Chapter 2

Linear algebra

Linear algebra is a branch of mathematics that is widely used throughout science and engineering. However, because linear algebra is a form of continuous rather than discrete mathematics, many computer scientists have little experience with it. A good understanding of linear algebra is essential for understanding deep learning and working with deep learning algorithms. We therefore begin the technical content of the book with a focused presentation of the key linear algebra ideas that are most important in deep learning.

If you are already familiar with linear algebra, feel free to skip this chapter. If you have previous experience with these concepts but need a detailed reference sheet to review key formulas, we recommend *The Matrix Cookbook* (Petersen and Pedersen, 2006). If you have no exposure at all to linear algebra, this chapter will teach you enough to get by, but we highly recommend that you also consult another resource focused exclusively on teaching linear algebra, such as (Shilov, 1977).

2.1 Scalars, vectors, matrices and tensors

The study of linear algebra involves several types of mathematical objects:

- *Scalars*: A scalar is just a single number, in contrast to most of the other objects studied in linear algebra, which are usually arrays of multiple numbers. We write scalars in italics. We usually give scalars lower-case variable names. When we introduce them, we specify what kind of number they are. For example, we might say "Let $s \in \mathbb{R}$ be the slope of the line," while defining a real-valued scalar, or "Let $n \in \mathbb{N}$ be the number of units," while defining a natural number scalar.

- *Vectors*: A vector is an array of numbers. The numbers have an order to them, and we can identify each individual number by its index in that ordering. Typically we give vectors lower case names written in bold typeface, such as $\boldsymbol{x}$. The elements of the vector are identified by writing its name in italic typeface, with a subscript. The first element of $\boldsymbol{x}$ is x_1, the second element is x_2, and so on. We also need to say what kind of numbers are stored in the vector. If each element is in $\mathbb{R}$,

and the vector has n elements, then the vector lies in the set formed by taking the Cartesian product of $\mathbb{R}$ n times, denoted as $\mathbb{R}^n$. When we need to explicitly identify the elements of a vector, we write them as a column enclosed in square brackets:

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}.$$

We can think of vectors as identifying points in space, with each element giving the coordinate along a different axis.

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 $\boldsymbol{x}_S$. We use the $-$ sign to index the complement of a set. For example $\boldsymbol{x}_{-1}$ is the vector containing all elements of $\boldsymbol{x}$ except for x_1, and $\boldsymbol{x}_{-S}$ is the vector containing all of the elements of $\boldsymbol{x}$ except for x_1, x_3, and x_6.

- *Matrices*: A matrix is a 2-D array of numbers, so each element is identified by two indices instead of just one. We usually give matrices upper-case variable names with bold typeface, such as $\boldsymbol{A}$. If a real-valued matrix $\boldsymbol{A}$ has a height of m and a width of n, then we say that $\boldsymbol{A} \in \mathbb{R}^{m \times n}$. We usually identify the elements of a matrix using its name in italic but not bold font, and the indices are listed with separating commas. For example, $A_{1,1}$ is the upper left entry of of $\boldsymbol{A}$ and $A_{m,n}$ is the bottom right entry. We can identify all of the numbers with vertical coordinate i by writing a ":" for the horizontal coordinate. For example, $\boldsymbol{A}_{i,:}$ denotes the horizontal cross section of $\mathbf{A}$ with vertical coordinate i. This is known as the i-th *row* of $\boldsymbol{A}$. Likewise, $\boldsymbol{A}_{:,i}$ is the i-th *column* of $\boldsymbol{A}$. When we need to explicitly identify the elements of a matrix, we write them as an array enclosed in square brackets:

$$\begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix}.$$

Sometimes we may need to index matrix-valued expressions that are not just a single letter. In this case, we use subscripts after the expression, but do not convert anything to lower case. For example, $f(A_{i,j}$ gives element (i, j) of the matrix computed by applying the function f to $\boldsymbol{A}$.

- *Tensors*: In some cases we will need an array with more than two axes. In the general case, an array of numbers arranged on a regular grid with a variable number of axes is known as a *tensor.*

By convention, we consider that adding (or subtracting) a scalar and a vector yields a vector with the additive operation performed on each element. The same thing happened

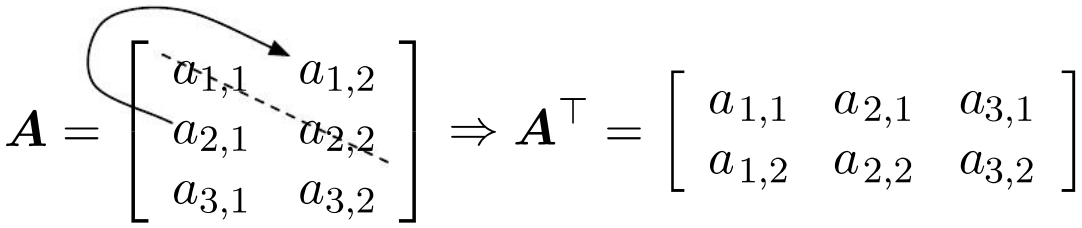

Figure 2.1: The transpose of the matrix can be thought of as a mirror image across the main diagonal.

with a matrix or a tensor. This is called a *broadcasting* operation in Python's `numpy` library.

One important operation on matrices is the *transpose*. The transpose of a matrix is the mirror image of the matrix across a diagonal line running down and to the right, starting from its upper left corner. See Fig. 2.1 for a graphical depiction of this operation. We denote the transpose of a matrix $\boldsymbol{A}$ as $\boldsymbol{A}^\top$, and it is defined such that

$$(\boldsymbol{A}^\top)_{i,j} = A_{j,i}.$$

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. Sometimes we define a vector by writing out its elements in the text inline as a row matrix, then using the transpose operator to turn it into a standard column vector, e.g. $\boldsymbol{x} = [x_1, x_2, x_3]^\top$.

We can add matrices to each other, as long as they have the same shape, just by adding their corresponding elements: $\boldsymbol{C} = \boldsymbol{A} + \boldsymbol{B}$ where $C_{i,j} = A_{i,j} + B_{i,j}$.

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: $\boldsymbol{D} = a \cdot \boldsymbol{B} + c$ where $D_{i,j} = a \cdot B_{i,j} + c$.

2.2 Multiplying matrices and vectors

One of the most important operations involving matrices is multiplication of two matrices. The *matrix product* of matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ is a third matrix $\boldsymbol{C}$. In order for this product to be defined, $\boldsymbol{A}$ must have the same number of columns as $\boldsymbol{B}$ has rows. If $\boldsymbol{A}$ is of shape $m \times n$ and $\boldsymbol{B}$ is of shape $n \times p$, then $\boldsymbol{C}$ is of shape $m \times p$. We can write the matrix product just by placing two or more matrices together, e.g.

$$\boldsymbol{C} = \boldsymbol{A}\boldsymbol{B}.$$

The product operation is defined by

$$c_{i,j} = \sum_k a_{i,k} b_{k,j}.$$

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 in this book[1] as $\boldsymbol{A} \circ \boldsymbol{B}$.

The *dot product* between two vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ of the same dimensionality is the matrix product $\boldsymbol{x}^\top \boldsymbol{y}$. We can think of the matrix product $\boldsymbol{C} = \boldsymbol{A}\boldsymbol{B}$ as computing $c_{i,j}$ as the dot product between row i of $\boldsymbol{A}$ and column j of $\boldsymbol{B}$.

Matrix product operations have many useful properties that make mathematical analysis of matrices more convenient. For example, matrix multiplication is distributive:

$$\boldsymbol{A}(\boldsymbol{B} + \boldsymbol{C}) = \boldsymbol{A}\boldsymbol{B} + \boldsymbol{A}\boldsymbol{C}.$$

It is also associative:

$$\boldsymbol{A}(\boldsymbol{B}\boldsymbol{C}) = (\boldsymbol{A}\boldsymbol{B})\boldsymbol{C}.$$

Matrix multiplication is *not* commutative, unlike scalar multiplication.

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

$$(\boldsymbol{A}\boldsymbol{B})^\top = \boldsymbol{B}^\top \boldsymbol{A}^\top.$$

Since the focus of this textbook is not linear algebra, we do not attempt to develop a comprehensive list of useful properties of the matrix product here, but the reader should be aware that many more exist.

We can also multiply matrices and vectors by scalars. To multiply by a scalar, we just multiply every element of the matrix or vector by that scalar:

$$sA = \begin{bmatrix} sa_{1,1} & \dots & sa_{1,n} \\ \vdots & \ddots & \vdots \\ sa_{m,1} & \dots & sa_{m,n} \end{bmatrix}$$

We now know enough linear algebra notation to write down a system of linear equations:

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b} \tag{2.1}$$

where $\boldsymbol{A} \in \mathbb{R}^{m \times n}$ is a known matrix, $b \in \mathbb{R}^m$ is a known vector, and $\boldsymbol{x} \in \mathbb{R}^n$ is a vector of unknown variables we would like to solve for. Each element x_i of $\boldsymbol{x}$ is one of these unknowns to solve for. Each row of $\boldsymbol{A}$ and each element of $\boldsymbol{b}$ provide another constraint. We can rewrite equation 2.1 as:

$$\boldsymbol{A}_{1,:}\, \boldsymbol{x} = b_1$$

$$\boldsymbol{A}_{2,:}\, \boldsymbol{x} = b_2$$

$$\dots$$

[1] The element-wise product is used relatively rarely, so the notation for it is not as standardized as the other operations described in this chapter.

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

Figure 2.2: *Example identity matrix*: This is $\boldsymbol{I}_3$.

$$\boldsymbol{A}_{m,:}\boldsymbol{x} = b_m$$

or, even more explicitly, as:

$$\begin{aligned} a_{1,1}x_1 + a_{1,2}x_2 + \cdots + a_{1,n}x_n &= b_1 \\ a_{2,1}x_1 + a_{2,2}x_2 + \cdots + a_{2,n}x_n &= b_2 \\ &\cdots \\ a_{m,1}x_1 + a_{m,2}x_2 + \cdots + a_{m,n}x_n &= b_m. \end{aligned}$$

Matrix-vector product notations provides a more compact representation for equations of this form.

2.3 Identity and inverse matrices

Linear algebra offers a powerful tool called *matrix inversion* that allows us to solve equation 2.1 for many values of $\boldsymbol{A}$.

To describe matrix inversion, we first need to define the concept of an *identity matrix*. An identity matrix is a matrix that does not change any vector when we multiply that vector by that matrix. We denote the n-dimensional identity matrix as $\boldsymbol{I}_n$. Formally,

$$\forall \boldsymbol{x} \in \mathbb{R}^n, \boldsymbol{I}_n\boldsymbol{x} = \boldsymbol{x}.$$

The structure of the identity matrix is simple: all of the entries along the *main diagonal* (where the row coordinate is the same as the column coordinate) are 1, while all of the other entries are zero. See Fig. 2.2 for an example.

The *matrix inverse* of $\boldsymbol{A}$ is denoted as $\boldsymbol{A}^{-1}$, and it is defined as the matrix such that

$$\boldsymbol{A}^{-1}\boldsymbol{A} = \boldsymbol{I}_n.$$

We can now solve equation 2.1 by the following steps:

$$\begin{aligned} \boldsymbol{A}\boldsymbol{x} &= \boldsymbol{b} \\ \boldsymbol{A}^{-1}\boldsymbol{A}\boldsymbol{x} &= \boldsymbol{A}^{-1}\boldsymbol{b} \\ \boldsymbol{I}_n\boldsymbol{x} &= \boldsymbol{A}^{-1}\boldsymbol{b} \\ \boldsymbol{x} &= \boldsymbol{A}^{-1}\boldsymbol{b}. \end{aligned}$$

Of course, this depends on it being possible to find $\boldsymbol{A}^{-1}$. We discuss the conditions for the existence of $\boldsymbol{A}^{-1}$ in the following section.

When $\boldsymbol{A}^{-1}$ exists, several different algorithms exist for finding it in closed form. In theory, the same inverse matrix can then be used to solve the equation many times for different values of $\boldsymbol{b}$. However, $\boldsymbol{A}^{-1}$ is primarily useful as a theoretical tool, and should not actually be used in practice for most software applications. Because $\boldsymbol{A}^{-1}$ can only be represented with limited precision on a digital computer, algorithms that make use of the value of $\boldsymbol{b}$ can usually obtain more accurate estimates of $\boldsymbol{x}$.

2.4 Linear dependence, span, and rank

In order for $\boldsymbol{A}^{-1}$ to exist, equation 2.1 must have exactly one solution for every value of $\boldsymbol{b}$. However, it is also possible for the system of equations to have no solutions or infinitely many solutions for some values of $\boldsymbol{b}$. It is not possible to have more than one but less than infinitely many solutions for a particular $\boldsymbol{b}$; if both $\boldsymbol{x}$ and $\boldsymbol{y}$ are solutions then

$$\boldsymbol{z} = \alpha\boldsymbol{x} + (1 - \alpha)\boldsymbol{y}$$

is also a solution for any real α.

To analyze how many solutions the equation has, we can think of the columns of $\boldsymbol{A}$ 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 $\boldsymbol{b}$. In this view, each element of $\boldsymbol{x}$ specifies how far we should travel in each of these directions, i.e. x_i specifies how far to move in the direction of column i:

$$\boldsymbol{A}\boldsymbol{x} = \sum_i x_i \boldsymbol{A}_{:,i}.$$

In general, this kind of operation is called a *linear combination*. Formally, a linear combination of some set of vectors $\{\boldsymbol{v}^{(1)}, \ldots, \boldsymbol{v}^{(n)}\}$ is given by multiplying each vector $\boldsymbol{v}^{(i)}$ by a corresponding scalar coefficient and adding the results:

$$\sum_i c_i \boldsymbol{v}^{(i)}.$$

The *span* of a set of vectors is the set of all points obtainable by linear combination of the original vectors.

Determining whether $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ has a solution thus amounts to testing whether $\boldsymbol{b}$ is in the span of the columns of $\boldsymbol{A}$. This particular span is known as the *column space* or the *range* of $\boldsymbol{A}$.

In order for the system $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ to have a solution for all values of $\boldsymbol{b} \in \mathbb{R}^m$, we therefore require that the column space of $\boldsymbol{A}$ be all of $\mathbb{R}^m$. If any point in $\mathbb{R}^m$ is excluded from the column space, that point is a potential value of $\boldsymbol{b}$ that has no solution. This implies immediately that $\boldsymbol{A}$ must have at least m columns, i.e., $n \geq m$. Otherwise, the dimensionality of the column space must be less than m. For example, consider a 3×2 matrix. The target $\boldsymbol{b}$ is 3-D, but $\boldsymbol{x}$ is only 2-D, so modifying the value of $\boldsymbol{x}$ at best

allows us to trace out a 2-D plane within $\mathbb{R}^3$. The equation has a solution if and only if $\boldsymbol{b}$ lies on that plane.

Having $n \geq m$ 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 equal to each other. This has the same column space as a 2×1 matrix containing only one copy of the replicated column. In other words, the column space is still just a line, and fails to encompass all of $\mathbb{R}^2$, even though there are two columns.

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 $\mathbb{R}^m$, the matrix must have at least m linearly independent columns. This condition is both necessary and sufficient for equation 2.1 to have a solution for every value of $\boldsymbol{b}$.

In order for the matrix to have an inverse, we additionally need to ensure that equation 2.1 has *at most* one solution for each value of $\boldsymbol{b}$. To do so, we need to ensure that the matrix has at most m columns. Otherwise there is more than one way of parameterizing 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 $\boldsymbol{A}$ is not square or is square but singular, it can still be possible to solve the equation. However, we can not 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:

$$\boldsymbol{A}\boldsymbol{A}^{-1} = \boldsymbol{I}.$$

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

2.5 Norms

Sometimes we need to measure the size of a vector. In machine learning, we usually measure the size of vectors using an L^p norm:

$$||\boldsymbol{x}||_p = \left(\sum_i |x_i|^p \right)^{\frac{1}{p}}$$

for $p \in \mathbb{R}, p \geq 1$.

Norms, including the L^p norm, are functions mapping vectors to non-negative values, satisfying these properties that make them behave like distances between points:

- $f(\boldsymbol{x}) = 0 \Rightarrow \boldsymbol{x} = \boldsymbol{0}$

- $f(\boldsymbol{x} + \boldsymbol{y}) \leq f(\boldsymbol{x}) + f(\boldsymbol{y})$ (the *triangle inequality*)
- $\forall \alpha \in \mathbb{R}, f(\alpha \boldsymbol{x}) = |\alpha| f(\boldsymbol{x})$

The L^2 norm, with $p = 2$, is known as the *Euclidean norm*. It is simply the Euclidean distance from the origin to the point identified by $\boldsymbol{x}$. This is probably the most common norm used in machine learning. It is also common to measure the size of a vector using the squared L^2 norm, which can be calculated simply as $\boldsymbol{x}^\top \boldsymbol{x}$.

The squared L^2 norm is more convenient to work with mathematically and computationally than the L^2 norm itself. For example, the derivatives of the squared L^2 norm with respect to each element of $\boldsymbol{x}$ each depend only on the corresponding element of $\boldsymbol{x}$, while all of the derivatives of the L^2 norm depend on the entire vector. In many contexts, the L^2 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 L^1 norm. The L^1 norm may be simplified to

$$||\boldsymbol{x}||_1 = \sum_i |x_i|.$$

The L^1 norm is commonly used in machine learning when the difference between zero and nonzero elements is very important. Every time an element of $\boldsymbol{x}$ moves away from 0 by ϵ, the L^1 norm increases by ϵ.

We sometimes measure the size of the vector by counting its number of nonzero elements (and when we use the L^1 norm, we often use it as a proxy for this function). Some authors refer to this function as the "l_0 norm," but this is incorrect terminology, because scaling the vector by α does not change the number of nonzero entries.

One other norm that commonly arises in machine learning is the l_∞ norm, also known as *the max norm*. This norm simplifies to

$$||\boldsymbol{x}||_\infty = \max_i |x_i|,$$

e.g., 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*

$$||A||_F = \sqrt{\sum_{i,j} a_{i,j}^2},$$

which is analogous to the L^2 norm of a vector.

The dot product of two vectors can be rewritten in terms of norms. Specifically,

$$\boldsymbol{x}^\top \boldsymbol{y} = ||\boldsymbol{x}||_2 \, ||\boldsymbol{y}||_2 \cos\theta$$

where θ is the angle between $\boldsymbol{x}$ and $\boldsymbol{y}$.

2.6 Special kinds of matrices and vectors

Some special kinds of matrices and vectors are particularly useful.

Diagonal matrices only have non-zero entries along the main diagonal. Formally, a matrix $\boldsymbol{D}$ is diagonal if and only if $d_{i,j} = 0$ for all $i \neq j$. We've already seen one example of a diagonal matrix: the identity matrix, where all of the diagonal entries are 1. In this book[2], we write $\text{diag}(\boldsymbol{v})$ to denote a square diagonal matrix whose diagonal entries are given by the entries of the vector $\boldsymbol{v}$. Diagonal matrices are of interest in part because multiplying by a diagonal matrix is very computationally efficient. To compute $\text{diag}(\boldsymbol{v})\boldsymbol{x}$, we only need to scale each element x_i by v_i. In other words, $\text{diag}(\boldsymbol{v})\boldsymbol{x} = \boldsymbol{v} \circ \boldsymbol{x}$. Inverting a diagonal matrix is also efficient. The inverse exists only if every diagonal entry is nonzero, and in that case, $\text{diag}(\boldsymbol{v})^{-1} = \text{diag}([1/v_1, \ldots, 1/v_n]^\top)$. In many cases, we may derive some very general machine learning algorithm in terms of arbitrary matrices, but obtain a less expensive (and less descriptive) algorithm by restricting some matrices to be diagonal.

A *symmetric* matrix is any matrix that is equal to its own transpose:

$$\boldsymbol{A} = \boldsymbol{A}^\top .$$

Symmetric matrices often arise when the entries are generated by some function of two arguments that does not depend on the order of the arguments. For example, if $\boldsymbol{A}$ is a matrix of distance measurements, with $a_{i,j}$ giving the distance from point i to point j, then $a_{i,j} = a_{j,i}$ because distance functions are symmetric.

A *unit vector* is a vector with *unit norm*:

$$||\boldsymbol{x}||_2 = 1.$$

A vector $\boldsymbol{x}$ and a vector $\boldsymbol{y}$ are *orthogonal* to each other if $\boldsymbol{x}^\top \boldsymbol{y} = 0$. If both vectors have nonzero norm, this means that they are at 90 degree angles to each other. In $\mathbb{R}^n$, at most n vectors may be mutually orthogonal with nonzero norm. 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:

$$\boldsymbol{A}^\top \boldsymbol{A} = \boldsymbol{A}\boldsymbol{A}^\top = \boldsymbol{I}.$$

This implies that

$$\boldsymbol{A}^{-1} = \boldsymbol{A}^\top ,$$

so orthogonal matrices are of interest because their inverse is very cheap to compute. Pay careful attention to the definition of orthogonal matrices. Counterintuitively, their rows are not merely orthogonal but fully orthonormal. There is no special term for a matrix whose rows or columns are orthogonal but not orthonormal.

[2] There is not a standardized notation for constructing a diagonal matrix from a vector.

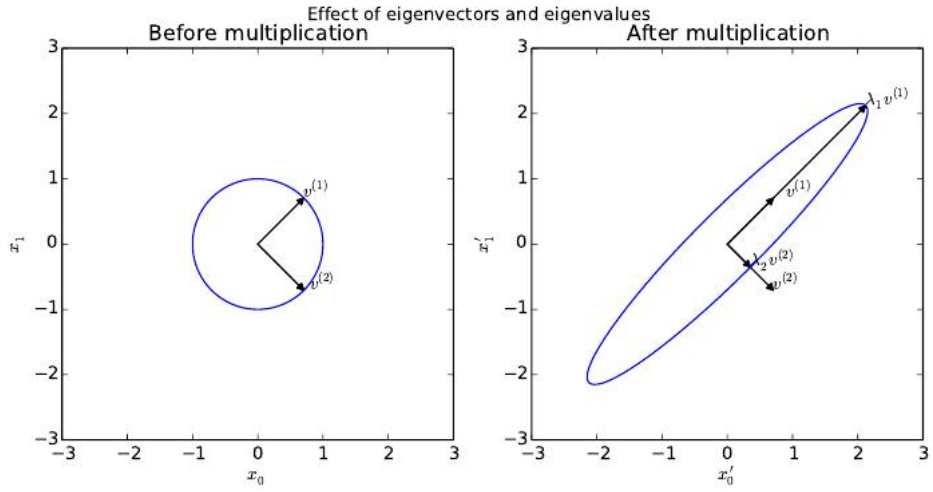

Figure 2.3: An example of the effect of eigenvectors and eigenvalues. Here, we have a matrix $\boldsymbol{A}$ with two orthonormal eigenvectors, $\boldsymbol{v}^{(1)}$ with eigenvalue λ_1 and $\boldsymbol{v}^{(2)}$ with eigenvalue λ_2. *Left)* We plot the set of all unit vectors $\boldsymbol{u} \in \mathbb{R}^2$ as a blue circle. *Right)* We plot the set of all points $\boldsymbol{Au}$. By observing the way that $\boldsymbol{A}$ distorts the unit circle, we can see that it scales space in direction $\boldsymbol{v}^{(i)}$ by λ_i.

2.7 Eigendecomposition

An *eigenvector* of a square matrix $\boldsymbol{A}$ is a non-zero vector v such that multiplication by $\boldsymbol{A}$ alters only the scale of $\boldsymbol{v}$:

$$\boldsymbol{Av} = \lambda \boldsymbol{v}.$$

The scalar λ is known as the *eigenvalue* corresponding to this eigenvector. (One can also find a *left eigenvector* such that $\boldsymbol{v}^\top \boldsymbol{A} = \lambda \boldsymbol{v}$, but we are usually concerned with right eigenvectors).

Note that if $\boldsymbol{v}$ is an eigenvector of $\boldsymbol{A}$, then so is any rescaled vector $s\boldsymbol{v}$ for $s \in \mathbb{R}, s \neq 0$. Moreover, $s\boldsymbol{v}$ still has the same eigenvalue. For this reason, we usually only look for unit eigenvectors.

We can construct a matrix $\boldsymbol{A}$ with eigenvectors $\{\boldsymbol{v}^{(1)}, \ldots, \boldsymbol{v}^{(n)}\}$ and corresponding eigenvalues $\{\lambda_1, \ldots, \lambda_n\}$ by concatenating the eigenvectors into a matrix $\boldsymbol{v}$, one column per eigenvector, and concatenating the eigenvalues into a vector $\boldsymbol{\lambda}$. Then the matrix

$$\boldsymbol{A} = \boldsymbol{v}\text{diag}(\boldsymbol{\lambda})\boldsymbol{v}^{-1}$$

has the desired eigenvalues and eigenvectors. If we make $\boldsymbol{v}$ an orthogonal matrix, then we can think of $\boldsymbol{A}$ as scaling space by λ_i in direction $\boldsymbol{v}^{(i)}$. See Fig. 2.3 for an example.

We have seen that *constructing* matrices with specific eigenvalues and eigenvectors allows us to stretch space in desired directions. However, we often want to *decompose* matrices into their eigenvalues and eigenvectors. Doing so can help us to analyze certain properties of the matrix, much as decomposing an integer into its prime factors can help us understand the behavior of that integer.

Not every matrix can be decomposed into eigenvalues and eigenvectors. In some cases, the decomposition exists, but may involve complex rather than real numbers. Fortunately, in this book, we usually need to decompose only a specific class of matrices that have a simple decomposition. Specifically, every real symmetric matrix can be decomposed into an expression using only real-valued eigenvectors and eigenvalues:

$$\boldsymbol{A} = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^\top,$$

where $\boldsymbol{Q}$ is an orthogonal matrix composed of eigenvectors of $\boldsymbol{A}$, and $\boldsymbol{\Lambda}$ is a diagonal matrix, with $\lambda_{i,i}$ being the eigenvalue corresponding to $\boldsymbol{Q}_{:,i}$.

While any real symmetric matrix $\boldsymbol{A}$ is guaranteed to have an eigendecomposition, the eigendecomposition is not unique. If any two or more eigenvectors share the same eigenvalue, then any set of orthogonal vectors lying in their span are also eigenvectors with that eigenvalue, and we could equivalently choose a $\boldsymbol{Q}$ using those eigenvectors instead. By convention, we usually sort the entries of $\boldsymbol{\Lambda}$ in descending order. Under this convention, the eigendecomposition is unique only if all of the eigenvalues are unique.

The eigendecomposition of a matrix tells us many useful facts about the matrix. The matrix is singular if and only if any of the eigenvalues are 0. The eigendecomposition can also be used to optimize quadratic expressions of the form $f(\boldsymbol{x}) = \boldsymbol{x}^\top\boldsymbol{A}\boldsymbol{x}$ subject to $||\boldsymbol{x}||_2 = 1$. Whenever $\boldsymbol{x}$ is equal to an eigenvector of $\boldsymbol{A}$, 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 $\forall \boldsymbol{x}, \boldsymbol{x}^\top \boldsymbol{A}\boldsymbol{x} \geq 0$. Positive definite matrices additionally guarantee that $\boldsymbol{x}^\top\boldsymbol{A}\boldsymbol{x} = 0 \Rightarrow \boldsymbol{x} = \boldsymbol{0}$.

2.8 Singular Value Decomposition

The singular value decomposition is a general purpose matrix factorization method that decomposes an $n \times m$ matrix $\boldsymbol{X}$ into three distinct matrices: $\boldsymbol{X} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{W}^\top$, where $\boldsymbol{\Sigma}$ is a rectangular diagonal matrix[3], $\boldsymbol{U}$ is an $n \times n$-dimensional matrix whose columns are mutually orthonormal and similarly $\boldsymbol{W}$ is an $m \times m$-dimensional matrix whose columns are mutually orthonormal. The elements along the diagonal of $\boldsymbol{\Sigma}$, σ_i for $i \in \{1, \ldots, \min\{n, m\}\}$, are referred to as the singular values and the n columns or μ and the m columns of $\boldsymbol{W}$ are commonly referred to the *left-singular vectors* and *right-singular vectors* of $\mathbf{X}$ respectively.

[3] A rectangular $n \times m$-dimensional diagonal matrix can be seen as a consisting of two sub-matrices, one square diagonal matrix of size $c \times c$ where $c = \min\{n, m\}$ and a matrix of to fill out the rest of the rectangular matrix.

The singular value decomposition has many useful properties, however for our purposes, we will concentrate on their relation to the eigen-decomposition. Specifically, the left-singular vectors of $\boldsymbol{X}$ (the columns $\boldsymbol{U}_{:,i}$, for $i \in \{1, \ldots, n\}$) are the eigenvectors of $\boldsymbol{X}\boldsymbol{X}^\top$. The right-singular vectors of $\boldsymbol{X}$ (the columns $\boldsymbol{W}_{:,j}$, for $j \in \{1, \ldots, m\}$) are the eigenvectors of $\boldsymbol{X}^\top\boldsymbol{X}$. Finally, the non-zero singular values of $\boldsymbol{X}$ (σ_i: for all i such that $\sigma_i \neq 0$) are the square roots of the non-zero eigenvalues for both $\boldsymbol{X}\boldsymbol{X}^\top$ and $\boldsymbol{X}^\top\boldsymbol{X}$.

2.9 The trace operator

The trace operator gives the sum of all of the diagonal entries of a matrix:

$$\text{Tr}(\boldsymbol{A}) = \sum_i a_{i,i}.$$

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:

$$||\boldsymbol{A}||_F = \sqrt{\text{Tr}(\boldsymbol{A}^\top\boldsymbol{A})}.$$

The trace operator also has many useful properties that make it easy to manipulate expressions involving the trace operator. For example, the trace operator is invariant to the transpose operator:

$$\text{Tr}(\boldsymbol{A}) = \text{Tr}(\boldsymbol{A}^\top).$$

The trace of a square matrix composed of many factors is also invariant to moving the last factor into the first position:

$$\text{Tr}(\boldsymbol{A}\boldsymbol{B}\boldsymbol{C}) = \text{Tr}(\boldsymbol{B}\boldsymbol{C}\boldsymbol{A}) = \text{Tr}(\boldsymbol{C}\boldsymbol{A}\boldsymbol{B})$$

or more generally,

$$\text{Tr}(\Pi_{i=1}^n \boldsymbol{F}^{(i)}) = \text{Tr}(\boldsymbol{F}^{(n)}\Pi_{i=1}^{n-1}\boldsymbol{F}^{(i)}).$$

2.10 Determinant

The determinant of a square matrix, denoted $\det(\boldsymbol{A})$, is a function mapping matrices to real scalars. The determinant is equal to the product of all the matrix's eigenvalues. 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 of its volume. If the determinant is 1, then the transformation is volume-preserving.

2.11 Example: Principal components analysis

One simple machine learning algorithm, *principal components analysis (PCA)* can be derived using only knowledge of basic linear algebra.

Suppose we have a collection of m points $\{\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(m)}\}$ in $\mathbb{R}^n$. Suppose we would like to apply lossy compression to these points, i.e. we would like to find a way of storing the points that requires less memory but may lose some precision. We would like to lose as little precision as possible.

One way we can encode these points is to represent a lower-dimensional version of them. For each point $\boldsymbol{x}^{(i)} \in \mathbb{R}^n$ we will find a corresponding code vector $\boldsymbol{c}^{(i)} \in \mathbb{R}^l$. If l is smaller than n, it will take less memory to store the code points than the original data. We can use matrix multiplication to map the code back into $\mathbb{R}^n$. Let the reconstruction $r(\boldsymbol{c}) = \boldsymbol{D}\boldsymbol{c}$, where $\boldsymbol{D} \in \mathbb{R}^{n \times l}$ is the matrix defining the decoding.

To simplify the computation of the optimal encoding, we constrain the columns of $\boldsymbol{D}$ to be orthogonal to each other. (Note that $\boldsymbol{D}$ is still not technically "an orthogonal matrix" unless $l = n$)

With the problem as described so far, many solutions are possible, because we can increase the scale of $\boldsymbol{D}_{:,i}$ if we decrease c_i proportionally for all points. To give the problem a unique solution, we constrain all of the columns of $\boldsymbol{D}$ to have unit norm.

In order to turn this basic idea into an algorithm we can implement, the first thing we need to do is figure out how to generate the optimal code point $\boldsymbol{c}^*$ for each input point $\boldsymbol{x}$. One way to do this is to minimize the distance between the input point $\boldsymbol{x}$ and its reconstruction, $r(\boldsymbol{c})$. We can measure this distance using a norm. In the principal components algorithm, we use the L^2 norm:

$$\boldsymbol{c}^* = \arg\min_{\boldsymbol{c}} ||\boldsymbol{x} - r(\boldsymbol{c})||_2.$$

We can switch to the squared L^2 norm instead of the L^2 norm itself, because both are minimized by the same value of $\boldsymbol{c}$. This is because the L^2 norm is non-negative and the squaring operation is monotonically increasing for non-negative arguments.

$$\boldsymbol{c}^* = \arg\min_{\boldsymbol{c}} ||\boldsymbol{x} - r(\boldsymbol{c})||_2^2$$

The function being minimized simplifies to

$$(\boldsymbol{x} - r(\boldsymbol{c}))^\top (\boldsymbol{x} - r(\boldsymbol{c}))$$

(by the definition of the L^2 norm)

$$= \boldsymbol{x}^\top \boldsymbol{x} - \boldsymbol{x}^\top r(\boldsymbol{c}) - r(\boldsymbol{c})^\top \boldsymbol{x} + r(\boldsymbol{c})^\top r(\boldsymbol{c})$$

(by the distributive property)

$$= \boldsymbol{x}^\top \boldsymbol{x} - 2\boldsymbol{x}^\top r(\boldsymbol{c}) + r(\boldsymbol{c})^\top r(\boldsymbol{c})$$

(because a scalar is equal to the transpose of itself).

We can now change the function being minimized again, to omit the first term, since this term does not depend on $\boldsymbol{c}$:

$$\boldsymbol{c}^* = \arg\min_{\boldsymbol{c}} -2\boldsymbol{x}^\top r(\boldsymbol{c}) + r(\boldsymbol{c})^\top r(\boldsymbol{c}).$$

To make further progress, we must substitute in the definition of $r(\boldsymbol{c})$:

$$\boldsymbol{c}^* = \arg\min_{\boldsymbol{c}} -2\boldsymbol{x}^\top \boldsymbol{D}\boldsymbol{c} + \boldsymbol{c}^\top \boldsymbol{D}^\top \boldsymbol{D}\boldsymbol{c}$$

$$= \arg\min_{\boldsymbol{c}} -2\boldsymbol{x}^\top \boldsymbol{D}\boldsymbol{c} + \boldsymbol{c}^\top \boldsymbol{I}_l \boldsymbol{c}$$

(by the orthogonality and unit norm constraints on $\boldsymbol{D}$)

$$= \arg\min_{\boldsymbol{c}} -2\boldsymbol{x}^\top \boldsymbol{D}\boldsymbol{c} + \boldsymbol{c}^\top \boldsymbol{c}$$

We can solve this optimization problem using vector calculus (see section 4.3 if you do not know how to do this):

$$\nabla_{\boldsymbol{c}}(-2\boldsymbol{x}^\top \boldsymbol{D}\boldsymbol{c} + \boldsymbol{c}^\top \boldsymbol{c}) = \boldsymbol{0}$$

$$-2\boldsymbol{x}^\top \boldsymbol{D} + 2\boldsymbol{c} = \boldsymbol{0}$$

$$\boldsymbol{c} = \boldsymbol{x}^\top \boldsymbol{D}. \tag{2.2}$$

This is good news: we can optimally encode $\boldsymbol{x}$ just using a vector-product operation.

Next, we need to choose the encoding matrix $\boldsymbol{D}$. To do so, we revisit the idea of minimizing the L^2 distance between inputs and reconstructions. However, since we will use the same matrix $\boldsymbol{D}$ to decode all of the points, we can no longer consider the points in isolation. Instead, we must minimize the L^2 distance for all the points simultaneously:

$$\boldsymbol{D}^* = \arg\min_{\boldsymbol{D}} \sqrt{\sum_{i,j} (x_j^{(i)} - r_j^{(i)})^2} \text{ subject to } \boldsymbol{D}^\top \boldsymbol{D} = \boldsymbol{I}_l \tag{2.3}$$

To derive the algorithm for finding $\boldsymbol{D}^*$, we will start by considering the case where $l = 1$. In this case, $\boldsymbol{D}$ is just a single vector, $\boldsymbol{d}$. Substituting equation 2.2 into equation 2.3 and simplifying $\boldsymbol{D}$ into $\boldsymbol{d}$, the problem reduces to

$$\boldsymbol{d}^* = \arg\min_{\boldsymbol{d}} \sum_i ||\boldsymbol{x}^{(i)} - \boldsymbol{x}^{(i)\top} \boldsymbol{d}\boldsymbol{d}||_2^2 \text{ subject to } ||d||_2 = 1.$$

At this point, it can be helpful to rewrite the problem in terms of matrices. This will allow us to use more compact notation. Let $\boldsymbol{x} \in \mathbb{R}^{m \times n}$ be the matrix defined by stacking all of the vectors describing the points, such that $\boldsymbol{x}_{i,:} = \boldsymbol{x}^{(i)}$. We can now rewrite the problem as

$$\boldsymbol{d}^* = \arg\min_{\boldsymbol{d}} ||\boldsymbol{x} - \boldsymbol{x}\boldsymbol{d}\boldsymbol{d}^\top||_F^2 \text{ subject to } \boldsymbol{d}^\top \boldsymbol{d} = 1$$

$$= \arg\min_{\boldsymbol{d}} \operatorname{Tr} \left(\boldsymbol{x} - \boldsymbol{x}\boldsymbol{d}\boldsymbol{d}^\top \right)^\top \left(\boldsymbol{x} - \boldsymbol{x}\boldsymbol{d}\boldsymbol{d}^\top \right) \text{ subject to } \boldsymbol{d}^\top \boldsymbol{d} = 1$$

(by the alternate definition of the Frobenius norm)

$$= \arg\min_{\boldsymbol{d}} \operatorname{Tr}(\boldsymbol{x}^\top \boldsymbol{x} - \boldsymbol{x}^\top \boldsymbol{x}\boldsymbol{d}\boldsymbol{d}^\top - \boldsymbol{d}\boldsymbol{d}^\top \boldsymbol{x}^\top \boldsymbol{x} + \boldsymbol{d}\boldsymbol{d}^\top \boldsymbol{x}^\top \boldsymbol{x}\boldsymbol{d}\boldsymbol{d}^\top) \text{ subject to } \boldsymbol{d}^\top \boldsymbol{d} = 1$$

$$= \arg\min_{\boldsymbol{d}} \operatorname{Tr}(\boldsymbol{x}^\top \boldsymbol{x}) - \operatorname{Tr}(\boldsymbol{x}^\top \boldsymbol{x}\boldsymbol{d}\boldsymbol{d}^\top) - \operatorname{Tr}(\boldsymbol{d}\boldsymbol{d}^\top \boldsymbol{x}^\top \boldsymbol{x}) + \operatorname{Tr}(\boldsymbol{d}\boldsymbol{d}^\top \boldsymbol{x}^\top \boldsymbol{x}\boldsymbol{d}\boldsymbol{d}^\top) \text{ subject to } \boldsymbol{d}^\top \boldsymbol{d} = 1$$

$$= \arg\min_{\boldsymbol{d}} - \operatorname{Tr}(\boldsymbol{x}^\top \boldsymbol{x}\boldsymbol{d}\boldsymbol{d}^\top) - \operatorname{Tr}(\boldsymbol{d}\boldsymbol{d}^\top \boldsymbol{x}^\top \boldsymbol{x}) + \operatorname{Tr}(\boldsymbol{d}\boldsymbol{d}^\top \boldsymbol{x}^\top \boldsymbol{x}\boldsymbol{d}\boldsymbol{d}^\top) \text{ subject to } \boldsymbol{d}^\top \boldsymbol{d} = 1$$

(because terms not involving $\boldsymbol{d}$ do not affect the $\arg\min$)

$$= \arg\min_{\boldsymbol{d}} -2 \operatorname{Tr}(\boldsymbol{x}^\top \boldsymbol{x}\boldsymbol{d}\boldsymbol{d}^\top) + \operatorname{Tr}(\boldsymbol{d}\boldsymbol{d}^\top \boldsymbol{x}^\top \boldsymbol{x}\boldsymbol{d}\boldsymbol{d}^\top) \text{ subject to } \boldsymbol{d}^\top \boldsymbol{d} = 1$$

(because the trace is invariant to transpose)

$$= \arg\min_{\boldsymbol{d}} -2 \operatorname{Tr}(\boldsymbol{x}^\top \boldsymbol{x}\boldsymbol{d}\boldsymbol{d}^\top) + \operatorname{Tr}(\boldsymbol{x}^\top \boldsymbol{x}\boldsymbol{d}\boldsymbol{d}^\top \boldsymbol{d}\boldsymbol{d}^\top) \text{ subject to } \boldsymbol{d}^\top \boldsymbol{d} = 1$$

(because we can cycle the order of the matrices inside a trace)

$$= \arg\min_{\boldsymbol{d}} -2 \operatorname{Tr}(\boldsymbol{x}^\top \boldsymbol{x}\boldsymbol{d}\boldsymbol{d}^\top) + \operatorname{Tr}(\boldsymbol{x}^\top \boldsymbol{x}\boldsymbol{d}\boldsymbol{d}^\top) \text{ subject to } \boldsymbol{d}^\top \boldsymbol{d} = 1$$

(due to the constraint)

$$= \arg\min_{\boldsymbol{d}} - \operatorname{Tr}(\boldsymbol{x}^\top \boldsymbol{x}\boldsymbol{d}\boldsymbol{d}^\top) \text{ subject to } \boldsymbol{d}^\top \boldsymbol{d} = 1$$

$$= \arg\max_{\boldsymbol{d}} \operatorname{Tr}(\boldsymbol{x}^\top \boldsymbol{x}\boldsymbol{d}\boldsymbol{d}^\top) \text{ subject to } \boldsymbol{d}^\top \boldsymbol{d} = 1$$

$$= \arg\max_{\boldsymbol{d}} \operatorname{Tr}(\boldsymbol{d}^\top \boldsymbol{x}^\top \boldsymbol{x}\boldsymbol{d}) \text{ subject to } \boldsymbol{d}^\top \boldsymbol{d} = 1$$

This optimization problem may be solved using eigendecomposition. Specifically, the optimal $\boldsymbol{d}$ is given by the eigenvector of $\boldsymbol{x}^\top \boldsymbol{x}$ corresponding to the largest eigenvalue.

In the general case, where $l > 1$, $\boldsymbol{D}$ is given by the l eigenvectors corresponding to the largest eigenvalues. This may be shown using proof by induction. We recommend writing this proof as an exercise.