Matrix-mult-perspectives

September 7, 2017

1 Perspectives on matrix multiplication

One of the basic operations in linear algebra is matrix multiplication $C = AB$, computing the product of an $m \times n$ matrix A with an $n \times p$ matrix B to produce an $m \times p$ matrix C.

Abstractly, the rules for matrix multiplication are determined once you define how to multiply matrices by vectors Ax, the central linear operation of 18.06, by requiring that multiplication be associative. That is, we require:

$$A(Bx) = (AB)x$$

for all matrices A and B and all vectors x. The expression $A(Bx)$ involves only matrix $\times$ vector (computing $y = Bx$ then Ay), and requiring this to equal $(AB)x$ actually uniquely defines the matrix–matrix product AB.

1.1 Perspective 1: rows $\times$ columns

Regardless of how you derive it, the end result is the familar definition that you take **dot products of rows of A with columns of B** to get the product C. For example:

$$\begin{pmatrix} -14 & 5 & 10 \\ -5 & -20 & 10 \\ -6 & 10 & 6 \end{pmatrix} = \begin{pmatrix} 2 & -1 & 5 \\ 3 & 4 & 4 \\ -4 & -2 & 0 \end{pmatrix} \begin{pmatrix} 1 & 0 & -2 \\ 1 & -5 & 1 \\ -3 & 0 & 3 \end{pmatrix}$$

where we have highlighted the entry $-5 = 3 \times 1 + 4 \times 1 + 4 \times -3$ (second row of A $\cdot$ first column of B).

This can be written out as the formula

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

in terms of the entries of the matrices, e.g. c_{ij} is the entry in row i, column j of C, assuming A has n columns and B has n rows.

Essentially all matrix multiplications in practice are done with a version of this formula — at least, with the same operations, but often the *order* in which you multiply/add individual numbers is re-arranged.

In this notebook, we will explore several ways to *think* about these operations by re-arranging their order.

1.2 Julia matrix multiplication and dot products

Of course, Julia (along with many other software packages) can perform the arithmetic for you:

```
In [1]: A = [ 2  -1  5
              3   4  4
             -4  -2  0]
        B = [ 1   0  -2
              1  -5   1
             -3   0   3]
        C = A * B
```

```
Out[1]: 3×3 Array{Int64,2}:
         -14    5  10
          -5  -20  10
          -6   10   6
```

If we want, we can compute the individual dot products in Julia too. For example, let's compute $c_{2,1} = -5$ (the 2nd row and first column of C, or `C[2,1]` in Julia) by taking the dot product of the second row of A with the first column of B.

To extract rows and columns of a matrix, Julia supports a syntax for "array slicing" pioneered by Matlab. The second row of A is `A[2,:]`, and the first column of `B` is `B[:,1]`:

```
In [2]: A[2,:]
```

```
Out[2]: 3-element Array{Int64,1}:
         3
         4
         4
```

```
In [3]: B[:,1]
```

```
Out[3]: 3-element Array{Int64,1}:
          1
          1
         -3
```

Now we can compute $c_{2,1}$ by their dot product via the `dot` function:

```
In [4]: dot(A[2,:], B[:,1])
```

```
Out[4]: -5
```

This matches $c_{2,1}$ from above, or `C[2,1]` in Julia:

```
In [5]: C[2,1]
```

```
Out[5]: -5
```

1.3 Perspective 2: matrix × columns

AB can be viewed as multiplying A on the *left* by each *column* of B.

For example, let's multiply A by the first column of B:

```
In [6]: A*B[:,1]
```

```
Out[6]: 3-element Array{Int64,1}:
         -14
          -5
          -6
```

This is the first column of C! If we do this to *all* the columns of B, we get C:

```
In [7]: [ A*B[:,1]  A*B[:,2]  A*B[:,3] ] == C
```

```
Out[7]: true
```

Equivalently, each column of B specifies a linear combination of *columns* of A to produce the columns of C. So, **if you want to rearrange the *columns* of a matrix, multiply it by another matrix on the *right***.

For example, let's do the transformation that *flips the sign of the first column of A* and *swaps the second and third columns*.

```
In [8]: A * [ -1  0  0
               0  0  1
               0  1  0  ]
```

```
Out[8]: 3×3 Array{Int64,2}:
         -2  5  -1
         -3  4   4
          4  0  -2
```

1.4 Perspective 3: rows × matrix

AB can be viewed as multiplying each *row* of A by the matrix B on the *right*. Multiplying a row vector by a matrix on the right produces another row vector.

For example, here is the first row of A:

```
In [9]: A[1,:]
```

```
Out[9]: 3-element Array{Int64,1}:
          2
         -1
          5
```

Whoops, slicing a matrix in Julia produces a 1d array, which is interpreted as a column vector, no matter how you slice it. We can't multiply a column vector by a matrix B on the *right* — that operation is not defined in linear algebra (the dimensions don't match up). Julia will give an error if we try it:

```
In [10]: A[1,:] * B
```

```
DimensionMismatch("matrix A has dimensions (3,1), matrix B has dimensions (3,3)")


 in _generic_matmatmul!(::Array{Int64,2}, ::Char, ::Char, ::Array{Int64,2}, ::Array{Int64,2}) at

 in generic_matmatmul!(::Array{Int64,2}, ::Char, ::Char, ::Array{Int64,2}, ::Array{Int64,2}) at

 in *(::Array{Int64,1}, ::Array{Int64,2}) at ./linalg/matmul.jl:86
```

To get a row vector we must transpose it. In linear algebra, the transpose of a vector x is usually denoted x^T. In Julia, the transpose is `x.'`.

If we omit the `.` and just write `x'` it is the complex-conjugate of the transpose, sometimes called the *adjoint*, often denoted x^H (in matrix textbooks), x^* (in pure math), or $x^\dagger$ (in physics). For real-valued vectors (no complex numbers), the conjugate transpose is the same as the transpose, and correspondingly we usually just do `x'` for real vectors.

```
In [11]: A[1,:]'
```

```
Out[11]: 1×3 Array{Int64,2}:
          2  -1  5
```

Now, let's multiply this by B, which should give the first *row* of C:

```
In [12]: A[1,:]' * B
```

Yup!

Note that if we multiply a row vector by a matrix on the *left*, it doesn't really make sense. Julia will give an error:

```
In [13]: B * A[1,:]'
```

```
DimensionMismatch("matrix A has dimensions (3,3), matrix B has dimensions (1,3)")



 in _generic_matmatmul!(::Array{Int64,2}, ::Char, ::Char, ::Array{Int64,2}, ::Array{Int64,2}) at

 in generic_matmatmul!(::Array{Int64,2}, ::Char, ::Char, ::Array{Int64,2}, ::Array{Int64,2}) at

 in A_mul_Bc(::Array{Int64,2}, ::Array{Int64,1}) at ./operators.jl:320
```

If we multiply B on the right by *all* the rows of A, we get C again:

```
In [14]: [ A[1,:]'*B
           A[2,:]'*B
           A[3,:]'*B ] == C

Out[14]: true
```

Equivalently, each row of A specifies a linear combination of *rows* of B to produce the rows of C. So, **if you want to rearrange the *rows* of a matrix, multiply it by another matrix on the *left*.**

For example, let's do the transformation that *adds two times the first row of B to the third row, and leaves the other rows untouched.* This is one of the steps of Gaussian elimination!

```
In [15]: [ 1 0 0
           -1 1 0
           3 0 1 ] * B

Out[15]: 3×3 Array{Int64,2}:
          1   0  -2
          0  -5   3
          0   0  -3
```

1.5 Perspective 4: columns × rows

The key to this perspective is to observe:

- elements in column i of A only multiply elements in row j of B
- a column times a row vector, sometimes denoted xy^T, is an outer product and produces a "rank-1" *matrix*

For example, here is column 1 of A times row 1 of B:

```
In [16]: A[:,1] * B[1,:]'

Out[16]: 3×3 Array{Int64,2}:
           2  0  -4
           3  0  -6
          -4  0   8
```

If we do this for all three rows and columns and add them up, we get C:

```
In [17]: A[:,1] * B[1,:]' + A[:,2] * B[2,:]' + A[:,3] * B[3,:]' == C
```

```
Out[17]: true
```

So, from this perspective, we could write:

$$AB = \sum_{k=1}^{3} (\text{column } k \text{ of } A)(\text{row } k \text{ of } B) = \sum_{k=1}^{3} A[:,k]\, B[k,:]^T$$

where in the last expression we have used Julia notation for slices.

Perspective 5: submatrix blocks $\times$ blocks

It turns out that all of the above are special cases of a more general rule, by which we can break up a matrix in to "submatrix" blocks and multiply the blocks. Rows, columns, etc. are just blocks of different shapes. In homework you will explore another version of this: dividing a 4×4 matrix into 2×2 blocks.

1.6 More Gaussian elimination

Let's look more closely at the process of Gaussian elimination in matrix form, using the matrix from lecture 1.

```
In [18]: A = [1 3  1
              1 1 -1
              3 11 6]
```

```
Out[18]: 3×3 Array{Int64,2}:
          1   3   1
          1   1  -1
          3  11   6
```

Gaussian elimination produces the matrix U, which we can compute in Julia as in lecture 1:

```
In [19]: # LU factorization (Gaussian elimination) of the matrix A,
         # passing the undocumented option Val{false} to prevent row re-ordering
         L, U = lu(A, Val{false})
         U # just show U
```

```
Out[19]: 3×3 Array{Float64,2}:
          1.0   3.0   1.0
          0.0  -2.0  -2.0
          0.0   0.0   1.0
```

Now, let's go through **Gaussian elimination in matrix form**, by **expressing the elimination steps as matrix multiplications.** In Gaussian elimination, we make linear combination of *rows* to cancel elements below the pivot, and we now know that this corresponds to multiplying on the *left* by some *elimination matrix* E.

The first step is to eliminate in the first column of A. The pivot is the 1 in the upper-left-hand corner. For this A, we need to:

1. Leave the first row alone.
2. Subtract the first row from the second row to get the new second row.
3. Subtract $3\times$ first frow from the third row to get the new third row.

This corresponds to multiplying A on the left by the matrix `E1`. As above (in the "row $\times$ matrix" picture), the three rows of `E1` correspond exactly to the three row operations listed above:

In [20]:
```
E1 = [ 1 0 0
      -1 1 0
      -3 0 1]
```

Out[20]:
```
3×3 Array{Int64,2}:
  1  0  0
 -1  1  0
 -3  0  1
```

In [21]:
```
E1*A
```

Out[21]:
```
3×3 Array{Int64,2}:
 1   3   1
 0  -2  -2
 0   2   3
```

As desired, this introduced zeros below the diagonal in the first column. Now, we need to eliminate the 2 below the diagonal in the *second* column of `E1*A`. Our new pivot is -2 (in the second row), and we just add the second row of `E1*A` with the third row to make the new third row.

This corresponds to multiplying on the left by the matrix `E2`, which leaves the first two rows alone and makes the new third row by adding the second and third rows:

In [22]:
```
E2 = [1 0 0
      0 1 0
      0 1 1]
```

Out[22]:
```
3×3 Array{Int64,2}:
 1  0  0
 0  1  0
 0  1  1
```

In [23]:
```
E2*E1*A
```

Out[23]:
```
3×3 Array{Int64,2}:
 1   3   1
 0  -2  -2
 0   0   1
```

As expected, this is upper triangular, and in fact the same as the `U` matrix returned by the Julia `lu` function above:

In [24]:
```
E2*E1*A == U
```

Out[24]:
```
true
```

Thus, we have arrived at the formula:

$$\underbrace{E_2 E_1}_{E} A = U$$

Notice that we multiplied A by the elimination matrices from *right to left* in the order of the steps: it is $E_2 E_1 A$, *not* $E_1 E_2 A$. Because matrix multiplication is generally not commutative, $E_2 E_1$ and $E_1 E_2$ give *different* matrices:

In [25]:
```
E2*E1
```

Out[25]:
```
3×3 Array{Int64,2}:
  1  0  0
 -1  1  0
 -4  1  1
```

In [26]: E1*E2

Out[26]: 3×3 Array{Int64,2}:
```
  1  0  0
 -1  1  0
 -3  1  1
```

Notice, furthermore, that the matrices E_1 and E_2 are both *lower-triangular matrices.* This is a consequence of the structure of Gaussian elimination (assuming no row re-ordering): we always add the pivot row to rows *below* it, never *above* it.

In homework, you will explore the fact that the *product* of lower-triangular matrices is always lower-triangular too. In consequence, the product $E = E_2 E_1$ is lower-triangular, and Gaussian elimination can be viewed as yielding $EA = U$ where E is lower triangular and U is upper triangular.

However, in practice, it turns out to be more useful to write this as $A = E^{-1}U$, where E^{-1} is the inverse of the matrix E. We will have more to say about matrix inverses later in 18.06, but for now we just need to know that it is the matrix that **reverses the steps** of Gaussian elimination, taking us back from U to A. Computing matrix inverses is laborious in general, but in this particular case it is easy. We just need to *reverse the steps one by one* starting with the *last* elimination step and working back to the *first* one.

Hence, we need to reverse (invert) E_2 *first* on U, and *then* reverse (invert) E_1: $A = E_1^{-1}E_2^{-1}U$. But reversing an individual elimination step like E_2 is easy: we just **flip the signs below the diagonal**, so that wherever we *added* the pivot row we *subtract* and vice-versa. That is:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix}^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 1 \end{pmatrix}$$

(The last elimination step was adding the second row to the third row, so we reverse it by *subtracting* the second row from the third row of U.)

Julia can compute matrix inverses for us with the `inv` function. (It doesn't know the trick of flipping the sign, which only works for very special matrices, but it can compute it the "hard way" so quickly (for such a small matrix) that it doesn't matter.) Of course that gives the same result:

In [27]: inv(E2)

Out[27]: 3×3 Array{Float64,2}:
```
 1.0   0.0  0.0
 0.0   1.0  0.0
 0.0  -1.0  1.0
```

Similarly for E_1:

In [28]: inv(E1)

Out[28]: 3×3 Array{Float64,2}:
```
 1.0  0.0  0.0
 1.0  1.0  0.0
 3.0  0.0  1.0
```

If we didn't make any mistakes, then $E_1^{-1}E_2^{-1}U$ should give A, and it does:

In [29]: inv(E1)*inv(E2)*U == A

Out[29]: true

We call *inverse* elimination matrix $L = E^{-1} = E_1^{-1}E_2^{-1}$ Since the inverses of each elimination matrix were lower-triangular (with flipped signs), their product L is also lower triangular:

```
In [30]: L = inv(E1)*inv(E2)

Out[30]: 3×3 Array{Float64,2}:
          1.0   0.0  0.0
          1.0   1.0  0.0
          3.0  -1.0  1.0
```

As mentioned above, this is the same as the inverse of $E = E_2 E_1$:

```
In [31]: inv(E2*E1)

Out[31]: 3×3 Array{Float64,2}:
          1.0   0.0  0.0
          1.0   1.0  0.0
          3.0  -1.0  1.0
```

The final result, therefore, is that Gaussian elimination (without row swaps) can be viewed as a *factorization* of the original matrix A

$$A = LU$$

into a **product of lower- and upper-triangular matrices**. (Furthermore, although we didn't comment on this above, L is always 1 along its diagonal.) This factorization is called the LU factorization of A. (It's why we used the `lu` function in Julia above.) When a computer performs Gaussian elimination, what it computes are the L and U factors.

What this accomplishes is to break a complicated matrix A into **much simpler pieces** L and U. It may not seem at first that L and U are *that* much simpler than A, but they are: lots of operations that are very difficult with A, like solving equations or computing the determinant, become *easy* once you known L and U.