MATRIX MULTIPLICATION AND INVERSION

MATH 196, SECTION 57 (VIPUL NAIK)

Corresponding material in the book: Sections 2.3 and 2.4.

Executive summary

Note: The summary does not include some material from the lecture notes that is not important for present purposes, or that was intended only for the sake of illustration.

(1) *Recall*: A $n \times m$ matrix A encodes a linear transformation $T : \mathbb{R}^m \to \mathbb{R}^n$ given by $T(\vec{x}) = A\vec{x}$.
(2) We can add together two $n \times m$ matrices entry-wise. Matrix addition corresponds to addition of the associated linear transformations.
(3) We can multiply a scalar with a $n \times m$ matrix. Scalar multiplication of matrices corresponds to scalar multiplication of linear transformations.
(4) If $A = (a_{ij})$ is a $m \times n$ matrix and $B = (b_{jk})$ is a $n \times p$ matrix, then AB is defined and is a $m \times p$ matrix. The $(ik)^{th}$ entry of AB is the sum $\sum_{j=1}^{n} a_{ij}b_{jk}$. Equivalently, it is the dot product of the i^{th} row of A and the k^{th} column of B.
(5) Matrix multiplication corresponds to composition of the associated linear transformations. Explicitly, with notation as above, $T_{AB} = T_A \circ T_B$. Note that $T_B : \mathbb{R}^p \to \mathbb{R}^n$, $T_A : \mathbb{R}^n \to \mathbb{R}^m$, and $T_{AB} : \mathbb{R}^p \to \mathbb{R}^m$.
(6) Matrix multiplication makes sense *only if* the number of columns of the matrix on the left equals the number of rows of the matrix on the right. This comports with its interpretation in terms of composing linear transformations.
(7) Matrix multiplication is associative. This follows from the interpretation of matrix multiplication in terms of composing linear transformations and the fact that function composition is associative. It can also be verified directly in terms of the algebraic definition of matrix multiplication.
(8) Some special cases of matrix multiplication: multiplying a row with a column (the inner product or dot product), multiplying a column with a row (the outer product or Hadamard product), and multiplying two $n \times n$ diagonal matrices.
(9) The $n \times n$ identity matrix is an identity (both left and right) for matrix multiplication wherever matrix multiplication makes sense.
(10) Suppose n and r are positive integers. For a $n \times n$ matrix A, we can define A^r as the matrix obtained by multiplying A with itself repeatedly, with A appearing a total of r times.
(11) For a $n \times n$ matrix A, we define A^{-1} as the unique matrix such that $AA^{-1} = I_n$. It also follows that $A^{-1}A = I_n$.
(12) For a $n \times n$ invertible matrix A, we can define A^r for all integers r (positive, zero, or negative). A^0 is the identity matrix. $A^{-r} = (A^{-1})^r = (A^r)^{-1}$.
(13) Suppose A and B are matrices. The question of whether $AB = BA$ (i.e., of whether A and B commute) makes sense only if A and B are both square matrices of the same size, i.e., they are both $n \times n$ matrices for some n. However, $n \times n$ matrices need not always commute. An example of a situation where matrices commute is when both matrices are powers of a given matrix. Also, diagonal matrices commute with each other, and scalar matrices commute with all matrices.
(14) Consider a system of simultaneous linear equations with m variables and n equations. Let A be the coefficient matrix. Then, A is a $n \times m$ matrix. If $\vec{y}$ is the output (i.e., the augmenting column) we can think of this as solving the vector equation $A\vec{x} = \vec{y}$ for $\vec{x}$. If $m = n$ and A is invertible, we can write this as $\vec{x} = A^{-1}\vec{y}$.

(15) There are a number of algebraic identities relating matrix multiplication, addtion, and inversion. These include distributivity (relating multiplication and addition) and the involutive nature or reversal law (namely, $(AB)^{-1} = B^{-1}A^{-1}$). See the "Algebraic rules governing matrix multiplication and inversion" section in the lecture notes for more information.

Computational techniques-related ...

(1) The arithmetic complexity of matrix addition for two $n \times m$ matrices is $\Theta(mn)$. More precisely, we need to do mn additions.
(2) Matrix addition can be completely parallelized, since all the entry computations are independent. With such parallelization, the arithmetic complexity becomes $\Theta(1)$.
(3) The arithmetic complexity for multiplying a generic $m \times n$ matrix and a generic $n \times p$ matrix (to output a $m \times p$ matrix) using naive matrix multiplication is $\Theta(mnp)$. Explicitly, the operation requires mnp multiplications and $m(n-1)p$ additions. More explicitly, computing each entry as a dot product requires n multiplications and $(n-1)$ additions, and there is a total of mp entries.
(4) Matrix multiplication can be massively but not completely parallelized. All the entries of the product matrix can be computed separately, already reducing the arithmetic complexity to $\Theta(n)$. However, we can parallelized further the computation of the dot product by parallelizing addition. This can bring the arithmetic complexity (in the sense of the depth of the computational tree) down to $\Theta(\log_2 n)$.
(5) We can compute powers of a matrix quickly by using repeated squaring. Using repeated squaring, computing A^r for a positive integer r requires $\Theta(\log_2 r)$ matrix multiplications. An explicit description of the minimum number of matrix multiplications needed relies on writing r in base 2 and counting the number of 1s that appear.
(6) To assess the invertibility and compute the inverse of a matrix, augment with the identity matrix, then row reduce the matrix to the identity matrix (note that if its rref is not the identity matrix, it is not invertible). Now, see what the augmented side has turned to. This takes time (in the arithmetic complexity sense) $\Theta(n^3)$ because that's the time taken by Gauss-Jordan elimination (about n^2 row operations and each row operation requires $O(n)$ arithmetic operations).
(7) We can think of pre-processing the row reduction for solving a system of simultaneous linear equations as being equivalent to computing the inverse matrix first.

Material that you can read in the lecture notes, but not covered in the summary.

(1) Real-world example(s) to illustrate matrix multiplication and its associativity (Sections 3.4 and 6.3).
(2) The idea of fast matrix multiplication (Section 4.2).
(3) One-sided invertibility (Section 8).
(4) Noncommuting matrix examples and finite state automata (Section 10).

1. The goal of what we are trying to do

Recall that a linear transformation $T : \mathbb{R}^m \to \mathbb{R}^n$ can be encoded using a $n \times m$ matrix. Note that the dimension of the output space equals the number of rows and the dimension of the input space equals the number of columns. If A is the matrix for T, then $T(\vec{x}) = A\vec{x}$ for any vector $\vec{x} \in \mathbb{R}^m$.

Recall also that A is uniquely determined by T, because we know that the j^{th} column of A is the image in $\mathbb{R}^n$ of the standard basis vector $\vec{e_j} \in \mathbb{R}^m$. In other words, knowing T as a function determines A as a matrix.

Linear transformations are *functions*. This means that we can perform a number of operations on these in the context of operations on functions. Specifically, we can do the following:

- Pointwise addition: For $T_1, T_2 : \mathbb{R}^m \to \mathbb{R}^n$, we can define $T_1 + T_2$ as the operation $\vec{v} \mapsto T_1(\vec{v}) + T_2(\vec{v})$. It is easy to verify from the definition that if T_1 and T_2 are both linear transformations, then $T_1 + T_2$ is also a linear transformation.
- Scalar multiplication by a scalar constant: For $T : \mathbb{R}^m \to \mathbb{R}^n$ and a scalar a, define $aT : \mathbb{R}^m \to \mathbb{R}^n$ as the operation $\vec{v} \mapsto aT(\vec{v})$. It is easy to verify from the definition that if T is a linear transformation and a is a real number, then aT is also a linear transformation.

- Composition: If the co-domain of one linear transformation is the domain of another, it makes sense to *compose* the linear transformations. As you have seen in homework problems, a composite of linear transformations is a linear transformation.
- For a bijective linear transformation, we can define the *inverse*, which (as you have seen in a homework) also turns out to be a linear transformation.

In each case, we would like a description of the matrix for the new linear transformation in terms of matrices for the original linear transformation(s). The description should be purely algebraic, and for specific choices of the matrices, should involve pure arithmetic. Once we have done this, we have converted a tricky conceptual idea involving huge, hard-to-visualize spaces into easily coded numbers with easily coded operations.

We will see that:

- Addition of linear transformations corresponds to an operation that we will call *matrix addition*.
- Scalar multiplication of linear transformations corresponds to *scalar multiplication* of matrices.
- Composition of linear transformations corresponds to *matrix multiplication*.
- Computing the inverse of a linear transformation corresponds to computing the *matrix inverse*.

Let's get to it now!

2. Matrix addition and scalar multiplication

2.1. **Formal definition of matrix addition.** For matrices A and B, we define the sum $A + B$ only if A and B have an equal number of rows and also have an equal number of columns. Suppose A and B are both $n \times m$ matrices. Then $A + B$ is also a $n \times m$ matrix, and it is defined as the matrix whose $(ij)^{th}$ entry is the sum of the $(ij)^{th}$ entries of A and B. Explicitly, if $A = (a_{ij})$ and $B = (b_{ij})$, with $C = A + B = (c_{ij})$, then:

$$c_{ij} = a_{ij} + b_{ij}$$

2.2. **Matrix addition captures pointwise addition of linear transformations.** Suppose $T_1, T_2 : \mathbb{R}^m \to \mathbb{R}^n$ are linear transformations. Then, $T_1 + T_2$ is also a linear transformation from $\mathbb{R}^m$ to $\mathbb{R}^n$, defined by *pointwise addition*, similar to how we define addition for functions:

$$(T_1 + T_2)(\vec{v}) = T_1(\vec{v}) + T_2(\vec{v})$$

Note that the pointwise definition of addition uses only the vector space structure of the target space.

The claim is that the matrix for the pointwise sum of two linear transformations is the sum of the corresponding matrices. This is fairly easy to see. Explicitly, it follows from the distributivity of matrix-vector multiplication.

2.3. **Multiplying a matrix by a scalar.** Suppose A is a $n \times m$ matrix and λ is a real number. We define λA as the matrix obtained by multiplying each entry of A by the scalar λ.

2.4. **Matrices form a vector space.** We can think of $n \times m$ matrices as mn-dimensional vectors, just written in a matrix format. The matrix addition is the same as vector addition, and scalar multiplication of matrices mimics scalar-vector multiplication. In other words, the structure we have so far does not really make use of the two-dimensional storage format of the matrix. The two-dimensional storage format was, however, crucial to thinking about matrices as encoding linear transformations: the columns corresponding to the input coordinates, and the rows corresponded to the output coordinates. We will see that our definition of matrix multiplication is different and not just entry-wise: it involves using the row-column structure.

2.5. **Arithmetic complexity of matrix addition and scalar multiplication.** To add two $n \times m$ matrices, we need to do nm additions, one for each entry of the sum. Thus, the arithmetic complexity of matrix addition is $\Theta(nm)$. Similarly, to multiply a scalar with a $n \times m$ matrix, we need to do nm multiplication, so the arithmetic complexity of scalar multiplication is $\Theta(nm)$.

Both these operations can be massively parallelized if all our parallel processors have access to the matrices. Explicitly, we can make each processor compute a different entry of the sum. Thus, the parallelized arithmetic complexity is $\Theta(1)$. This ignores communication complexity issues.

3. Matrix multiplication: preliminaries

3.1. **Formal definition.** Let A and B be matrices. Denote by a_{ij} the entry in the i^{th} row and j^{th} column of A. Denote by b_{jk} the entry in the j^{th} row and k^{th} column of B.

Suppose the number of columns in A equals the number of rows in B. In other words, suppose A is a $m \times n$ matrix and B is a $n \times p$ matrix. Then, AB is a $m \times p$ matrix, and if we denote it by C with entries c_{ik}, we have the formula:

$$c_{ik} = \sum_{j=1}^{n} a_{ij} b_{jk}$$

The previous cases of dot product (vector-vector multiplication) and matrix-vector multiplication can be viewed as special cases of matrix-matrix multiplication. Explicitly, matrix-matrix multiplication can be thought of in three ways:

- It is a bunch of matrix-vector multiplications: The k^{th} column of the product matrix is the matrix-vector product of the matrix A with the k^{th} column of B.
- It is a bunch of vector-matrix multiplications: The i^{th} row of the product matrix is the vector-matrix product of the i^{th} row of A with the entire matrix B.
- Each entry is a dot product: The $(ik)^{th}$ entry of the product matrix is the dot product of the i^{th} row of A with the k^{th} column of B.

3.2. **It captures composition of linear transformations.** Suppose A is a $m \times n$ matrix and B is a $n \times p$ matrix. Suppose T_A is the linear transformation whose matrix is A and T_B is the linear transformation whose matrix is B. Note that T_B is a transformation $\mathbb{R}^p \to \mathbb{R}^n$ and T_A is a transformation $\mathbb{R}^n \to \mathbb{R}^m$. Then, $T_A \circ T_B$ is the linear transformation $\mathbb{R}^p \to \mathbb{R}^m$ that corresponds to the matrix product AB. In other words, the matrix for the composite of two linear transformations is the product of the matrices.

Note also that the matrix product makes sense in precisely the same circumstance (in terms of number of columns equaling number of rows) as the circumstance where the composite of linear transformations makes sense (the dimension of the output space to the operation done first, namely the one on the right, equals the dimension of the input space to the operation done second, namely the one on the left).

Let us now understand a little more in depth *why* this happens. The idea is to think about where exactly the standard basis vectors for $\mathbb{R}^p$ go under the composite.

Suppose $\vec{e_k}$ is the k^{th} standard basis vector in $\mathbb{R}^p$. We know that, under the transformation T_B, $\vec{e_k}$ gets mapped to the k^{th} column of B. In particular, the j^{th} coordinate of the image $T_B(\vec{e_k})$ is the matrix entry b_{jk}. Explicitly:

$$T_B(\vec{e_k}) = \sum_{j=1}^{n} b_{jk} \vec{e_j}$$

We now want to apply T_A to both sides. We get:

$$T_A(T_B(\vec{e_k})) = T_A\left(\sum_{j=1}^{n} b_{jk} \vec{e_j}\right)$$

We know that T_A is linear, so this can be rewritten as:

$$T_A(T_B(\vec{e_k})) = \sum_{j=1}^{n} (b_{jk} T_A(\vec{e_j}))$$

$T_A(\vec{e_j})$ is just the j^{th} column of the matrix A, so we get:

$$T_A(T_B(\vec{e_k})) = \sum_{j=1}^{n} b_{jk} \left(\sum_{i=1}^{m} a_{ij} \vec{e_i}\right)$$

The i^{th} coordinate in this is thus:

$$\sum_{j=1}^{n} b_{jk}a_{ij}$$

Thus, we get that:

$$c_{ik} = \sum_{j=1}^{n} a_{ij}b_{jk}$$

3.3. **Chaining and multiple intermediate paths.** The intuitive way of thinking of matrix multiplication is as follows. The value c_{ik} roughly describes the strength of the total pathway from $\vec{e_k}$ in $\mathbb{R}^p$ to $\vec{e_i}$ in $\mathbb{R}^m$. This "total pathway" is a sum of pathways via intermediate routes, namely the basis vectors for $\mathbb{R}^n$. The strength of the pathway via an intermediate vector $\vec{e_j}$ is $a_{ij}b_{jk}$. The total strength is the sum of the individual strengths.

This is very similar (and closely related) to the relationship between the chain rule for differentiation for a composite of functions of one variable and the chain rule for differentiation for a composite of functions of multiple variables.

3.4. **A real-world example.** Suppose there is a bunch of m people, a bunch of n foodstuffs that each person could consume, and a bunch of p nutrients. Let $A = (a_{ij})$ be the "person-foodstuff" matrix. This is the matrix whose rows are indexed by people and columns are indexed by foodstuffs, and where the entry in a particular row and particular column is the amount of the column foodstuff that the row person consumes. A is a $m \times n$ matrix. Note that we can set units in advance, but the units do not need to be uniform across the matrix entries, i.e., we could choose different units for different foodstuffs. It does make sense to use the same units for a given foodstuff across multiple persons, for reasons that will become clear soon.

Let $B = b_{jk}$ be the "foodstuff-nutrient" matrix. This is the matrix whose rows are indexed by foodstuffs and columns are indexed by nutrients, and where the entry in a particular row and particular column is the amount of the column nutrient contained per unit of the row foodstuff. B is a $n \times p$ matrix. Note that we want the units used for measuring the foodstuff to be the same in the matrices A and B.

Now, let us say we want to construct a "person-nutrient" matrix $C = (c_{ik})$. The rows are indexed by persons. The columns are indexed by nutrients. The entry in the i^{th} row and k^{th} column represents the amount of the k^{th} nutrient consumed by the i^{th} person. C is a $m \times p$ matrix. We now proceed to explain why $C = AB$, and in the process, provide a concrete illustration that justifies our definition of matrix multiplication.

The $(ik)^{th}$ entry of C is the amount of the k^{th} nutrient consumed by the i^{th} person. Now, the i^{th} person might get his fix of the k^{th} nutrient from a combination of multiple foodstuffs. In a sense, each foodstuff gives the i^{th} person some amount of the k^{th} nutrient (though that amount may well be zero). What is the contribution of each foodstuff?

Consider the j^{th} foodstuff, with $1 \leq j \leq n$. The i^{th} person consumes a_{ij} units of the j^{th} foodstuff. Each unit of the j^{th} foodstuff contains b_{jk} units of the k^{th} nutrient. The total amount of the k^{th} nutrient consumed by the i^{th} person via the j^{th} foodstuff is the product $a_{ij}b_{jk}$.

We now need to sum this up over all the foodstuffs. We thus get:

$$c_{ik} = \sum_{j=1}^{n} a_{ij}b_{jk}$$

This agrees with our definition of matrix multiplication, vindicating its utility.

3.5. **Matrix multiplication requires dimensions to match.** Matrix multiplication *only* makes sense when the number of columns in the first matrix equals the number of rows in the second matrix. Otherwise, it *just doesn't make sense.*

For instance, if m, n, and p are all different, A is a $m \times n$ matrix, and B is a $n \times p$ matrix, then AB makes sense whereas BA does not make sense. On the other hand, if $m = p$, then both AB and BA make sense. However, whereas AB is a $m \times (m = p)$ square matrix, BA is a $n \times n$ square matrix.

3.6. **Multiplication by the identity matrix.** We denote by I_n or Id_n the $n \times n$ identity matrix. The following are true:

- $I_n A = A$ for any $n \times p$ matrix A.
- $AI_n = A$ for any $m \times n$ matrix A.
- $I_n A = AI_n = A$ for any $n \times n$ matrix A.

3.7. **Reinterpreting the inverse in terms of matrix multiplication.** We had earlier defined a notion of *inverse* of a bijective linear transformation. We can interpret inverse easily in terms of matrix multiplication. The inverse of a $n \times n$ matrix A, denoted A^{-1}, satisfies the condition that both AA^{-1} and $A^{-1}A$ are equal to the $n \times n$ identity matrix. The $n \times n$ identity matrix is written as I_n or Id_n.

4. Computation of matrix products

4.1. **Time and space complexity of naive matrix multiplication.** The naive matrix multiplication procedure is to compute each cell entry by computing all the products involved and then summing them up. If we are considering the product of a $m \times n$ matrix with a $n \times p$ matrix, the product is a $m \times p$ matrix, with each matrix entry a sum of n products. This means that each matrix entry, naively, requires n multiplications and $(n-1)$ additions. The total number of multiplications is thus mnp and the total number of additions is $mp(n-1)$.

In the special case that we are multiplying two $n \times n$ square matrices, we require n^3 multiplications and $n^2(n-1)$ additions. The arithmetic complexity of naive matrix multiplication for $n \times n$ square matrices is thus $\Theta(n^3)$.

4.2. **Fast matrix multiplication: the idea.** *I may not get time to cover this in class.*

One key thing to note is that to multiply matrices, the only things we finally care about are the sums of products. We do not care to know the individual products at all. If there is some way of calculating the sums of products without calculating the individual products, that might be better.

Consider a simpler but related example. The expression here is not the expression that appears in a matrix product, but it offers a simple illustration of an idea whose more complicated version appears in fast matrix multiplication algorithms.

Consider the following function of four real numbers a, b, c, and d:

$$(a, b, c, d) \mapsto ac + bd + ad + bc$$

The naive approach to computing this function is to calculate separately the four products ac, bd, ad, and bc, and then add them up. This requires 4 multiplications and 3 additions.

An alternative approach is to note that this is equivalent to:

$$(a, b, c, d) \mapsto (a + b)(c + d)$$

Calculating the function in this form is considerably easier. It requires two additions and one multiplication. When we use the new method of calculation, we end up not getting to know the values of the individual products ac, bd, ad, and bc. But we were not interested in these anyway. Our interest was in the sum, which can be computed through this alternate method.

Some algorithms for fast matrix multiplication rely on the underlying idea here. Unfortunately, the idea needs a lot of tweaking to effectively be used for fast matrix multiplication. If you are interested, look up the *Strassen algorithm* for fast matrix multiplication. The Strassen algorithm combines a divide-and-conquer strategy with an approach that does a 2×2 matrix using only 7 multiplications instead of the 8 multiplications used in the naive algorithm. The saving accumulates when combined with the divide-and-conquer strategy, and we end up with an arithmetic complexity of $\Theta(n^{\log_2 7})$. The number $\log_2 7$ is approximately 2.8, which is somewhat of a saving over the 3 seen in naive matrix multiplication.

4.3. **Parallelizability.** Naive matrix multiplication is massively parallelizable as long as it is easy for multiple processors to access specific entries from the matrix. Explicitly, each entry of the product matrix can be computed by a different processor, since the computations of the different entries are independent under naive matrix multiplication. To compute the product of a $m \times n$ matrix and a $n \times p$ matrix, whereby the output is a $m \times p$ matrix, we can calculate each cell entry of the output using a separate processor which computes a dot product. Each processor does n multiplications and $n - 1$ additions, so the arithmetic complexity is $2n - 1$ operations, or $\Theta(n)$.

However, if we have access to more processors, we can do even better. Let's take a short detour into how to parallelize addition.

4.4. **Parallelizing addition.** Suppose we want to add n numbers using a (replicable) black box that can add two numbers at a time. The naive approach uses $n - 1$ steps: add the first two numbers, then add the third number to that, then add the fourth number to that, and so on. Without access to parallel computing, this is the best we can do. A parallel algorithm can do better using a *divide and conquer* strategy.

The one-step strategy would be to divide the list into two roughly equal halves, have two separate processors sum them up, and then add up their respective totals. This makes sense in the real world. Note that the time seems to have halved, but there is an extra step of adding up the two halves.

The smart approach is to then recurse: divide each half again into two halves, and so on. Ultimately, we will be left with finding sums of pairs of elements, then adding up those pairs, and so on. Effectively, our computation tree is a binary tree where the leaves are the values to be added. The depth of this tree describes the time complexity of the algorithm, and it is now $\Theta(\log_2 n)$.

4.5. **Extreme parallel processing for matrix multiplication.** The above idea for parallelizing addition allows us to compute each entry of the product matrix in time $\Theta(\log_2 n)$ (in terms of the depth of the arithmetic computation tree). Since all the entries are being computed in parallel, the arithmetic complexity of this massively parallelized algorithm is $\Theta(\log_2 n)$. Note how this is massively different from ordinary naive matrix multiplication, which takes times $\Theta(n^3)$, and in fact is better than any possible non-parallel algorithm, since any such algorithm must take time $\Omega(n^2)$ just to fill in all the entries.

5. Some particular cases of matrix multiplication

5.1. **The inner product or dot product.** A matrix product AB where A is a single row matrix and B is a single column matrix, and where the number of columns of A is the number of rows of B, becomes a dot product. Explicitly, the matrix product of a $1 \times n$ matrix and a $n \times 1$ matrix is simply their dot product as vectors, written as a 1×1 matrix.

5.2. **The outer product or Hadamard product.** A matrix product AB where A is a single column matrix and B is a single row matrix gives a rectangular matrix with as many rows as in A and as many columns as in B. Explicitly, the matrix product of a $m \times 1$ matrix and a $1 \times n$ matrix is a $m \times n$ matrix. The entries of the product can be thought of as being constructed from a *multiplication table*. The entry in the i^{th} row and j^{th} column is the product of the i^{th} entry of A and the j^{th} entry of B.

This particular type of product is called the *outer product* or *Hadamard product*.

5.3. **Multiplication of diagonal matrices.** Suppose A and B are both diagonal $n \times n$ matrices. Then, the matrix product AB is also a diagonal matrix, and each diagonal entry is obtained as a product of the corresponding diagonal entry of A and of B. Explicitly, if $C = AB$, then:

$$c_{ii} = a_{ii}b_{ii}$$

and

$$c_{ij} = 0 \text{ for } i \neq j$$

In terms of composition of linear transformations, each diagonal matrix involves scaling the basis vectors by (possibly) different factors. Composing just means composing the operations separately on each basis vector. In a diagonal transformation, the basis vectors do not get mixed up with each other. So, we do not have to worry about adding up multiple pathways.

6. Products of more than two matrices

6.1. **Associativity of matrix multiplication.** Suppose A is a $m \times n$ matrix, B is a $n \times p$ matrix, and C is a $p \times q$ matrix. Then, AB is a $m \times p$ matrix and BC is a $n \times q$ matrix. Thus, both $(AB)C$ and $A(BC)$ make sense, and they are both $m \times q$ matrices. The associativity law for matrix multiplication states that in fact, they are equal as matrices.

Intuitively, this makes sense, because it is just adding up weights over a lot of different pathways. Explicitly, the $(il)^{th}$ entry of the product is:

$$\sum_{j,k} a_{ij} b_{jk} c_{kl}$$

6.2. **Explaining associativity in terms of interpretation as linear transformations.** With the setup as above, we have that:

- The $m \times n$ matrix A represents a linear transformation $T_A : \mathbb{R}^n \to \mathbb{R}^m$.
- The $n \times p$ matrix B represents a linear transformation $T_B : \mathbb{R}^p \to \mathbb{R}^n$.
- The $p \times q$ matix C represents a linear transformation $T_C : \mathbb{R}^q \to \mathbb{R}^p$.

Then, we can say that:

- The $m \times p$ matrix AB represents the linear transformation $T_{AB} = T_A \circ T_B : \mathbb{R}^p \to \mathbb{R}^m$.
- The $n \times q$ matrix BC represents the linear transformation $T_{BC} = T_B \circ T_C : \mathbb{R}^q \to \mathbb{R}^n$.

Finally, we obtain that:

- The $m \times q$ matrix $(AB)C$ represents the linear transformation $T_{(AB)C} = (T_A \circ T_B) \circ T_C : \mathbb{R}^q \to \mathbb{R}^m$.
- The $m \times q$ matrix $A(BC)$ represents the linear transformation $T_{A(BC)} = T_A \circ (T_B \circ T_C) : \mathbb{R}^q \to \mathbb{R}^m$.

We know that function composition is associative. Therefore, the linear transformations $(T_A \circ T_B) \circ T_C$ and $T_A \circ (T_B \circ T_C)$ are equal. Hence, their corresponding matrices $(AB)C$ and $A(BC)$ are also equal. This gives another proof of the associativity of matrix multiplication.

6.3. **Associativity of matrix multiplication in a real-world context.** Consider the following four types of entities:

- A *community* is a set of people living in a geographical area. Suppose there are m communities.
- A *diet* is a particular specification ofhow much of each foodstuff an individual should consume daily. For instance, there may be a "standard paleo diet" or a "South Beach diet" or an "Atkins diet." Suppose there are N diets.
- A *foodstuff* is a type of food. Suppose there are p foodstuffs.
- A *nutrient* is something that a person needs for survival or human flourishing in his or her diet. Suppose there are q nutrients.

We can now construct matrices:

- A $m \times n$ matrix A called the "community-diet matrix" whose rows are indexed by communities and columns by diets. The entry in the i^{th} row and j^{th} column specifies the fraction of people in the i^{th} community that follows the j^{th} diet.
- A $n \times p$ matrix B called the "diet-foodstuff matrix" whose rows are indexed by diets and columns by foodstuffs. The entry in the j^{th} row and k^{th} column specifies the amount of the k^{th} food that is to be consumed daily in the j^{th} diet.
- A $p \times q$ matrix C called the "foodstuff-nutrient matrix" whose rows are indexed by foodstuffs and columns by nutrients. The entry in the k^{th} row and l^{th} column specifies the amount of the l^{th} nutrient per unit of the k^{th} foodstuff.

The matrix products are as follows:

- The $m \times p$ matrix AB is the "community-foodstuff matrix" whose rows are indexed by communities and columns by foodstuffs. The entry in the i^{th} row and k^{th} column specifies the average amount of the k^{th} foodstuff consumed in the i^{th} community.
- The $n \times q$ matrix BC is the "diet-nutrient matrix" whose rows are indexed by diets and columns by nutrients. The entry in the j^{th} row and l^{th} column specifies the total amount of the l^{th} nutrient in the j^{th} diet.

- The $m \times q$ matrix $A(BC) = (ABC)C$ is the "community-nutrient matrix" whose rows are indexed by communities and columns by nutrients. The entry in the i^{th} row and l^{th} column specifies the total amount of the l^{th} nutrient in the i^{th} community.

6.4. **Powers of a matrix.** Note that in order to multiply a matrix by itself, its number of columns must equal its number of rows, i.e., it must be a square matrix. If A is a $n \times n$ square matrix, we can make sense of $A^2 = AA$. Since matrix multiplication is associative, we can uniquely interpret higher powers of A as well, so A^r is the product of A with itself r times. In particular, $A^3 = A^2A = AA^2$. We also define A^0 to be the identity matrix.

This will make sense after you read the section on inverses: If A is invertible, then we can also make sense of *negative* powers of A. We define A^{-n} (for n a natural number) as $(A^{-1})^n$, and it is also equal to $(A^n)^{-1}$.

6.5. **Computing powers of a matrix.** We will discuss later how to invert a matrix. For now, let us consider how to find the positive powers of a matrix.

The *naive* approach to computing A^r where A is a $n \times n$ matrix is to just keep multiplying by A as many times as necessary. Recall that each naive matrix multiplication takes $n^2(2n-1)$ arithmetic operations (n^3 multiplications and $n^2(n-1)$ additions). If we naively multiply, we need $(r-1)$ matrix multiplications, so the total number of operations is $n^2(2n-1)(r-1)$. In order terms, it is $\Theta(n^3 r)$.

There is a trick to speeding it up at both ends. First, we can speed up individual matrix multiplication using fast matrix multiplication. Second, we can use repeated squaring. For instance, to compute A^8, we can square A three times. In general, the number of matrix multiplications that we need to do if chosen smartly is $\Theta(\log_2 r)$.

Note that the minimum number of multiplications that we need to carry out through repeated squaring to compute A^r is *not* an increasing, or even a non-decreasing, function of r. When r is a power of 2, the process of using repeated squaring is particularly efficient: we just square $\log_2 r$ times. If r is *not* a power of 2, we need to perform some repeated squarings, but we *also* need to perform other operations that multiply things together. For instance, to compute A^5, we do $(A^2)^2A$.

The general strategy is to compute A^{2^l} where 2^l is the largest power of 2 that is less than or equal to r, and in the process store all the intermediate A^{2^k}, $0 \leq k \leq l$. Now, A^r is a product of some of these matrices. Just multiply them all. The "worst case" scenario is when r is one less than a power of 2. In this case, $r = 2^{l+1} - 1 = 1 + 2 + \cdots + 2^l$ so we need to perform a total of $2l$ multiplications (l repeated squarings and then l "piecing together" multiplications). For instance, for A^7, we first compute A^2 (one multiplication), then compute $(A^2)^2 = A^4$ (second multiplication), then multiply A^4A^2A (two more operations).

The explicit formula for the minimum number of matrix multiplications needed is as follows: write r in base 2 notation (i.e., binary notation). The number of multiplications needed is:

(Total number of digits) + (Number of digits that are equal to 1) - 2

The reason is as follows: we need to do as many repeated squarings as (total number of digits) - 1, and then do as many multiplications as (number of digits that are equal to 1) - 1.

The first few examples are in the table below:

r	r in base 2	total number of digits	number of 1s	number of matrix multiplications
1	1	1	1	0
2	10	2	1	1
3	11	2	2	2
4	100	3	1	2
5	101	3	2	3
6	110	3	2	3
7	111	3	3	4
8	1000	4	1	3
9	1001	4	2	4

7. Inverting a matrix: the meaning and the method

7.1. **How to invert a matrix: augment with the identity matrix.** The following is a procedure for inverting a $n \times n$ matrix A:

- Write down the matrix A, and augment it with the $n \times n$ identity matrix.
- Complete the conversion of the matrix to reduced row-echelon form, and perform the same operations on the augmenting matrix.
- If the reduced row-echelon form of the original matrix is *not* the identity matrix, the matrix is non-invertible.
- If the reduced row-echelon form of the matrix is the identity matrix, then the matrix on the augmenting side is precisely the inverse matrix we are looking for.

There are two ways of explaining why this works.

First, think of the matrix A as the matrix of a linear transformation T. The matrix A^{-1} has columns equal to the vectors $T^{-1}(\vec{e}_1)$, $T^{-1}(\vec{e}_2)$, ..., $T^{-1}(\vec{e}_n)$. To find $T^{-1}(\vec{e}_1)$ is tantamount to solving $A\vec{x} = \vec{e}_1$, which can be obtained by augmenting A with $\vec{e}_1$ and solving. Similarly, to find $T^{-1}(\vec{e}_2)$ is tantamount to solving $A\vec{x} = \vec{e}_2$, which can be obtained by augmeting A with $\vec{e}_2$ and solving. In order to do all these computations together, we need to augment A with all of the columns $\vec{e}_1$, $\vec{e}_2$, and so on. This means augmenting A with the identity matrix. After A is converted to rref, the respective columns on the augmenting side are $T^{-1}(\vec{e}_1)$, $T^{-1}(\vec{e}_2)$, ..., $T^{-1}(\vec{e}_n)$. That's exactly what we want of A^{-1}.

The second approach is more subtle and involves thinking of the elementary row operations as matrices that are being done and undone. This is a separate approach that we will consider later.

7.2. **Arithmetic complexity of matrix inversion.** The arithmetic complexity of matrix inversion is the same as the arithmetic complexity of Gauss-Jordan elimination (actually, we need to do about twice as many operations, since we have many more augmenting columns, but the order remains the same). Explicitly, for a $n \times n$ matrix:

- The worst-case arithmetic complexity (in terms of time) of inverting the matrix is $\Theta(n^3)$ and the space requirement is $\Theta(n^2)$.
- A parallelized version of the inversion algorithm can proceed in $\Theta(n)$ time with access to enough processors.

7.3. **Particular cases of matrix inversion.** The following is information on inverting matrices that have a particular format:

- The inverse of a $n \times n$ diagonal matrix with all entries nonzero is a $n \times n$ diagonal matrix where each diagonal entry is inverted in place. Note that if any diagonal entry is zero, the matrix does not have full rank and is therefore non-invertible.
- The inverse of a $n \times n$ scalar matrix with scalar value λ is a scalar matrix with scalar value $1/\lambda$.
- The inverse of a $n \times n$ upper-triangular matrix where all the diagonal entries are nonzero is also an upper-triangular matrix where all the diagonal entries are nonzero, and in fact the diagonal entries of the inverse matrix are obtained by inverting the original diagonal entries in place (we need to use the augmented identity method to compute the other entries of the inverse matrix). An analogous statement is true for lower-triangular matrices.
- *This will make sense after you've seen permutation matrices*: The inverse of a $n \times n$ permutation matrix is also a $n \times n$ permutation matrix corresponding to the inverted permutation.

7.4. **Using the matrix inverse to solve a linear system.** Suppose A is an invertible $n \times n$ matrix. Solving a linear system with coefficient matrix A and augmenting column (i.e., output vector) $\vec{y}$ means solving the following vector equation for $\vec{x}$:

$$A\vec{x} = \vec{y}$$

We earlier saw that this can be done using Gauss-Jordan elimination for the augmented matrix $[A \mid \vec{y}]$. We can now think of it more conceptually. Multiply both sides of this vector equation on the left by the matrix A^{-1}:

$$A^{-1}(A\vec{x}) = A^{-1}\vec{y}$$

Solving, we get:

$$\vec{x} = A^{-1}\vec{y}$$

Suppose now that A is known in advance, so that we can pre-compute A^{-1}. Then, in order to solve the linear system

$$A\vec{x} = \vec{y}$$

we simply need to execute the vector-matrix multiplication

$$A^{-1}\vec{y}$$

This is multiplication of a $n \times n$ matrix and a $n \times 1$ vector, so by the naive matrix multiplication algorithm, the time taken for this is $\Theta(n^2)$.

Did we already know this? Yes, by a different name. We had not earlier conceptualized A^{-1} as a transformation. Rather, if A was known in advance, we carried out Gauss-Jordan elimination on A, then "stored the steps" used, so that then on being told the output vector $\vec{y}$, we could apply the steps one after another to $\vec{y}$ and obtain back the input vector $\vec{x}$. Now, instead of applying the steps sequentially, we just multiply by another matrix, A^{-1}. The matrix A^{-1} stores the information somewhat differently.

One downside is that we can use A^{-1} *only* in the situation where the coefficient matrix is a square matrix and is invertible. On the other hand, Gauss-Jordan elimination as a process works in general. There are generalizations of the inverse to other cases that we will hint at shortly.

One upside of using A^{-1} is that matrix-vector multiplication can be massively parallelized down to $\Theta(\log_2 n)$ arithmetic complexity (that is the complexity of the computation tree). Storing the sequence of row operations that need to be applied does work, but it is harder to parallelize, because the *sequence* in which the operations are applied is extremely important.

8. One-sided invertibility

We will not cover this section in class.

Suppose A is a $m \times n$ matrix and B is a $n \times m$ matrix such that AB is the $m \times m$ identity matrix, but $m \neq n$. In that case, the following are true:

- We must have that $m < n$.
- A has full row rank (rank m), i.e., the corresponding linear transformation is surjective.
- B has full column rank (rank m), i.e., the corresponding linear transformation is injective.

Interestingly, the converse results are also true, namely:

- If a $m \times n$ matrix A has full row rank m, there exists a $n \times m$ matrix B such that AB is the $m \times m$ identity matrix.
- If a $n \times m$ matrix B has full column rank m, there exists a $m \times n$ matrix A such that AB is the $m \times m$ identity matrix.

Both of these can be thought in terms of solving systems of simultaneous linear equations. We will see the proofs of these statements later. (If you are interested, look up the *Moore-Penrose inverse*).

9. Algebraic rules governing matrix multiplication and inversion

9.1. **Distributivity of multiplication.** Matrix multiplication is both left and right distributive. Explicitly:

$$\begin{aligned} A(B+C) &= AB + AC \\ (A+B)C &= AC + BC \end{aligned}$$

The equality is conditional both ways: if the left side in any one equality of this sort makes sense, so does the right side, and they are equal. In particular, the first law makes sense if B and C have the same row

and column counts as each other, and the column count of A equals the row count of B and C. The second law makes sense if A and B have the same row count and the same column count and both column counts equal the row count of C.

9.2. **The interplay between multiplication and inversion.** The basic algebra relating matrix multiplication and inversion follows from the algebraic manipulation rules related to an abstract structure type called a *group*. In other words, all the algebraic rules we discuss here hold in arbitrary groups. Nonetheless, we will not think of it in that generality.

Matrix inversion is *involutive* with respect to multiplication in the following sense. These two laws hold:

- $(A^{-1})^{-1} = A$ for any invertible $n \times n$ matrix A.
- $(AB)^{-1} = B^{-1}A^{-1}$ for any two invertible $n \times n$ matrices A and B (note that A and B could be equal)
- $(A_1A_2 \ldots A_r)^{-1} = A_r^{-1}A_{r-1}^{-1} \ldots A_1^{-1}$ for any invertible $n \times n$ matrices A_1, A_2, ..., A_r.

9.3. **Commutativity and the lack thereof.** First off, there are situations where we have matrices A and B such that AB makes sense but BA does not make any sense. For instance, if A is a $m \times n$ matrix and B is a $n \times p$ matrix with $m \neq p$, this is exactly what happens. In particular, $AB = BA$, far from being true, makes no sense.

Second, there can be situations where both AB and BA are defined but their row counts and/or column counts don't match. For instance, if A is a $m \times n$ matrix and B is a $n \times m$ matrix, then AB is a $m \times m$ matrix and BA is a $n \times n$ matrix. In particular, $AB = BA$ again makes no sense.

The only situation where the question of whether $AB = BA$ can be legitimately asked is the case where A and B are both square matrices of the same size, say both are $n \times n$ matrices. In such a situation, $AB = BA$ may or may not be true. It depends on the choice of matrices. In the case that $AB = BA$, we say that the matrices A and B *commute*.

Recall that matrix multiplication corresponds to composing the associated linear transformations. $AB = BA$ corresponds to the statement that the order of *composition* for the corresponding linear transformations does not matter. We will have a lot more to say about the conditions under which two linear transformations commute, but briefly:

- Any matrix commutes with itself, its inverse (if the inverse exists) and with powers of itself (positive, and also negative if it is invertible).
- Any two $n \times n$ diagonal matrices commute with each other.
- A $n \times n$ scalar matrix commutes with every possible $n \times n$ matrix.
- The set of matrices that commute with any matrix A is closed under addition and multiplication.

10. Noncommuting matrix examples and finite state automata

The explanation given here is not adequate for understanding the material. So it is strongly recommended that you pay attention in class.

Suppose $f : \{0, 1, 2, \ldots, n\} \to \{0, 1, 2, \ldots, n\}$ is a function with $f(0) = 0$. Consider a linear transformation T_f given by $T_f(\vec{e}_i) = \vec{e}_{f(i)}$ if $f(i) \neq 0$ and $T(\vec{e}_i) = \vec{0}$ if $f(i) = 0$.. The matrix for T has at most one nonzero entry in each column, and if there is a nonzero entry, then that entry is 1.

If the range of f is $\{0, 1, 2, \ldots, n\}$ then the matrix for T is a permutation matrix: exactly one nonzero entry, with value 1, in each row and exactly one nonzero entry, with value 1, in each column. Permutation matrices are quite important for a number of purposes. Our interest right now, though, is in the other kinds of matrices, because they help us construct simple examples of noncommuting matrices.

Pictorially, these kinds of linear transformations can be described using what are called *finite state automata*. The finite state automaton is a directed graph with one edge out of every node. The nodes here correspond to the standard basis vectors and the zero vector. An edge from one node to another means that the latter node is the image of the former node under the linear transformation. There is a loop at the zero node.

Composing the linear transformations is equivalent to composing the corresponding functions. In symbols:

$$T_{f \circ g} = T_f \circ T_g$$

If we denote by M_f the matrix for T_f, then we obtain:

$$M_{f \circ g} = M_f M_g$$

What this means is that to compose two linear transformations of this sort, we can compose the corresponding functions by "following the arrows" for their finite state automaton diagrams.

To find a power of a linear transformation of this sort, we simply keep doing multiple steps along the automaton. By various finiteness considerations, the powers of the matrix will eventually start repeating. Further, once there is a repetition, the pattern after that will involve repetition. For instance, suppose $A^{13} = A^{17}$ is the first repetition. Then the sequence $A^{13}, A^{14}, A^{15}, A^{16}$, will be repeated *ad infinitum*: $A^{17} = A^{13}, A^{18} = A^{14}$, and so on.

Some special cases are worth mentioning:

- If $A = A^2$, we say that A is *idempotent*, and the corresponding linear transformation is a *projection*. Certain kinds of projections are termed *orthogonal projections*. We will explore the terminology later.
- If $A^r = 0$ for some r, we say that A is *nilpotent*.

The guarantee of a repetition is specific to linear transformations of this sort, and is not true for linear transformations in general.

Consider the case $n = 2$. We have the following examples. We are constructing the matrix in each example as follows. For the i^{th} column for the matrix, enter the vector that is $T_f(\vec{e}_i) = \vec{e}_{f(i)}$. For instance, if $f(1) = 2$, the first column is the vector $\vec{e}_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$.

$f(1)$ and $f(2)$ (in order)	Matrix A	Smallest $0 \leq k < l$ s.t. $A^k = A^l$	Type of matrix
0 and 0	$\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$	1 and 2	zero matrix
0 and 1	$\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$	2 and 3	nilpotent (square is zero)
0 and 2	$\begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$	1 and 2	idempotent (orthogonal projection)
1 and 0	$\begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$	1 and 2	idempotent (orthogonal projection)
1 and 1	$\begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix}$	1 and 2	idempotent (non-orthogonal projection)
1 and 2	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$	0 and 1	identity matrix
2 and 0	$\begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}$	2 and 3	nilpotent (square is zero)
2 and 1	$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$	0 and 2	permutation matrix (square is identity)
2 and 2	$\begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix}$	1 and 2	idempotent (non-orthogonal projection)

These automata can be used to construct examples of noncommuting pairs of matrices. For instance, suppose we want matrices A and B such that $AB = 0$ but $BA \neq 0$. We try to construct functions $f, g : \{0, 1, 2\} \to \{0, 1, 2\}$ with $f(0) = g(0) = 0$ such that $f \circ g$ maps everything to 0 but $g \circ f$ does not. A little thought reveals that we can take:

$$f(0) = 0, f(1) = 0, f(2) = 1$$

$$g(0) = 0, g(1) = 1, g(2) = 0$$

To write the matrices explicitly, we look at the above and write in the i^{th} column the vector $\vec{e}_{f(i)}$. We get:

$$A = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}, B = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$$

The matrix products are:

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

and:

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

We will discuss more about matrix algebra and examples and counterexamples later.