Computational Linear Algebra

The Matrix Multiplication

Julien Brenneck

April 2018

Introduction

This is a (very) brief introduction to numerical linear algebra, aimed towards readers with a familiarity of linear algebra. Some computational experience may also help. A more thorough introduction can be found in books by Trefethen [1] or Watkins [2], and a good reference of the field can be found in *Matrix Computations* by Golub [3]. A good refresher for the basic concepts and intuitions of linear algebra, presented in a beautifully geometric manner, can be found in the video series *Essence of linear algebra* by 3Blue1Brown on YouTube. A more traditional resource is the book *Introduction to Linear Algebra* by Strang.

3Blue1Brown,
by Grant Sanderson
3blue1brown.com.

A great number of problems in engineering, physical simulation, statistical learning, and many other fields, are fundamentally posed as matrix computations. Viewed in this way, the numerical algorithms used in matrix computation underpin almost all of scientific computing. An argument for why numerical linear algebra (and linear algebra in general) is so prevalent in scientific computation can be made by comparing it to the prevalence of calculus. In some sense, we often can only directly solve linear problems. With non linear problems, as we do in calculus, we often approximate it as a locally linear problem. A linear problem is exactly what numerical linear algebra has been built to solve.

With the rise of increasingly powerful general purpose computers in the last century, countless phenomena have been modeled as linear problems that required previously impossible numerical calculation. We see the importance of linear algebra in machine learning, where neural networks can be viewed as extensions of traditional linear models, adding a special type of differentiable non linearity, and using matrices as the core computational building block [4]. Another particularly important use has been in material science, where first principles simulation make use of linear solvers to compute estimates of material properties from quantum structure.

1 Definitions and Notation

Familiarity of the main definitions in linear algebra is expected, but for reference and to explain notation we will go over some important ones here.

Matrix: An $n \times m$ matrix $A \in \mathbb{R}^{n \times m}$ with n rows and m columns, and an $m \times 1$ column vector $x \in \mathbb{R}^m$ are written as follows:

Definition 1

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & \dots & a_{2m} \\ \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nm} \end{bmatrix}, \qquad x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix}$$

Matrix Vector Multiplication: When taking the product $b = Ax$ of a matrix $A \in \mathbb{R}^{n \times m}$ with a vector $x \in \mathbb{R}^m$ we define the ith element of the resulting vector $b \in \mathbb{R}^n$ as the inner (dot) product of the ith row of A with x.

Definition 2

$$b_i = \sum_{j=1}^{m} a_{ij} x_j$$

$$\begin{bmatrix} 1 & 2 \\ 0 & 3 \end{bmatrix} \begin{bmatrix} 3 \\ 2 \end{bmatrix} = \begin{bmatrix} 7 \\ 6 \end{bmatrix}$$

This represents a *linear mapping* f from $\mathbb{R}^m \to \mathbb{R}^n$, that is $f(x) = Ax$, or equivalently called a linear transformation and denoted $x \to Ax$. Any finite dimensional linear transformation can be represented by a matrix, and vice versa, so we think of them as equivalent. Here linear means for any $x, y \in \mathbb{R}^m$ and scalars $a, b \in \mathbb{R}$, the following holds

$$f(ax + by) = af(x) + bf(y).$$

A useful extension is an *affine* transformation, which has the form

$$x \to Ax + b$$

Matrix Multiplication: When taking the product $C = AB$ of a matrix $A \in \mathbb{R}^{n \times m}$ with a matrix $B \in \mathbb{R}^{m \times p}$ we define the (i, j) entry of the resulting matrix $C \in \mathrm{R}^{n \times p}$ as the inner (dot) product of the ith row of A with the jth column of B.

Definition 3

$$c_{ij} = \sum_{k=1}^{m} a_{ik} b_{kj}$$

$$\begin{bmatrix} 1 & 2 \\ 0 & 3 \end{bmatrix} \begin{bmatrix} 3 & 1 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 7 & 3 \\ 6 & 3 \end{bmatrix}$$

Where matrix vector multiplication is a linear transformation, matrix multiplication is a composition of linear transformations. If we have two linear transformations (matrices) $A \in \mathbb{R}^{n \times m}$ and $B \in \mathbb{R}^{m \times p}$, we can write them as $f(x) = Ax$, $g(y) = By$ and their composition as $f(g(y)) = ABy$.

Matrix multiplication is associative, so we can leave off any parenthesis denoting the order of operations. We see that BA is a new matrix (and thus a new linear transformation) of dimension $n \times p$. This works because A maps $\mathbb{R}^n$ to $\mathbb{R}^m$ and B maps $\mathbb{R}^m$ to $\mathbb{R}^p$, so the range of A is the domain of B and the functions can be composed.

It is important to note that matrix multiplication is not commutative in general, that is $AB \neq BA$, except in special cases such as $AI = IA$. Recall that I denotes the identity matrix, specifically the multiplicative identity of $n \times n$ matrices, whose diagonal elements are one and non-diagonal elements are zero.

$$\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$$

2 Computing the Matrix Multiplication

Of great practical importance is the speed at which a computer can perform matrix multiplication. The rise of modern convolutional neural networks is due largely to huge advancements of GPU technology that brought the capability for fast parallel matrix multiplication of small to medium sized matrices at an unprecedented scale [4]. This capability of fast matrix multiplication was needed for the advancement of real-time computer graphics starting in the 1980s, with GPUs only finding use in computational fields more recently.

2.1 Naive Matrix Multiplication

We are mainly concerned with square matrices ($n \times n$) in this paper, which simplifies the computational analysis of matrix multiplication. The naive algorithm essentially follows directly from the definition, each element of the new matrix is a sum of n products. The terminology of the *naive* algorithm is a convention of computer science, where a naive algorithm is obvious or follows directly from definitions [5]. To analyze a numerical computation we need to first understand how a computer understands calculations at the fundamental level. A single calculation of a computer is called an *operation*, it is a command that can be executed in a single computational step, often called a cycle. The base operations we are interested in are scalar multiplication and addition, both of which can be completed in approximately one cycle. The reality of physical hardware is much more complex, but this is a practical model for this type of analysis. In this context, we see that multiplying AB for $A, B \in \mathrm{R}^{n \times n}$ takes a total of $2n^3$ operations. From the definition, we see computing c_{ij} is a summation of n products,

The speed of a processor is derived from the *cycles per second* (Hz) it can run at. Modern computers run around four billion cycles per second.

which takes n scalar multiplications and n additions to compute, a total of $2n$. Doing this n^2 times, once for each element, results in $2n^3$ operations. Formally one only considers the asymptotic behavior of this function, up to a constant factor; here we would say matrix multiplication takes order n^3 operations, with notation $O(n^3)$. Similarly one can show that matrix vector multiplication takes $O(n^2)$ operations. An implementation of naive matrix multiplication in the Julia language is given in figure 1.

A pedantic note: there are n additions and not $n-1$ due to an initial addition with 0 that occurs when placing the output of our computation into newly initialized memory.

If you look at matrix multiplication holistically, you might notice that there is some amount of repeated computation. By recursively subdividing the matrices into blocks and using some algebra to exploit this repetition, we arrive at the Strassen Algorithm, which improves the number of operations from $O(n^3)$ to $O(n^{2.807})$. While the optimal number of operations is not known, there is a trivial lower bound of $O(n^2)$ from the number of elements in the resulting matrix. The best known algorithm, at least theoretically, is an extension of the Coppersmith–Winograd algorithm, with $O(n^{2.3728642})$ operations, though this is not useful in application due to large constant factors [5].

In practice optimizing for cache coherence is the critical aspect of writing an implementation, often using carefully written assembly routines hand optimized for particular machine instruction sets. In general these carefully written implementations are usually of the naive algorithm, often using a block sub-matrix approach, though for matrices larger than 500×500 there can be an improvement gain from partial application of the Strassen algorithm [6].

```
function matmul(A::Matrix, B::Matrix)::Matrix
    C = zero(A)
    for i=1:n, j=1:n, k=1:n
        C[i,j] += A[i,k]*B[k,j]
    end
    return C
end
```

FIGURE 1: Naive Matrix Multiplication in Julia

3 The Strassen Algorithm

We want to compute the matrix C given matrices A, B such that $C = AB$. For the Strassen algorithm to work, we require that they all have dimension $2^n \times 2^n$, so that they can be subdivided n times. This is a recursive algorithm that relies on subdividing the matrices as follows:

$$A = \left[\begin{array}{c|c} A_{00} & A_{01} \\ \hline A_{10} & A_{11} \end{array}\right], \quad B = \left[\begin{array}{c|c} B_{00} & B_{01} \\ \hline B_{10} & B_{11} \end{array}\right], \quad C = \left[\begin{array}{c|c} C_{00} & C_{01} \\ \hline C_{10} & C_{11} \end{array}\right].$$

Note that each A_{ij}, B_{ij}, C_{ij} is a matrix of dimension $2^{n-1} \times 2^{n-1}$

3.1 Recursive Matrix Multiplication

We first examine the naive matrix multiplication rewritten in a recursive form to build some intuition. This method is *recursive* because we again use the algorithm for the multiplication of $2^{n-1} \times 2^{n-1}$ sized matrices, which then relies on multiplication of $2^{n-2} \times 2^{n-2}$ sized matrices, and so on, recursively computing the final result. The base case reduces simply to multiplication of real numbers.

$$
\begin{aligned}
C_{00} &= A_{00}B_{00} + A_{01}B_{10} \\
C_{01} &= A_{00}B_{01} + A_{01}B_{11} \\
C_{10} &= A_{10}B_{00} + A_{11}B_{10} \\
C_{11} &= A_{10}B_{01} + A_{11}B_{11}
\end{aligned}
$$

Note that in the 2×2 case, this is the same as regular matrix multiplication, with each element being a scalar.

Rewriting in this form, we see that matrix multiplication requires eight recursive matrix multiplications. The cost of the matrix additions is only $O(n^2)$ so it is asymptotically dominated by the $O(n^3)$ cost and goes away in the limit. This is identical to the original algorithm in terms of operations, the computational complexity is still $O(n^3)$. The computational complexity in this form can be found by analyzing the recurrence relation, giving $O(n^{\log_2 8})$ which is equal to $O(n^3)$. The analysis is outside the scope of this paper but the result is important. It gives the same result but in a more intuitive form, the 8 is due to the 8 recursive multiplications and the 2 is due to the reduction in size by half.

A reference for this analysis can be found in *Introduction to Algorithms* (CLRS) [5], in the chapter on divide-and-conquer recursive algorithms. Of particular interest is the Master theorem.

3.2 Strassen's Method

The Strassen algorithm uses some clever, though not intuitive, algebra to compute the result with only 7 recursive matrix multiplications. To do this we need to use some intermediate forms, M_1 through M_7 of size $2^{n-1} \times 2^{n-1}$. Using these intermediate forms we then compute the sub-matrix values of the resulting output, C_{00}, C_{01}, C_{10}, and C_{11}.

$$
\begin{aligned}
M_1 &= (A_{00} + A_{11})(B_{00} + B_{11}) \\
M_2 &= (A_{10} + A_{11})B_{00} \\
M_3 &= A_{00}(B_{01} - B_{11}) \\
M_4 &= A_{11}(B_{10} - B_{00}) \\
M_5 &= (A_{00} + A_{01})B_{11} \\
M_6 &= (A_{10} - A_{11})(B_{00} + B_{01}) \\
M_7 &= (A_{01} - A_{11})(B_{10} + B_{11})
\end{aligned}
$$

$$
\begin{aligned}
C_{00} &= M_1 + M_4 - M_5 + M_7 \\
C_{01} &= M_3 + M_5 \\
C_{10} &= M_2 + M_4 \\
C_{11} &= M_1 - M_2 + M_3 + M_6
\end{aligned}
$$

Using the same analysis of the recurrence relation we find that the computational complexity is $O(n^{\log_2 7})$ since we now need only 7 recursive multiplications.

One might reasonably ask where these forms come from, to see the underlying mathematical reasoning hiding behind it all. Unfortunately trying to find intuition here is immensely unsatisfying. Through tedious algebraic calculation the algorithm can be verified as correct and mathematically sound, but doing so does nothing to elucidate the mystery of the result. Somewhat humorously in the algorithms textbook *Introduction to Algorithms* [5] (referred to as CLRS) the authors remark "Strassen's method is not at all obvious. (This might be the biggest understatement in this book)". The main intuition behind the algorithm is that the computational repetition in matrix multiplication can be exploited to reduce the overall number of computational operations, albeit through serious algebraic trickery. Another shortcoming of this algorithm is the inability to use by hand, even the smallest non-trivial example (4×4 matrices) results in several pages of tedious calculation. It is truly an algorithm meant for machines, which perhaps explains why it was not discovered until after the advent of computers.

A proof of correctness and associated algebraic verifications can be found on page 79 of CLRS [5].

The main significance of this discovery, made by Strassen in 1969, is that the naive approach was not optimal, something that had been largely not considered in the hundreds of years since the formalization of linear algebra and matrix multiplication. The value of this is increased by the fact that many other fundamental operations in linear algebra, such as matrix inversion, computing the determinant, Gaussian elimination, LU factorization, among others, have their computational complexity intrinsically linked to that of matrix multiplication. Improving the computational cost of matrix multiplication gives you a theoretical improvement in a number of other important algorithms essentially for free.

$$\left[\begin{array}{c|c} 1 & 0 \\ \hline 0 & 1 \end{array}\right]\left[\begin{array}{c|c} 2 & 1 \\ \hline 1 & 3 \end{array}\right] = \left[\begin{array}{c|c} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ \hline M_2 + M_4 & M_1 - M_2 + M_3 + M_6 \end{array}\right]$$

A trivial (in name only) example. Even in 2×2 where there is no recursion, it is a tedious calculation to do by hand.

$$\begin{aligned} M_1 + M_4 - M_5 + M_7 &= (1+1)(2+3) + 1(1-2) - (1+0)3 + (0-1)(1+3) = 2 \\ M_3 + M_5 &= 1(1-3) + (1+0)3 = 1 \\ M_2 + M_4 &= (0+1)2 + 1(1-2) \\ M_1 - M_2 + M_3 + M_6 &= (1+1)(2+3) - (0+1)2 + 1(1-3) + (0-1)(2+1) = 3 \end{aligned}$$

$$\implies \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} 2 & 1 \\ 1 & 3 \end{bmatrix} = \begin{bmatrix} 2 & 1 \\ 1 & 3 \end{bmatrix}$$

4 Matrix Multiplication in Practice

In reality computers are much more complex than the simple models used when analyzing algorithms. Implementations of linear algebra routines must be numerically robust, meaning the computations should be resilient to the numerical error inherent to floating point and fixed point arithmetic. These are approximation of the real numbers used by fundamentally discrete machines. Real implementations must also optimize for memory access time, also known as cache locality.

A processor has a hierarchy of memory, each level exponentially slower but also exponentially larger than the last. The fastest and smallest memory is embedded deep in the processor, the L1 cache (working memory) [5]. It is in general too small to hold all the necessary data for a linear algebra routine, so a fraction of it is loaded (cached) in working memory, while the rest sits in slower memory until it is needed. When implementing matrix multiplication for example, special care must be taken to load all the necessary data into the working memory. If data is needed it is initially looked for in the working memory. If it resides elsewhere, we have what is called a *cache miss*, where the processor must wait for potentially thousands of cycles while the necessary data is found in slower memory. This is an eternity in processor time, and needs to be avoided for an efficient implementation of an algorithm.

A modern processor is a maze of complexity, and in practice it is best to use existing implementations that have already dealt with these issues. Libraries such as OpenBLAS and BLIS contain robust, efficient implementation of common routines, such as matrix multiplication. These libraries do not tend to use the Strassen algorithm, though it can be helpful for very large dense matrices [6]. Due to large constant factors, the Strassen algorithm performs much worse for matrices smaller than roughly 1000×1000, depending on the machine. It also has non-trivial memory locality issues and sometimes non-optimal numerical stability. Despite this, it can be useful to break up a very large matrix with one or two recursive applications of the Strassen algorithm, followed by traditional matrix routines. For example breaking up a 1000×1000 sized matrix multiplication into 500×500 sized problems that can be computed with naive multiplication, which is faster for this smaller size.

See `github.com/flame/blis`

Understanding the algorithms of numerical linear algebra and their implementations is necessary when dealing with the many numerical methods that rely on them. Characterizing the speed and numerical stability of these algorithms allows the practitioner to know what methods to use for a given problem, how they can fail, and how to fix them when they do. Learning these algorithms is the first step towards well designing your own, as well as solving new and interesting problems in numerical linear algebra and scientific computing.

Bibliography

[1] L.N. Trefethen and D. Bau. *Numerical Linear Algebra*. Society for Industrial and Applied Mathematics, 1997.

[2] David S. Watkins. *Fundamentals of Matrix Computations (3rd Ed.)*. Wiley, 2010.

[3] Gene H. Golub and Charles F. Van Loan. *Matrix Computations (3rd Ed.)*. Johns Hopkins University Press, Baltimore, MD, USA, 1996.

[4] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[6] Jianyu Huang, Tyler M. Smith, Greg M. Henry, and Robert A. van de Geijn. Strassen's algorithm reloaded. *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 690–701, 2016.