**Chapter 2**

**LINEAR ALGEBRA**

Linear algebra is a form of continuous rather than discrete mathematics

Detailed reference sheet to review key formulas, we recommend *The Matrix Cookbook* (Petersen and Pedersen, 2006).

Another resource focused exclusively on teaching linear algebra, such as Shilov (1977).

* 1. **Scalars, Vectors, Matrices and Tensors**
     1. **Scalars**
* A scalar is just a single number.
* We write scalars in italics. We usually give scalars lower-case variable names.
* Example: “Let *s ∈* R be the slope of the line,”
  + 1. **Vectors**
  + A vector is an array of numbers, arranged in order.
  + Lower case names written in bold typeface
  + The elements of the vector are identified by writing its name in italic typeface, with a subscript. The first element of x is x1, the second element is x2 and so on
  + If each element is in R, and the vector has n elements, then the vector lies in the set formed by taking the Cartesian product of R n times, denoted as Rn.
  + Sometimes we need to index a set of elements of a vector. In this case, we deﬁne a set containing the indices and write the set as a subscript. For example, to access *x*1, *x*3 and *x*6, we deﬁne the set *S* = *{*1*,* 3*,* 6*}* and write *xS* .We use the *−* sign to index the complement of a set. For example, *x−*1 is the vector containing all elements of *x* except for *x*1 and *x−S* is the vector containing all the elements of *x* except for *x*1, *x*3 and *x*6.
    1. **Matrices**
* A matrix is a 2-D array of numbers
* Represented by upper-case variable names with bold typeface
* matrix **A** has a height of m and a width of n, then we say that **A** ∈ Rm×n
* We can identify all the numbers with vertical coordinate i by writing a “:” for the horizontal coordinate. For example, Ai,: denotes ith rowof A. Likewise, A:,i is ith columnof A
  + 1. **Tensors**
* An array of numbers arranged on a regular grid with a variable number of axes is known as a tensor*.*
* We denote a tensor named “A” with this typeface: **A**.
* We identify the element of **A** at coordinates (i, j, k) by writing Ai,j,k
* We denote the transpose of a matrix Aas *AT*, and it is deﬁned such that

(*AT*)*i,j* = *Aj,i*

* Vectors can be thought of as matrices that contain only one column. The transpose of a vector is therefore a matrix with only one row.
* scalar is its own transpose: a = aT.
* We can add matrices to each other, as long as they have the same shape, just by adding their corresponding elements*: C =* *A* + *B* where *C i,j* = *Ai,j* + *B i,j*
* We can also add a scalar to a matrix or multiply a matrix by a scalar, just by performing that operation on each element of a matrix: D = a · B + c where *Di,j* = *a · Bi,j* + *c*.
* In the context of deep learning, we allow the addition of matrix and a vector, yielding another matrix: C = A + b, where Ci,j=Ai,j + bj. In other words, the vector b is added to each row of the matrix. This implicit copying of b to many locations is called **broadcasting**.
  1. **Multiplying Matrices and Vectors**
* The product operation is defined by
* For matrix product to be defined, A must have the same number of columns as B has rows.
* Note that the standard product of two matrices is not just a matrix containing the product of the individual elements. Such an operation exists and is called the element-wise product or Hadamard product, and is denoted as A ⨀B.
* matrix multiplication is distributive and associative but not commutative:

A(B + C) = AB +AC

A(BC) = (AB)C

AB ≠ BA

* However, the dot product between two vectors is commutative:

xTy = yTx

* The transpose of a matrix product has a simple form:

(AB)T = BTAT

* 1. **Identity and Inverse Matrices**
* Identity matrix is a square matrix in which all the elements of the principal diagonal are ones and all other elements are zeros.
* The matrix inverseof A is denoted as A−1, and it is defined as the matrix such that

A−1A = In

* However, A −1 is primarily useful as a theoretical tool, and should not actually be used in practice for most software applications. Because A−1 can be represented with only limited precision on a digital computer, algorithms that make use of the value of b can usually obtain more accurate estimates of x.
  1. **Linear Dependence and Span**
* For A−1 to exist,

**A**x = b

must have exactly one solution for every value of b.

* To analyze how many solutions the equation has, we can think of the columns of A as specifying different directions we can travel from the origin(the point specified by the vector of all zeros), and determine how many ways there are of reaching b. In this view, each element of x specifies how far we should travel in each of these directions, with xi specifying how far to move in the direction of column i:
* In general, this kind of operation is called a linear combination. Formally, a linear combination of some set of vectors {v(1) , . . . , v(n) } is given by multiplying each vector v(i) by a corresponding scalar coefficient and adding the results:
* The spanof a set of vectors is the set of all points obtainable by linear combination of the original vectors.
* Determining whether Ax = b has a solution thus amounts to testing whether b is in the span of the columns of A. This particular span is known as the columnspaceor the rangeof A.
* In order for the system Ax = b to have a solution for all values of b ∈ Rm, we therefore require that the column space of A be all of Rm. If any point in Rm is excluded from the column space, that point is a potential value of b that has no solution. The requirement that the column space of A be all of Rm implies immediately that A must have at least m columns, i.e., n ≥ m. Otherwise, the dimensionality of the column space would be less than m. For example, consider a 3 × 2 matrix. The target b is 3-D, but x is only 2-D, so modifying the value of x at best allows us to trace out a 2-D plane within R3. The equation has a solution if and only if b lies on that plane.
* Having n ≥ m is only a necessary condition for every point to have a solution. It is not a sufficient condition, because it is possible for some of the columns to be redundant. Consider a 2 ×2 matrix where both of the columns are identical.
* Formally, this kind of redundancy is known as **linear dependence**. A set of vectors is **linearly independent**if no vector in the set is a linear combination of the other vectors. If we add a vector to a set that is a linear combination of the other vectors in the set, the new vector does not add any points to the set’s span. This means that for the column space of the matrix to encompass all of Rm, the matrix must contain at least one set of m linearly independent columns. This condition is both necessary and sufficient for Ax = b to have a solution for every value of b. Note that the requirement is for a set to have exactly m linear independent columns, not at least m. No set of m-dimensional vectors can have more than m mutually linearly independent columns, but a matrix with more than m columns may have more than one such set.
* In order for the matrix to have an inverse, we additionally need to ensure that Ax=b has at mostone solution for each value of b. To do so, we need to ensure that the matrix has at most m columns. Otherwise there is more than one way of parametrizing each solution.
* Together, this means that the matrix must be *square*, that is, we require that m = n and that all of the columns must be linearly independent. A square matrix with linearly dependent columns is known as **singular**.
* If A is not square or is square but singular, it can still be possible to solve the equation. However, we cannot use the method of matrix inversion to find the solution.
* So far, we have discussed matrix inverses as being multiplied on the left. It is also possible to define an inverse that is multiplied on the right:

AA-1 = I

For square matrices, the left inverse and right inverse are equal.

* 1. **Norms**
* In machine learning, we usually measure the size of vectors using a function called a *norm*. Formally, the LP norm is given by
* More rigorously, a norm is any function f that satisfies the following properties:

1. f (x) = 0 ⇒ x = 0
2. f (x + y) ≤ f(x) + f(y) (the *triangle inequality*)
3. ∀α ∈ R, f(αx) = |α|f(x)

* The **L2 norm**, with p = 2, is known as the *Euclidean norm*. It is simply the Euclidean distance from the origin to the point identified by x, often denoted simply as ||x||.
* It is also common to measure the size of a vector using the squared L2 norm, which can be calculated simply as xTx.
* In many contexts, the squared L2 norm may be undesirable because it increases very slowly near the origin. In several machine learning applications, it is important to discriminate between elements that are exactly zero and elements that are small but nonzero. In these cases, we turn to a function that grows at the same rate in all locations, but retains mathematical simplicity: the **L1 norm**. The L1 norm may be simplified to
* We sometimes measure the size of the vector by counting its number of nonzero elements. Some authors refer to this function as the “**L0 norm**,” but this is incorrect terminology.
* One other norm that commonly arises in machine learning is the L∞ norm, also known as the max norm. This norm simplifies to the absolute value of the element with the largest magnitude in the vector
* Sometimes we may also wish to measure the size of a matrix. In the context of deep learning, the most common way to do this is with the otherwise obscure Frobenius norm which is analogous to the L2 norm of a vector
  1. **Special Kinds of Matrices and Vectors**
* **Diagonalmatrices** consist mostly of zeros and have non-zero entries only along the main diagonal.
* one example of a diagonal matrix is the identity matrix
* We write diag(v) to denote a square diagonal matrix whose diagonal entries are given by the entries of the vector v.
* Diagonal matrices are of interest in part because multiplying by a diagonal matrix is very computationally efficient.

diag( v)x = v ⨀ x

* Inverting a square diagonal matrix is also efficient.

diag(v)−1 = diag([1/v 1, . . . , 1/vn ]T)

* Not all diagonal matrices need be square. It is possible to construct a rectangular diagonal matrix. Non-square diagonal matrices do not have inverses but it is still possible to multiply by them cheaply.
* A **symmetric****matrix** is any matrix that is equal to its own transpose:

A = AT

* A **unit vector**is a vector with unit norm:
* A vector x and a vector y are orthogonal to each other if xTy = 0.
* If the vectors are not only orthogonal but also have unit norm, we call them **orthonormal**.
* An **orthogonal matrix**is a square matrix whose rows are mutually orthonormal and whose columns are mutually orthonormal:

ATA = AAT = I

i.e. A−1 = AT

* 1. **Eigendecomposition**
* To begin, let v be a Vector and A be a matrix. If we multiply A by v we get new vector Av. If you can draw a line through the three points (0,0), v and Av, then Av is just v multiplied by a number λ; that is,

Av=λv

In this case, we call v an **eigenvector** and λ an **eigenvalue** of corresponding eigenvector.

* If v is an eigenvector of A, then so is any rescaled vector sv for s ∈ R, s ≠ 0. Moreover, sv still has the same eigenvalue.
* Suppose that a matrix A has n linearly independent eigenvectors, {v(1) , . . . ,v(n)}, with corresponding eigenvalues {λ1, . . . , λn }. We may concatenate all the eigenvectors to form a matrix V with one eigenvector per column: V = [v(1) , . . . ,v(n)]. Likewise, we can concatenate the eigenvalues to form a vector λ = [λ1, . . . ,λn ]. The **eigendecomposition**of A is then given by

A = V diag(λ)V −1

* Not every matrix can be decomposed into eigenvalues and eigenvectors. In some cases, the decomposition exists, but may involve complex rather than real numbers.
* Specifically, every real symmetric matrix can be decomposed into an expression using only real-valued eigenvectors and eigenvalues:

A = QΛQT

where Q is an orthogonal matrix composed of eigenvectors of A, and Λ is a diagonal matrix. The eigenvalue Λi,i is associated with the eigenvector in column i of Q, denoted as Q:,i.

* The eigendecomposition of a matrix tells us many useful facts about the matrix.

1. The matrix is singular if and only if any of the eigenvalues are zero.
2. The eigendecomposition of a real symmetric matrix can also be used to optimize quadratic expressions of the form

f(x) = xTAx subject to ||x||2 = 1.

Whenever x is equal to an eigenvector of A, f takes on the value of the corresponding eigenvalue.

The maximum value of f within the constraint region is the maximum eigenvalue

and its minimum value within the constraint region is the minimum eigenvalue.

* A matrix whose eigenvalues are all positive is called **positive definite**. A matrix whose eigenvalues are all positive or zero-valued is called **positive semidefinite**. Likewise, if all eigenvalues are negative, the matrix is**negative definite**, and if all eigenvalues are negative or zero-valued, it is **negative semidefinite**.
* Positive semidefinite matrices are interesting because they guarantee that ∀x, xTAx ≥ 0.
* Positive definite matrices additionally guarantee that xTAx = 0 ⇒ x = 0.
  1. **Singular Value Decomposition**
* The singular value decomposition(SVD) provides another way to factorize a matrix, into **singular vectors**and singular values. The SVD allows us to discover some of the same kind of information as the eigendecomposition. However, the SVD is more generally applicable.
* Every real matrix has a singular value decomposition, but the same is not true of the eigenvalue decomposition. For example, if a matrix is not square, the eigendecomposition is not defined, and we must use a singular value decomposition instead.
* The singular value decomposition is similar, except this time we will write A as a product of three matrices:

A = UDV T

* The matrices U and V are both defined to be orthogonal matrices. The matrix D is defined to be a diagonal matrix. Note that D is not necessarily square.
* The elements along the diagonal of D are known as the **singular values**of the matrix A. The columns of U are known as the **left-singular****vectors**. The columns of V are known as the **right-singular vectors**.
* The left-singular vectors of A are the eigenvectors of AAT. The right-singular vectors of A are the eigenvectors of ATA.
* The non-zero singular values of A are the square roots of the eigenvalues of ATA. The same is true for AAT.
  1. **The Moore-Penrose Pseudoinverse**
* Matrix inversion is not defined for matrices that are not square. Suppose we want to make a left-inverse B of a matrix A, so that we can solve a linear equation

**A**x = y

* If A is taller than it is wide, then it is possible for this equation to have no solution. If A is wider than it is tall, then there could be multiple possible solutions.
* The Moore-Penrose pseudoinverseallows us to make some headway in these cases. The pseudoinverse of A is defined as a matrix
* Practical algorithms for computing the pseudoinverse are not based on this definition but rather the formula:
* where U, D and V are the singular value decomposition of A, and the pseudoinverse D+ of a diagonal matrix D is obtained by taking the reciprocal of its non-zero elements then taking the transpose of the resulting matrix.
* When A has more columns than rows, then solving a linear equation using the pseudoinverse provides one of the many possible solutions.
* When A has more rows than columns, it is possible for there to be no solution. In this case, using the pseudoinverse gives us the x for which Ax is as close as possible to y in terms of Euclidean norm

||Ax-y||2.

* 1. **The Trace Operator**
* The trace operator gives the sum of all the diagonal entries of a matrix:
* The trace operator is useful for a variety of reasons. Some operations that are difficult to specify without resorting to summation notation can be specified using matrix products and the trace operator. For example, the trace operator provides an alternative way of writing the Frobenius norm of a matrix:
* Writing an expression in terms of the trace operator opens opportunities to manipulate the expression using many useful identities. For example, the trace operator is invariant to the transpose operator:

Tr(A) = Tr(AT)

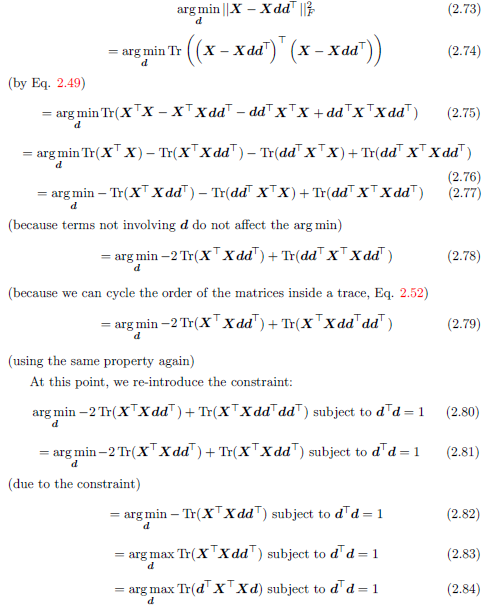
* The trace of a square matrix composed of many factors is also invariant to moving the last factor into the first position, if the shapes of the corresponding matrices allow the resulting product to be defined:

Tr(ABC) = Tr(CAB) = Tr(BCA)

* Another useful fact to keep in mind is that a scalar is its own trace: a = Tr(a ).
  1. **The Determinant**
* The determinant is equal to the product of all the eigenvalues of the matrix.
* The absolute value of the determinant can be thought of as a measure of how much multiplication by the matrix expands or contracts space.
* If the determinant is 0, then space is contracted completely along at least one dimension, causing it to lose all its volume. If the determinant is 1, then the transformation is volume-preserving.
  1. **Example: Principal Components Analysis**
* Suppose we have a collection of m points {x(1) , . . . , x(m)} in Rn . Suppose we would like to apply lossy compression to these points.
* For each point x(i) ∈ Rn we will find a corresponding code vector c(i) ∈ Rl. If l is smaller than n, it will take less memory to store the code points than the original data. We will want to find some encoding function that produces the code for an input, f (x) = c, and a decoding function that produces the reconstructed input given its code, x ≈ g(f (x)).
* Specifically, to make the decoder very simple, we choose to use matrix multiplication to map the code back into Rn. Let g(c) = Dc, where D ∈ Rnxl is the matrix defining the decoding.
* To keep the encoding problem easy, PCA constrains the columns of D to be orthogonal to each other. (Note that D is still not technically “an orthogonal matrix” unless l = n)
* To give the problem a unique solution, we constrain all of the columns of D to have unit norm.
* The first thing we need to do is figure out how to generate the optimal code point c∗ for each input point x. One way to do this is to minimize the distance between the input point x and its reconstruction, g(c∗). We can measure this distance using a norm. In the principal components algorithm, we use the L2 norm.
* We can switch to the squared L2 norm instead of the L2 norm itself, because both are minimized by the same value of c. This is because the L2 norm is nonnegative and the squaring operation is monotonically increasing for non-negative arguments.
* The function being minimized simplifies to
* We can now change the function being minimized again, to omit the first term, since this term does not depend on c:
* To make further progress, we must substitute in the definition of g(c):
* by the orthogonality and unit norm constraints on D:
* We can solve this optimization problem using vector calculus
* This makes the algorithm efficient: we can optimally encode x just using a matrix-vector operation. To encode a vector, we apply the encoder function
* Using a further matrix multiplication, we can also define the PCA reconstruction operation:
* Next, we need to choose the encoding matrix D. To do so, we revisit the idea of minimizing the L2 distance between inputs and reconstructions. However, since we will use the same matrix D to decode all of the points, we can no longer consider the points in isolation. Instead, we must minimize the Frobenius norm of the matrix of errors computed over all dimensions and all points:
* To derive the algorithm for finding D∗, we will start by considering the case where l = 1. In this case, D is just a single vector, d. substituting above equations and simplifying D into d, the problem reduces to
* It is more conventional to write scalar coefficients on the left of vector they operate on. We therefore usually write such a formula as

or, exploiting the fact that a scalar is its own transpose, as

* At this point, it can be helpful to rewrite the problem in terms of a single design matrix of examples, rather than as a sum over separate example vectors. This will allow us to use more compact notation. Let X ∈ Rm×n be the matrix defined by stacking all of the vectors describing the points, such that Xi,: = x(i)T. We can now rewrite the problem as
* Disregarding the constraint for the moment, we can simplify the Frobenius norm portion as follows:



* This optimization problem may be solved using eigendecomposition. Specifically, the optimal d is given by the eigenvector of XT X corresponding to the largest eigenvalue.
* In the general case, where l > 1, the matrix D is given by the l eigenvectors corresponding to the largest eigenvalues.