N. That is,

$$x_{i+jN} = x_i$$
$$y_{i+jN} = y_i \tag{1}$$

for any integer j. We'd like to compute the discrete convolution of x with y, which is defined as

$$g_k = (x * y)_k = \sum_{n=0}^{N-1} x_n y_{k-n} \tag{2}$$

for $k = 0, ..., N-1$.

(Scribe note: for some visual explanations on convolutions, the reader may visit https://en.wikipedia.org/wiki/Convolution and check out the "Visual Explanation" subsection)

Convolution as a Matrix/Vector multiplication

Notice that (2) can be written as

$$g = Yx$$

where g is a column vector with elements $g_k = (x * y)_k$, x is a column vector with elements x_k, and Y is an NxN matrix. By examining (2), we can deduce that the elements of the first row of the matrix Y should be

$$Y_{0,:} = \{y_0, y_{-1}, y_{-2}, ..., y_{-(N-1)}\}$$

Similarly, the second row should be

$$Y_{1,:} = \{y_1, y_0, y_{-1}, ..., y_{-(N-2)}\}$$

Now, we may take advantage of the periodicity of the signal y. Namely, we note that

$$y_{-1} = y_{N-1}$$

$$y_{-2} = y_{N-2}$$

and so on (see equation (1)). With this in mind, the rows of the matrix Y can now be written as

$$Y_{0,:} = \{y_0, y_{N-1}, y_{N-2}, ..., y_1\}$$

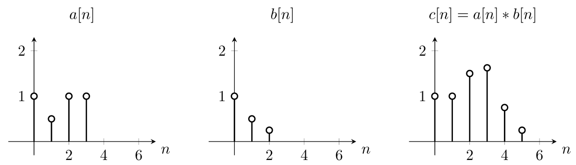

Figure 1: An example of a discrete convolution, taken from https://tex.stackexchange.com/questions/328627/compute-convolution-of-discrete-signals-in-tikz.

$$C = \begin{bmatrix} c_0 & c_{n-1} & \cdots & c_2 & c_1 \\ c_1 & c_0 & c_{n-1} & & c_2 \\ \vdots & c_1 & c_0 & \ddots & \vdots \\ c_{n-2} & & \ddots & \ddots & c_{n-1} \\ c_{n-1} & c_{n-2} & \cdots & c_1 & c_0 \end{bmatrix}.$$

Figure 2: An example of a circulant matrix.

$$Y_{1,:} = \{y_1, y_0, y_{N-1}, ..., y_2\}$$

and so on. Notice the pattern here - a given row is almost the same as the row above it in the matrix, except that the last element has been made the first, and every other element has been shifted to the right. Indeed, this is true for every row of the Y matrix, and as a result, the Y matrix (as well as any other matrix satisfying this property that each row is a shift of the one above/below it with an element on the end "wrapping around") is called a *circulant matrix.* See figure 1.

Well that's nice, but how does this help us? Recall that we're trying to compute the matrix vector product

$$g = Yx$$

It turns out that circulant matrices work well with the Fourier transform, in a way that will provide us a smart way to perform the above operation

Fourier Transform of a Convolution

Recall from last time that the Discrete Fourier Transform of a signal (vector) x is

$$y_k = \frac{1}{\sqrt{2\pi}} \sum_{j=0}^{N-1} x_j e^{-2\pi jk/N}$$

and can be written as a matrix/vector product:

$$y = Fx$$

Recall also that we have an algorithm, the Fast Fourier Transform, which allows us to compute the Fourier Transform in $O(n \log n)$, where n is the size of the original signal x. It turns out (and we will show) that, if Y is a circulant matrix, then FYF^* is diagonal.

Let us start by left multiplying both sides of $g = Yx$ by F.

$$Fg = FYx$$

$$\hat{g}_k = \sum_{l=0}^{N-1} \sum_{n=0}^{N-1} x_n y_{l-n} e^{-2\pi ilk/N}$$

Now we multiply the right side by $e^{-2\pi ik(n-n)/N}$ (which is equal to 1) and we get

$$\hat{g}_k = \sum_{l=0}^{N-1} \sum_{n=0}^{N-1} x_n y_{l-n} e^{-2\pi ilk/N} e^{-2\pi ik(n-n)/N}$$

$$= \sum_{n=0}^{N-1} x_n e^{-2\pi ikn/N} \sum_{l=0}^{N-1} y_{l-n} e^{-2\pi i(l-n)/N}$$

Now we notice that, due to the periodicity of y, the inner sum constitutes a Fourier transform of y regardless of what n is. Therefore, we have

$$\hat{g}_k = \sum_{n=0}^{N-1} x_n e^{-2\pi ikn/N} \hat{y}_k$$
$$= \hat{x}_k \hat{y}_k$$

Ah! In matrix/vector form, this says that

$$Fg = Diag(\hat{y})Fx$$

where $Diag(\hat{y})$ is a diagonal matrix whose elements are $D_{ii} = \hat{y}_i$. Then, left multiplying by F^* (which is the inverse of F), we get

$$g = F^* Diag(\hat{y})Fx \tag{3}$$

which, since $g = Yx$, implies that

$$F^* Diag(\hat{y})F = Y$$

or

$$Diag(\hat{y}) = FYF^*$$

which completes our proof that FYF^* is diagonal.

So now, instead of the naive matrix multiplication of Yx that would take $O(n^2)$ steps, we may perform the operations in equation (3).

1. First, FFT x to get $\hat{x}$ – $O(n\log(n)$ steps,
2. then FFT y to get $\hat{y}$ – $O(n\log(n))$ steps,
3. then multiply $\hat{x}$ by $Diag(\hat{y})$ – $O(n)$ steps,
4. then finally perform an inverse Fourier transform to get g, – $O(n\log(n))$ steps.

This takes $O(n\log(n))$ time.

There are two other important classes of matrices that we talk about today.

1. Toeplitz matrices, whose entries T_{ij} depend only on $i - j$. You can imagine that the "diagonals" are constant. Example: circulant matrices..
2. Hankel matrices, whose entries H_{ij} depend only on $i + j$. You can imagine that the "anti-diagonals" are constant. Example: if you look at Figure 1's reflection in a mirror.

Exercise: you can embed a Toeplitz or a Hankel matrix of size N into a circulant matrix of size $2N - 1$, and speed up matrix multiplication that way. How?

(Scribe's answer at end of document)

2 Project Topic Example

Suppose we have

$$F(\omega) = \sum_{n=0}^{N-1} x_N(t)e^{-it\omega}$$

So far, we have considered this for evenly spaced ω_k, like so

$$\omega_k = \frac{2\pi k}{N}$$

This is an equispaced transform

What about ω_k that aren't evenly spaced? Or even, instead of summing over $t = 0, 1, 2...$, what about summing over t_k for $k = 0, 1, 2, ...$ and the t_k aren't necessarily evenly spaced. How do we approach such transforms algorithmically?

This leads to a set of algos known as USFFTs (unequally spaced FFTs) or NUFFTs (non-uniform FFTs).

A potential plan for the final project could be to

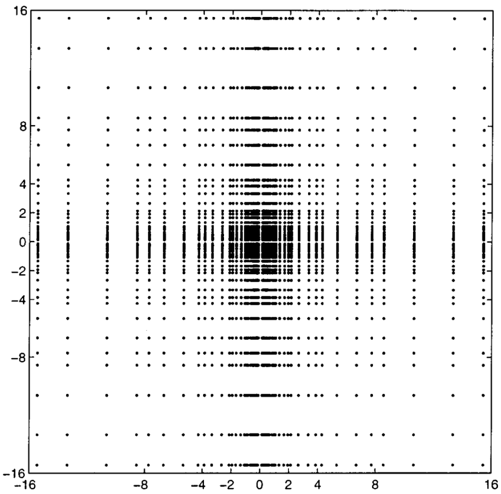

Figure 3: An example of an uneven frequency sampling. How do we efficiently and accurately transform to spatial coordinates?

- look at and describe/discuss these algorithms (similarities/differences),
- implement some of them, and look at how they scale (whether it's as expected or not), and explore their behavior and properties, and
- look into extensions into 2D with applications.

An example of an application might be the procedure by which many modern medical machines infer the density of a body part by firing signals at it from many different angles and seeing what goes through. The samples one gets from this procedure are not evenly spaced in frequency space, so something special needs to be done to convert the raw data into information on what's going on in spatial coordinates.

3 Beginning Fast Multipole Method

Now we're interested in computing

$$\Phi(x_i) = \sum_{j=1}^{N} K(x_i, x_j) q_j \tag{4}$$

for $i = 1, ..., N$

The q_i are weights, or charges.

For now, let's say $K(x_i, x_j) = \log(\|x_i - x_j\|)$ when $x_i \neq x_j$ and $K(x_i, x_j) = 0$ when $x_i = x_j$.

Much like in our previous discussions, we can consider (4) as a matrix/vector product.

$$\Phi = Kq$$

where K is a matrix defined by $K_{ij} = K(x_i, x_j)$.

Enter the Fast Multipole Method (FMM), which approximates the solution Φ in $O(N \log^{d-1}(\frac{1}{\epsilon}))$ time, where d is the dimension and ϵ is a measure of the precision. This runtime is accurate as $\epsilon \to 0$.

It does this by approximating $K(x, y)$ when it seems reasonable. For example, suppose you have a set of M "targets" x_i and a set of N "sources" y_i, and the sources are far from the targets in space, so that $K(x, y)$ is smooth. Suppose further that your kernel can be approximated like so

$$K(x, y) \approx \sum_{l=1}^{p} u_l(x) v_l(y)$$

which is to say, there is a low-rank approximation for the kernel. Then (4) may be written

$$\Phi(x_i) = \sum_{j=1}^{N} \sum_{l=1}^{p} u_l(x_i) v_l(y_j) q_j$$

(Scribe's note: I believe the switch from x to y is meant to emphasize that we are now discussing sets of sources and targets, rather than just some set of points).

Reordering the sums, we get

$$\Phi(x_i) = \sum_{l=1}^{p} u_l(x_i) \sum_{j=1}^{N} v_l(y_j) q_j$$

Notice that the inner sum has no dependence on i. Therefore, we may start by performing that sum for every l (which will take $O(MP)$ time), and then for each i we only need to compute the outer sum (which takes $O(P)$ time, and there are N different values of i, so in total it takes $O(NP)$). Therefore, we can find the entire Φ vector in $O(NP + MP)$ time.

Linear Algebra interpretation

If we just naively multiplied Kq to get Φ, that would take us $O(MN)$ time. However, with the FMM, we're essentially giving K a low-rank approximation:

$$K = uv^T$$

where u is an M by p matrix and v is an N by p matrix. Then $\Phi = Kq = uv^T q$ takes $O(MP+NP)$ steps.

Scribe's answer to exercise:

At first I tried typing it up, but it didn't go well. The following pages contain the answer for Toeplitz matrices - for Hankel matrices everything is the same except that C is formed in a slightly different way.

Note: there is a slightly better approach than what I wrote. Instead of doing an interlacing like I did, you can make the first row of C be the first row of A followed by the first $N-1$ entries of the last row of A. If you make a circulant matrix out of this, the upper left NxN block will be A, and in the final result, the vector we desire will be the first N entries of the output, rather than being every other entry as in the images below. Since it is easier to read/write sequential memory, this approach is better.

References

[1] Bracewell, R. (1986), *The Fourier Transform and Its Applications (2nd ed.),* McGraw–Hill, ISBN 0-07-116043-4.

[2] Carrier, J.; Greengard, L.; Rokhlin, V. *A fast adaptive multipole algorithm for particle simulations.* SIAM J. Sci. Statist. Comput. 9 (1988), no. 4, 669–686.

[3] Cooley, James W.; Tukey, John W. (1965). *An algorithm for the machine calculation of complex Fourier series.* Mathematics of Computation. 19 (90): 297–301.

[4] Damelin, S.; Miller, W. (2011), *The Mathematics of Signal Processing.* Cambridge University Press, ISBN 978-1107601048

[5] Davis, Philip J., *Circulant Matrices*, Wiley, New York, 1970

[6] Golub G.H., Van Loan C.F. (1996), *Matrix Computations* (Johns Hopkins University Press) §4.7—Toeplitz and Related Systems

[7] Golub, van Loan, §4.7.7 Circulant Systems

[8] Greengard, L.; Rokhlin, V. *A fast algorithm for particle simulations.* J. Comput. Phys. 73 (1987), no. 2, 325–348.

[9] Greengard, Leslie; Rokhlin, Vladimir *A new version of the fast multipole method for the Laplace equation in three dimensions.* Acta numerica, 1997, 229–269, Acta Numer., 6, Cambridge Univ. Press, Cambridge, 1997.

[10] Greengard L, Lin P (2000) *Spectral approximation of the free-space heat kernel.* Applied and Computational Harmonic Analysis 9:83–97.

[11] Nabors, K.; Korsmeyer, F. T.; Leighton, F. T.; White, J. *Preconditioned, adaptive, multipole-accelerated iterative methods for three-dimensional first-kind integral equations of potential theory.* Iterative methods in numerical linear algebra (Copper Mountain Resort, CO, 1992). SIAM J. Sci. Comput. 15 (1994), no. 3, 713–735.

[12] Rockmore, D.N. (January 2000). *The FFT: an algorithm the whole family can use.* Computing in Science Engineering. 2 (1): 60–64.

[13] Rokhlin, V. *Rapid solution of integral equations of scattering theory in two dimensions.* J. Comput. Phys. 86 (1990), no. 2, 414–439.

[14] Rokhlin, V. *Diagonal forms of translation operators for the Helmholtz equation in three dimensions.* Appl. Comput. Harmon. Anal. 1 (1993), no. 1, 82–93.

[15] Van Loan C.F., *Computational Frameworks for the Fast Fourier Transform* (SIAM, 1992).

$$A = \begin{bmatrix} a_0 & a_{-1} & a_{-2} & \cdots & a_{-(N-1)} \\ a_1 & a_0 & a_{-1} & & \cdots \\ a_2 & a_1 & a_0 & & \\ \vdots & & \ddots & & \\ a_{N-1} & & \vdots & & \end{bmatrix}$$

$\leftarrow$ Toeplitz

$N \times N$

$$C = \begin{bmatrix} a_0 & a_{N-1} & a_{-1} & a_{N-2} & a_{-2} & \cdots & a_1 & a_{-(N-1)} \\ a_{-(N-1)} & a_0 & a_{N-1} & a_{-1} & \cdots & & a_{-(N-2)} & a_1 \\ a_1 & a_{-(N-1)} & a_0 & a_{N-1} & \cdots & & & \\ & & & & & & & \end{bmatrix}$$

$\uparrow$ circulant

$2N-1 \times 2N-1$

Note

$$C_{2i,2j} = A_{i,j}$$

for i, j even

Suppose

$$Ax = y \qquad x = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{N-1} \end{bmatrix} \qquad y = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{N-1} \end{bmatrix}$$

Define $x_0 = \begin{bmatrix} x_0 \\ 0 \\ x_1 \\ 0 \\ x_2 \\ \vdots \\ x_{N-1} \end{bmatrix}$ size $2N-1 \times 1$

Then $Cx_0 = \begin{bmatrix} y_0 \\ z_0 \\ y_1 \\ z_1 \\ \vdots \\ y_{N-1} \end{bmatrix}$ size $2N-1 \times 1$

y's are same as above, z's are other unimportant (?) numbers

Input - A, x Output: $y = Ax$

Procedure - $O(n \log n)$

1) Perform FFT of x_0 to get $\hat{x}_0$

$O((2n-1)\log(2n-1))$
$= O(n \log n)$

2) Perform FFT of first row of C to get $\hat{c}$ $O(n \log n)$

DO NOT FORM C!

3) multiply $\mathrm{diag}(\hat{c})$ by $\hat{x}_0$ $O(n)$
to get $\hat{y}_0$

4) Inverse FFT $\hat{y}_0$ to get y_0 $O(n \log n)$

5) extract y from every other entry in y_0, Starting with first. $O(n)$