Chapter 12

Strassen Matrix Multiplication

Copyright Oliver Serang, 2018
University of Montana Department of Computer Science

Matrix multiplication is ubiquitous in computing, and large matrix multiplications are used in tasks such as meteorology, image analysis, and machine learning. The product of two \( n \times n \) matrices \( A \) and \( B \) is defined as \( C = A \cdot B \), where \( C_{i,j} = \sum_k A_{i,k} \cdot B_{k,j} \). Matrix multiplication can be implemented in a naive manner \( \in O(n^3) \) by using three nested for loops, one for \( i \), one for \( j \), and one for \( k \). This cubic runtime will be prohibitive for large problems, and will limit the applicability of matrix multiplication.

12.1 A recursive view of matrix multiplication

For this reason, a matrix multiplication algorithm \( \in o(n^3) \) would be a significant achievement. We will proceed in a manner reminiscent of Gauß and Karatsuba multiplication. For this reason, we will first construct a naive recursive form for matrix implementation. This can be done by sub-dividing the \( A \) and \( B \) matrices into \( 2 \times 2 \) sub-matrices, each of size \( \frac{n}{2} \times \frac{n}{2} \):

\[
\begin{bmatrix}
C_{1,1} & C_{1,2} \\
C_{2,1} & C_{2,2}
\end{bmatrix}
= \begin{bmatrix}
A_{1,1} & A_{1,2} \\
A_{2,1} & A_{2,2}
\end{bmatrix}
\cdot
\begin{bmatrix}
B_{1,1} & B_{1,2} \\
B_{2,1} & B_{2,2}
\end{bmatrix}.
\]
Here we can see

\[
C_{1,1} = A_{1,1} \cdot B_{1,1} + A_{1,2} \cdot B_{2,1} \\
C_{1,2} = A_{1,1} \cdot B_{1,2} + A_{1,2} \cdot B_{2,2} \\
C_{2,1} = A_{2,1} \cdot B_{1,1} + A_{2,2} \cdot B_{2,1} \\
C_{2,2} = A_{2,1} \cdot B_{1,2} + A_{2,2} \cdot B_{2,2}.
\]

An implementation is shown in Listing 12.1. The runtime of this divide and conquer will be given by the recurrence \( r(n) = 8r\left(\frac{n}{2}\right) + \Theta(n^2) \), where each \( n \times n \) matrix multiplication reduces to 8 multiplications of size \( \frac{n}{2} \times \frac{n}{2} \) and the \( \Theta(n^2) \) term is given from the element-wise sums of matrices. Using the master theorem, we observe that the number of leaves is \( \ell = n^{\log_2(8)} = n^3 \), which has a polynomial power significantly larger than the \( \Theta(n^2) \) term, and is thus a leaf-heavy case; therefore, the runtime will be \( r(n) \in \Theta(n^3) \), matching the naive non-recursive algorithm described at the start of this chapter.

Listing 12.1: Naive recursive matrix multiplication.

```python
import numpy

# defined for square matrices:
# runtime is r(n) = 8*r(n/2) + n^2 \in \Theta(n^3)
def multiply_matrices(A,B,LEAF_SIZE=512):
    rA,cA=A.shape
    rB,cB=B.shape
    n = rA
    assert(n==cA and n==rB and n==cB)
    if n <= LEAF_SIZE:
        return A*B
    a11=A[:n/2,:n/2]
    a12=A[:n/2,n/2:]
    a21=A[n/2:,:n/2]
    a22=A[n/2:,n/2:]
    b11=B[:n/2,:n/2]
    b12=B[:n/2,n/2:]
    b21=B[n/2:,:n/2]
    b22=B[n/2:,n/2:]
    result = numpy.matrix(numpy.zeros( (n,n) ))
    result[:n/2,:n/2] = a11*b11 + a12*b21
    result[:n/2,n/2:] = a11*b12 + a12*b22
    result[n/2:,:n/2] = a21*b11 + a22*b21
    result[n/2:,n/2:] = a21*b12 + a22*b22
    return result
```
12.2. FASTER MATRIX MULTIPLICATION

n = 1024
A = numpy.matrix(numpy.random.uniform(0.0, 1.0, n*n).reshape(n,n))
B = numpy.matrix(numpy.random.uniform(0.0, 1.0, n*n).reshape(n,n))

print 'Naive iterative (with numpy)'
exact = A*B
print exact
print

print 'Recursive'
fast = multiply_matrices(A,B)
print fast
print

print 'Absolute error', max( numpy.fabs(numpy.array(fast - exact).flatten()) )

12.2  Faster matrix multiplication

We will now begin summing sub-matrices before multiplying them (in a manner reminiscent of Karatsuba’s algorithm). We do not have a concrete strategy for how to proceed, only that we should imitate our strategy in Gauß’ method and Karatsuba’s method. Note that here there is one additional constraint: matrix multiplication is not commutative in the general case \((i.e., A \cdot B \neq B \cdot A)\) on all matrices.

We follow the lead left by Karatsuba: We start by summing rows and columns before multiplying. Specifically, we’ll sum rows of \(A\) and columns of \(B\).

\[
X_1 = A_{1,1} + A_{1,2} \\
X_2 = A_{2,1} + A_{2,2} \\
Y_1 = B_{1,1} + B_{2,1} \\
Y_2 = B_{1,2} + B_{2,2}.
\]

We can compute some products of these row and column sums as we did in Karatsuba’s method. Specifically, \(C_{1,2}\) contains terms composed of \(X_1\) and
Y_2:

\[ P_1 = X_1 \cdot B_{2,2} \]
\[ P_2 = A_{1,1} \cdot Y_2. \]

The sum

\[ P_1 + P_2 = A_{1,1} \cdot B_{2,2} + A_{1,2} \cdot B_{2,2} + A_{1,1} \cdot B_{1,2} + A_{1,1} \cdot B_{2,2} \]
\[ = A_{1,1} \cdot B_{1,2} + A_{1,2} \cdot B_{2,2} + 2A_{1,1} \cdot B_{2,2} \]
\[ = C_{1,2} + 2A_{1,1} \cdot B_{2,2}. \]

Notice that we have overcounted \( A_{1,1} \cdot B_{2,2} \) rather than canceling it. We proceed as we did with Gauß multiplication: if we change the signs on the column sums \( Y_1 \) and \( Y_2 \), we can eliminate the \( A_{1,1} \cdot B_{2,2} \) terms.

\[ X_1 = A_{1,1} + A_{1,2} \]
\[ Y_2 = B_{1,2} - B_{2,2} \]
\[ P_1 = X_1 \cdot B_{2,2} \]
\[ P_2 = A_{1,1} \cdot Y_2 \]
\[ P_1 + P_2 = A_{1,1} \cdot B_{2,2} + A_{1,2} \cdot B_{2,2} + A_{1,1} \cdot B_{1,2} - A_{1,1} \cdot B_{2,2} \]
\[ = A_{1,1} \cdot B_{1,2} + A_{1,2} \cdot B_{2,2} \]
\[ = C_{1,2} \]

In the same manner, we can recover \( C_{2,1} \):

\[ X_2 = A_{2,1} + A_{2,2} \]
\[ Y_1 = B_{2,1} - B_{1,1} \]
\[ P_3 = X_2 \cdot B_{1,1} \]
\[ P_4 = A_{2,2} \cdot Y_1 \]
\[ P_3 + P_4 = A_{2,1} \cdot B_{1,1} + A_{2,2} \cdot B_{1,1} + A_{2,2} \cdot B_{2,1} - A_{2,2} \cdot B_{1,1} \]
\[ = A_{2,1} \cdot B_{1,1} + A_{2,2} \cdot B_{2,1} \]
\[ = C_{2,1}. \]

We have already incurred 4 sub-matrix multiplications (remember that the naive method uses 8 sub-matrix multiplications total); therefore, we will now try to find \( C_{1,1} \) and \( C_{2,2} \) by using \( \leq 3 \) more sub-matrix multiplications.
12.2. FASTER MATRIX MULTIPLICATION

Remember that $C_{1,1} = A_{1,1} \cdot B_{1,1} + A_{1,2} \cdot B_{2,1}$. We have not yet combined $A_{1,1} \cdot B_{1,1}$ nor $A_{1,2} \cdot B_{2,1}$. Likewise, remember that $C_{2,2} = A_{2,1} \cdot B_{1,2} + A_{2,2} \cdot B_{2,2}$, and we have not yet combined $A_{2,1} \cdot B_{1,2}$ nor $A_{2,2} \cdot B_{2,2}$. To combine $A_{1,1} \cdot B_{1,1}$ and $A_{2,2} \cdot B_{2,2}$, in the same product, we will compute

$$P_5 = (A_{1,1} + A_{2,2}) \cdot (B_{1,1} + B_{2,2}).$$

$P_5$ does not yet compute $A_{1,2} \cdot B_{2,1}$ (needed by $C_{1,1}$) and does not yet compute $A_{2,1} \cdot B_{1,2}$ (needed by $C_{2,2}$). Also, $P_5$ includes terms $A_{1,1} \cdot B_{2,2} + A_{2,2} \cdot B_{1,1}$, which are unwanted by both $C_{1,1}$ and $C_{2,2}$.

Fortunately, our previously computed $P_2$ contributes a $-A_{1,1} \cdot B_{2,2}$ term and $P_4$ contributes a $-A_{2,2} \cdot B_{1,1}$ term:

$$P_5 + P_2 + P_4 = A_{1,1} \cdot B_{1,1} + A_{2,2} \cdot B_{2,2} + A_{1,1} \cdot B_{1,2} + A_{2,2} \cdot B_{2,1}.$$ 

$$C_{1,1} - (P_5 + P_2 + P_4) = A_{1,2} \cdot B_{2,1} - (A_{2,2} \cdot B_{2,2} + A_{1,1} \cdot B_{1,2} + A_{2,2} \cdot B_{2,1})$$

$$= (A_{1,2} - A_{2,2}) \cdot B_{2,1} - A_{2,2} \cdot B_{2,2} - A_{1,1} \cdot B_{1,2}$$

$$= (A_{1,2} - A_{2,2}) \cdot (B_{2,1} + B_{2,2}) - A_{1,1} \cdot B_{1,2}$$

$$C_{2,2} - (P_5 + P_2 + P_4) = A_{2,1} \cdot B_{1,2} - (A_{1,1} \cdot B_{1,1} + A_{1,1} \cdot B_{1,2} + A_{2,2} \cdot B_{2,1})$$

$$= (A_{2,1} - A_{1,1}) \cdot B_{1,2} - A_{1,1} \cdot B_{1,1} - A_{2,2} \cdot B_{2,1}$$

$$= (A_{2,1} - A_{1,1}) \cdot (B_{1,1} + B_{1,2}) - A_{2,1} \cdot B_{1,1} - A_{2,2} \cdot B_{2,1}$$

The last of these steps in each of the above equations is perhaps the most mysterious. We do this because we have a $-A_{2,2} \cdot B_{2,2}$ term in the $C_{1,1}$ equation and a $-A_{1,1} \cdot B_{1,1}$ term in the $C_{2,2}$ equation. Consider that the entire point of constructing $P_5$ as we did was to compute the products $A_{1,1} \cdot B_{1,1}$ and $A_{2,2} \cdot B_{2,2}$ (because they have not yet occurred in any $P_1$, $P_2$, $P_3$, or $P_4$); therefore, any step where we still need to compute those values would essentially be circular. For this reason, the last step in each of the equations above factors to exchange the $-A_{2,2} \cdot B_{2,2}$ and $-A_{1,1} \cdot B_{1,1}$ terms for $-A_{1,2} \cdot B_{2,2}$ and $-A_{2,1} \cdot B_{1,1}$, respectively.

The key to moving forward from there is seeing that the residual $A_{1,2} \cdot B_{2,2} + A_{1,1} \cdot B_{1,2}$ includes terms from both $P_2$ and $P_1$. $P_1$ came from the aggregate first row of $A$ combined with $B_{2,2}$ and $P_2$ came from $A_{1,1}$ with the aggregate second column of $B$: the term where $P_1$ and $P_2$ overlap is the precise term that $P_1$ and $P_2$ contain but the residual $A_{1,2} \cdot B_{2,2} + A_{1,1} \cdot B_{1,2}$
CHAPTER 12. STRASSEN

... does not. The same is true for the second residual \( A_{2,1} \cdot B_{1,1} + A_{2,2} \cdot B_{2,1} \); it matches terms from \( P_4 \) and \( P_3 \), and once again the term they share is the term we wish to delete.

For this reason, the residuals \( A_{1,2} \cdot B_{2,2} + A_{1,1} \cdot B_{1,2} \) and \( A_{2,1} \cdot B_{1,1} + A_{2,2} \cdot B_{2,1} \) can be reconstructed using products already computed. We can find this by searching our already computed products for parts of these residuals. From this, we can see the following:

\[
A_{1,2} \cdot B_{2,2} + A_{1,1} \cdot B_{1,2} = P_2 + P_1 = A_{1,1} \cdot (B_{1,2} - B_{2,2}) + (A_{1,1} + A_{1,2}) \cdot B_{2,2}
\]

\[
A_{2,1} \cdot B_{1,1} + A_{2,2} \cdot B_{2,1} = P_4 + P_3 = A_{2,2} \cdot (B_{2,1} - B_{1,1}) + (A_{2,1} + A_{2,2}) \cdot B_{1,1}.
\]

As we aimed for, we have only used 7 products (with \( P_6 = (A_{1,2} - A_{2,2}) \cdot (B_{2,1} + B_{2,2}) \) and \( P_7 = (A_{2,1} - A_{1,1}) \cdot (B_{1,1} + B_{1,2}) \), which were computed above):

\[
C_{1,1} = P_5 + P_4 - P_1 + P_6
\]

\[
C_{2,2} = P_5 + P_2 - P_3 + P_7.
\]

Thus, we can reduce an \( n \times n \) matrix product to \( 7 \frac{n}{2} \times \frac{n}{2} \) matrix products and a constant number of matrix additions and subtractions. Thus we have \( r(n) = 7r(\frac{n}{2}) + \Theta(n^2) \). Using the master theorem, this is a leaf-heavy case with runtime \( \in \Theta(n^{\log_2(7)}) \approx \Theta(n^{2.807...}) \).

An implementation of Strassen’s algorithm is shown in Listing 12.2

Listing 12.2: Strassen’s fast matrix multiplication algorithm.

```python
import numpy

# defined for square matrices:
def strassen(A,B,LEAF_SIZE=512):
    rA,cA=A.shape
    rB,cB=B.shape
    n = rA
    assert(n==cA and n==rB and n==cB)
    if n <= LEAF_SIZE:
        return A*B
    a11=A[:n/2,:n/2]
    a12=A[:n/2,n/2:]
    a21=A[n/2:,:n/2]
    a22=A[n/2:,n/2:]
```

...
12.3. **ZERO PADDING**

When $n$ is not a power of 2, we can simply round $n$ up to the next power of 2 and “pad” with zeros so that our matrix of interest is embedded into a larger square matrix whose width is a power of 2. We could then multiply
this larger matrix with Strassen’s algorithm. The runtime of this approach will be no more than $r(2n)$, which is still $\in O(n^\log_2(7))$.

In practice, it is generally practically faster to change to a naive algorithm when $n$ becomes small enough. That approach is more flexible, because including base cases such as $3 \times 3$ matrices means that the divide-and-conquer method can include a factor of 3 as well as powers of 2. This approach can be paired with zero padding.

### 12.4 The search for faster algorithms

To date, the fastest known matrix multiplication algorithms use a more complicated variant of Strassen’s approach. Those algorithms have exponents $\approx 2.373$, and it is not yet known if faster algorithms exist. Likewise, it is not yet known whether faster algorithms are possible.

These fancier, asymptotically faster algorithms are rarely used in practice, because only problems with a large $n$ will allow the faster asymptotic complexity to overcome the larger runtime constant of those fancier algorithms; however, the simpler Strassen algorithm and its variants are used in high-performance linear algebra libraries.

Matrix multiplication must be $\in \Omega(n^2)$, because we at least need to load the two matrix arguments and to eventually write the matrix result. If we ever discover matrix multiplication algorithms $\in O(n^{2+r})$ (note that this would include runtimes such as $n^2(\log(n))^{1000}$), that discovery will have important implications on other important problems. For that reason, people are actively searching for improvements.