Chapter 3
Linear Algebra

'Obvious' is the most dangerous word in mathematics.
– Eric Temple Bell, Scottish mathematician.

Abstract In this chapter we will summarise the most important aspects of linear algebra. We will look at vectors, matrices and with what operations they can be combined. We will look at dot and cross products between vectors and element-wise operations (often used in machine learning). Next, we will look at matrices, at the transpose, inverse and trace operations. We will discuss what is the determinant of a matrix and what are and how to calculate eigenvalues and eigenvectors. Then we will discuss what role matrices have in machine learning, and especially in deep learning.

3.1 Motivation

Linear algebra, in probably the most simplified description possibly conceived by anyone, concerns itself with arrays of numbers called vectors and matrices. These data structures appear continuously when you deal with any kind of data. Tables of data where each row makes up one observation and each column is a feature describing the data are the most common type of data. If you have difficulty imagining what I am talking about, imagine a Microsoft Excel Sheet full of numbers. Practically any data type can be brought in array form, even what is typically called *unstructured*. Images are 3-dimensional arrays made up of the x and y positions of a pixel and its colour values expressed as three numbers (The Red, Green and Blue values) respectively. In Figure 3.1 you can see how an image is nothing else that an array of numbers. In the left panel you can see the famous portrait of Einstein. The middle panel is the same image reshaped to 20×27 pixel with the gray value of each pixel overimposed . In the right panel you see only the array of numbers that make up the image. Sound is similar and can be converted analogously into an array of numbers.

It is therefore natural to expect that in deep learning arrays of numbers (in particular matrices) play a crucial role. Understanding linear algebra is therefore fundamental to be able to understand how deep learning works, and to implement the algorithms efficiently.

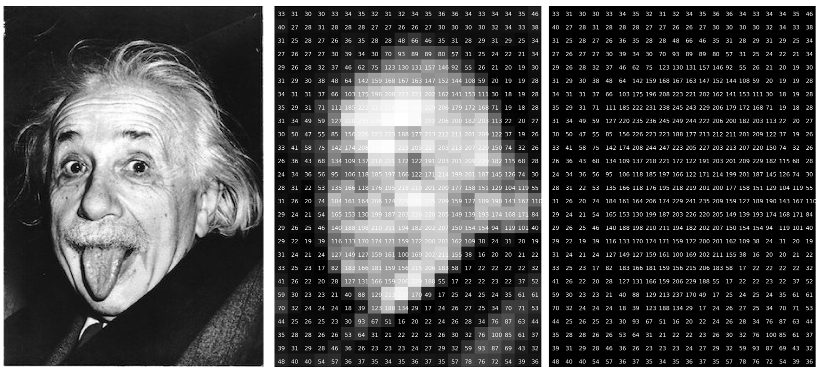

Fig. 3.1 Here you can see how an image is nothing else that an array of numbers. In the left panel you can see the famous portrait of Einstein. The middle panel is the same image reshaped to 20×27 pixel with the gray value of each pixel overimposed. In the right panel you see only the array of numbers that make up the image.

In Part II, you will see how in deep learning almost everything is a matrix[1] (the weights of a neural networks, the input dataset, etc.). All Python libraries that are used to implement and train neural networks today use multi-dimensional arrays as data types (the two most important libraries are pyTorch [1] and TensorFlow [2]). Arrays of numbers are so important, that the name of the library TensorFlow from Google comes from *Tensor*, that is, in its most simple definition, a multi-dimensional array of numbers.

Info Vectorization in Python

Linear algebra is such an important aspect of machine learning, that entire Python libraries are built around it. `numpy` for example is the most known Python numerical library. At its core lie functions implemented in C++ that takes advantage of arrays and allow to perform several important operations in deep learning in one shot instead of expensive (from a computational point of view) loops.
For example suppose you want to multiply two arrays of 10^7 numbers element-wise. In Python you can easily implement it as a loop or you can use a specifically optimised `numpy` function `multiply()`. This second option is roughly 100 times faster than a simple loop. In Deep Learning you perform operations over and over again, thus making this kind of optimisations extremely important.

[1] To be technically precise, often a multi-dimensional array. Remember that a matrix is a 2-dimensional array of numbers, multi-dimensional arrays of numbers are called Tensors.

This chapter contains the most basic concepts that everyone who want to use deep learning should understand. Many topics in linear algebra are very interesting and important (eigenvalues and eigenvectors, systems of linear equations, vector spaces, and many more) but could not be included in this short chapter for space reasons, and most importantly because they are not used as often in deep learning. The reader interested in learning more can consult several books, as for example [3, 4, 5].

3.2 Vectors

Let us start with the simpleset form of array: a 1-dimensional list of numbers called a *vector*. Vectors are ordered collections of numbers (real or complex). They typically are indicated with a letter in bold-face, with a line below the symbol, or an arrow above. If a vector contain real numbers it is called a *real* vector, if it contains complex numbers it is called a *complex* vector. In this book we will deal only with real vectors. A vector are indicated by the following symbols

$$\mathbf{x},\ \underline{x},\ \overrightarrow{x} \tag{3.1}$$

In this book we will denote vectors with a bold-face letter, e.g. $\mathbf{x}$. Some examples of vectors are

$$\begin{aligned}
\mathbf{x} &= (2,3,6)\\
\mathbf{x} &= (1,2,3.4,6.4,6)\\
\mathbf{x} &= (\pi,\pi/2,1)\\
\mathbf{x} &= (2,3,6).
\end{aligned} \tag{3.2}$$

In general the components of a n-dimensional vector $\mathbf{x} \in \mathbb{R}^n$ are indicated with x_i with $i = 1, ..., n$. So we can write

$$\mathbf{x} = (x_1, x_2, ..., x_n). \tag{3.3}$$

Sometime in books and on websites vectors are represented in horizontal notation (as we have done so far) and some time in vertical. For example you can find

$$\mathbf{x} = (x_1, x_2, ..., x_n) \tag{3.4}$$

or

$$\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \tag{3.5}$$

Technically speaking this should not be relevant if the operations that you want to perform on vectors are correctly defined. Additionally, as long as one deals only with vectors, this distinction, horizontal and vertical, is completely irrelevant. We will discuss it again when we will combine vectors and matrices.

When we sum or subtract vectors, this is done element-wise. For example consider two vectors $\mathbf{x} \in \mathbb{R}^n$ and $\mathbf{y} \in \mathbb{R}^n$. We have

$$\mathbf{x} + \mathbf{y} = (x_1 + y_1, x_2 + y_2, ..., x_n + y_n) \tag{3.6}$$

Note that we can sum only vectors that have the same number of components. In the same way we will have

$$\mathbf{x} - \mathbf{y} = (x_1 - y_1, x_2 - y_2, ..., x_n - y_n) \tag{3.7}$$

Multiplication between vectors is not so easily defined. We will deal with it in the next section. Division between vectors cannot be uniquely defined, as will be discussed at the end of Section 3.2.3.

Info Element-wise sum in Python

When you are programming in Python and you are using for example `numpy` [6] or `TensorFlow` [2], you will realize very soon that the standard operations between numpy or TensorFlow arrays are done element-wise. For example if `a` and `b` are two numpy array, `a+b` is the sum of the two arrays done elementwise, as you would expect from the previous discussion. Things get slightly more complicated when you are multiplying arrays, as we will discuss in Section 3.2.3.

3.2.1 Geometrical Interpretation of Vectors

We have defined the vector in an abstract way, as an ordered collection of numbers. But it is easy to get an intuitive understanding of what a vector is, if we restrict ourselves to 2 or 3 dimension.

For example in 2 dimensions, a vector $\mathbf{x} = (x_1, x_2)$ can be thought as an arrow in an x, y space with two components x_1 and x_2 along the x and y axis respectively. The length of the arrow is typically indicated with $|\mathbf{x}|$ and is called its norm (see Section 3.2.2 for more details). In Figure 3.2 you can see a graphical representation of a vector in 2 dimensions.

The same interpretation can be given in 3 dimensions. It is better understood graphically from Figure 3.3. Of course if the vector has n dimensions (with $n > 3$) then it is impossible to visualise it, but the interpretation remains valid. x_i is the component of the vector along the i^{th} axis in an n-dimensional space.

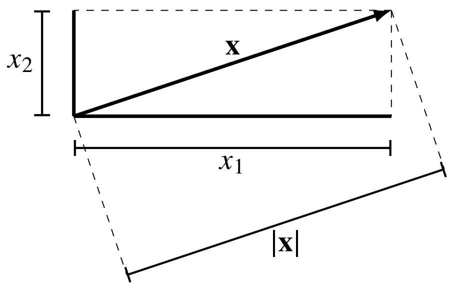

Fig. 3.2 A geometrical interpretation of a vector $\mathbf{x} \in \mathbb{R}^2$. The vector can be thought as an arrow in an x, y space with two components x_1 and x_2 along the x and y axis respectively. The length of the arrow is typically indicated with $|\mathbf{x}|$ and is called its norm (see Section 3.2.2 for more details).

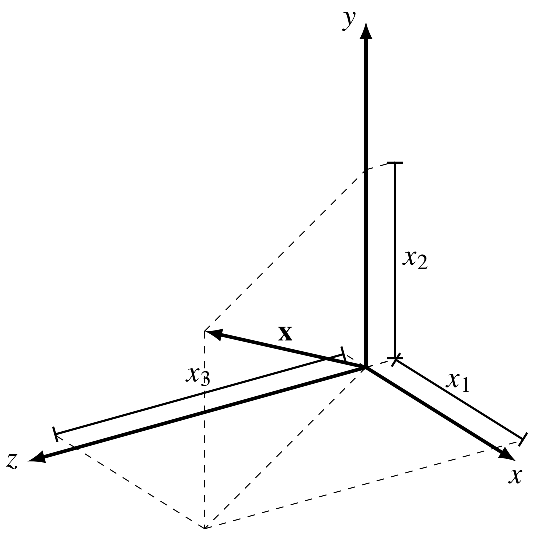

Fig. 3.3 A geometrical interpretation of a 3-dimensional vector $\mathbf{x} \in \mathbb{R}^3$. The vector can be thought as an arrow in an x, y, z space with 3 components x_1, x_2 and x_3 along the x,y and z axis respectively. The length of the arrow is typically indicated with $|\mathbf{x}|$ and is called its norm (see Section 3.2.2 for more details).

3.2.2 The Norm

In general terms a *norm* can be defined as follows:

Definition 3.1 A norm is a real-valued function $|\cdot| : \mathbb{R}^n \rightarrow \mathbb{R}$ that satisfy the following properties (expressed here in terms of a generic vector $\mathbf{x} \in \mathbb{R}^n$)

1. $|\mathbf{x}| > 0$ for $\mathbf{x} \neq \mathbf{0}$; $|\mathbf{x}| = \mathbf{0} \iff \mathbf{x} = \mathbf{0}$.
2. $|k\mathbf{x}| = |k||\mathbf{x}| \ \ \forall k \in \mathbb{R}$.

3. $|\mathbf{x}+\mathbf{y}| \leq |\mathbf{x}| + |\mathbf{y}|$

Where we have used the notation $\mathbf{0} = (0, 0, ...0)$. There are several norms that are typically used. Below is a list of the three most common.

1. (p-norm) $|\mathbf{x}|_p = \left(\sum_{i=1}^{n} |x_i|^p\right)^{1/p}$
2. (ℓ_2 norm) $|\mathbf{x}|_2 = \sqrt{x_1^2 + x_2^2 + \ldots + x_n^2}$
3. $|\mathbf{x}|_\infty = \max_{i\in\{1,\ldots,n\}} |x_i|$

It is interesting to note that the norm number 3 is just the limit of the p-norm for $p \longrightarrow \infty$. The most used and known norm is the ℓ_2 norm, also known as the euclidean norm.

Info ★ Proof that $|\mathbf{x}|_\infty = \lim_{p\to\infty} |\mathbf{x}|_p$

Theorem 3.1 $\lim_{p\to\infty} |\mathbf{x}|_p = \max_{i\in\{1,\ldots,n\}} |x_i|$

Proof Let us assume that $|x_j|$ is the largest of all the absolute values of the components of $\mathbf{x}$. We can re-write $|\mathbf{x}|_p$ as

$$|\mathbf{x}|_p = \left[|x_j|^p \left(\sum_{i=1,j\neq j}^{n} \left|\frac{x_i}{x_j}\right|^p + 1\right)\right]^{1/p} = |x_j| \left[\sum_{i=1,j\neq j}^{n} \left|\frac{x_i}{x_j}\right|^p + 1\right]^{1/p} \tag{3.8}$$

now note that since $|x_j|$ is the largest of the absolute value of all the components we have

$$\left|\frac{x_i}{x_j}\right| < 1 \quad \forall i \neq j \tag{3.9}$$

and therefore we have

$$\lim_{p\to\infty} \left|\frac{x_i}{x_j}\right|^p = 0 \quad \forall i \neq j \tag{3.10}$$

thus by taking the limit of Equation 3.8 for $p \longrightarrow \infty$ and by using Equation 3.10 we get

$$\lim_{p\to\infty} |\mathbf{x}|_p = |x_j| \lim_{p\to\infty} \left[\sum_{i=1,j\neq j}^{n} \left|\frac{x_i}{x_j}\right|^p + 1\right]^{1/p} = |x_j| \tag{3.11}$$

and that concludes our proof. Just remember that $|x_j|$ is the largest component in absolute value of $\mathbf{x}$. □

If you remember the geometrical interpretation of vectors, it is easy to see how the ℓ_2 norm is the length of the vector in an n-dimensional Euclidean space.

Info ★ Norms in Deep Learning

Norms are used extensively in Deep Learning. For example ℓ_1 and ℓ_2 norms are used in regularisation (more on that in Part II) and many loss functions can be written as some norm of some quantities. For example indicating with y_i some target variable that we want to predict, and with $\hat{y}_i$ the prediction of some model, the Mean Squared Error (MSE) is given by the equation (for $i = 1, ..., M$)

$$\text{MSE} = \frac{1}{M}\sum_{i=1}^{M}(y_i - \hat{y}_i)^2 \tag{3.12}$$

this can be written with the ℓ_2 norm as

$$\text{MSE} = \frac{1}{M}|\mathbf{y} - \hat{\mathbf{y}}|_2 \tag{3.13}$$

where we have indicated $\mathbf{y} = (y_1, ...y_M)$ and $\hat{\mathbf{y}} = (\hat{y}_1, ..., \hat{y}_M)$.

3.2.3 Dot Product

Multypling vectors is not as easily defined as summing them. What should the result be from an hypothetical multiplication? A vector? Or maybe a single real (or complex) number? In algebra there are two multiplication operations that give respectively a single number and a vector as results.

The first operation we will discuss is called the *dot*-product. The dot product between two vectors $\mathbf{x}$ and $\mathbf{y}$ is indicated with $\mathbf{x} \cdot \mathbf{y}$ and is defined by

$$\mathbf{x} \cdot \mathbf{y} = \sum_{i=1}^{n} x_i y_i \tag{3.14}$$

and again note that this operation is only possible between vectors of the same length. The results of the dot-product is a single real number[2].

Info Multiplication of Vectors in `numpy`

In Python you can easily multiply numbers with `numpy`. But you have to be careful. The `multiply()` function multiply two arrays element-wise, but the output will not be a single number, but an array. If consider two arrays $vecx = (x_1, x_2, ..., x_n)$ and $\mathbf{y} = (y_1, y_2, ..., y_n)$, the result of `multiply(`$\mathbf{x}, \mathbf{y}$`)` will be an array with components

[2] If the vectors have complex component, the dot-product will be a complex number naturally.

$$\texttt{multiply}(\mathbf{x}, \mathbf{y}) = (x_1y_1, x_2y_2, ..., x_ny_n) \tag{3.15}$$

If you want the dot product as we describe you have to use the `dot()` function in `numpy`. Just keep this in mind and be aware of the difference.

It can be shown that the dot product can be expressed in terms of the ℓ_2 norm of the vectors and the angle between them

$$\mathbf{x} \cdot \mathbf{y} = |\mathbf{x}|_2 |\mathbf{y}|_2 \cos\theta \tag{3.16}$$

So $\mathbf{x} \cdot \mathbf{y}$ is nothing else than the length of one vector multiplied by the length of the projected second vector along the first. To prove this, one can perform, with the help

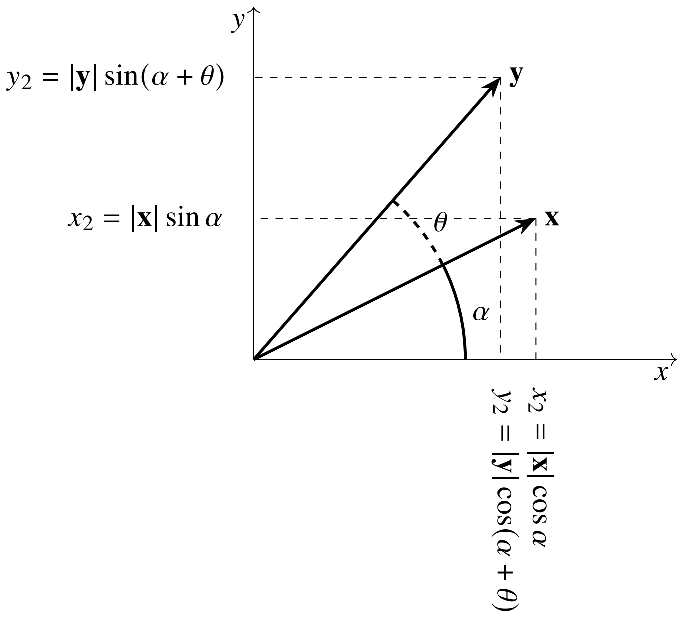
y
$y_2 = |\mathbf{y}|\sin(\alpha+\theta)$
$\mathbf{y}$
$x_2 = |\mathbf{x}|\sin\alpha$
θ
$\mathbf{x}$
α
x
$x_2 = |\mathbf{x}|\cos\alpha$
$y_2 = |\mathbf{y}|\cos(\alpha+\theta)$

Fig. 3.4 The components along two arbitrary axis can be expressed in terms of the ℓ_2 norm of the vectors multiplied by the cosine and sine of the respective angles as depicted in the figure.

of Figure 3.4, the following calculation

$$\begin{aligned} \mathbf{x} \cdot \mathbf{y} &= x_1y_1 + x_2y_2 = |\mathbf{x}|_2 \cos\alpha |\mathbf{y}|_2 \cos(\alpha+\theta) + |\mathbf{x}|_2 \sin\alpha |\mathbf{y}|_2 \sin(\alpha+\theta) = \\ &= |\mathbf{x}|_2 |\mathbf{y}|_2 (\cos\alpha\cos(\alpha+\theta) + \sin\alpha\sin(\alpha+\theta)) = \\ &= |\mathbf{x}|_2 |\mathbf{y}|_2 \cos\theta \end{aligned} \tag{3.17}$$

where in the last passage we have used the formula

$$\cos(\alpha - \theta) = \cos\alpha\cos\theta + \sin\alpha\sin\theta \tag{3.18}$$

Of course the proof would be slightly easier if one would choose the x axis along the **x** vector, thus having $\alpha = 0$. But the proof given is general and is worth understanding for the sake of generality.

3.2.4 ★ Cross Product

In Physics and Mathematics there is another operation (called the cross product) that is often used. We report it for completeness but it is never used in Machine Learning so feel free to skip this section if you are not interested. The cross product between two vectors **x** and **y** is indicated with the symbol $\times$ and is defined by

$$\mathbf{x} \times \mathbf{y} = |x|_2|y|_2 \sin(\theta)\mathbf{n} \tag{3.19}$$

where **n** is the unit vector (with ℓ_2 norm equal to one) perpendicular to the plane identified by **x** and **y** in the direction given by the right-hand rule depicted in Figure 3.5. If the vectors **x** and **y** are parallel the cross product is the zero vector 0. We can

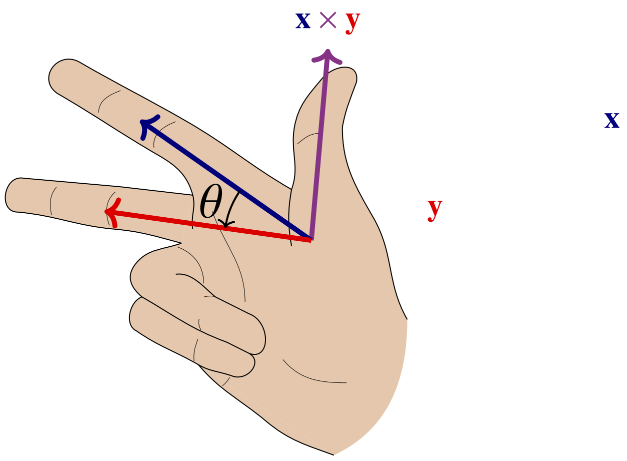

Fig. 3.5 The right-hand rule used to determined the direction of the vector $\mathbf{x} \times \mathbf{y}$.

express the cross product in terms of the components x_i and y_i **x** and **y** along the x, y and z axis. Indicating with **i**, **j** and **z** the unit vectors along the x, y and z axis respectively[3] it is easy to see that

[3] Assuming that the three axis x, y and z are normal to each other.

$$\begin{cases} \mathbf{i} \times \mathbf{j} = \mathbf{k} \\ \mathbf{j} \times \mathbf{k} = \mathbf{i} \\ \mathbf{k} \times \mathbf{i} = \mathbf{j} \end{cases} \tag{3.20}$$

Note that by definition

$$\begin{cases} \mathbf{x} = x_1\mathbf{i} + x_2\mathbf{j} + x_3\mathbf{k} \\ \mathbf{y} = y_1\mathbf{i} + y_2\mathbf{j} + y_3\mathbf{k} \end{cases} \tag{3.21}$$

therefore we can calculate

$$\begin{aligned} \mathbf{x} \times \mathbf{y} =& (x_1\mathbf{i} + x_2\mathbf{j} + x_3\mathbf{k}) \times (y_1\mathbf{i} + y_2\mathbf{j} + y_3\mathbf{k}) = \\ & x_1y_1(\mathbf{i} \times \mathbf{i}) + x_1y_2(\mathbf{i} \times \mathbf{j}) + x_1y_3(\mathbf{i} \times \mathbf{k}) + \\ & x_2y_1(\mathbf{j} \times \mathbf{i}) + x_2y_2(\mathbf{j} \times \mathbf{j}) + x_2y_3(\mathbf{j} \times \mathbf{k}) + \\ & x_3y_1(\mathbf{k} \times \mathbf{i}) + x_3y_2(\mathbf{k} \times \mathbf{j}) + x_3y_3(\mathbf{k} \times \mathbf{k}) = \\ & (x_2y_3 - x_3y_2)\mathbf{i} + (x_3y_1 - x_1y_3)\mathbf{j} + (x_1y_2 - y_2x_1)\mathbf{k} \end{aligned} \tag{3.22}$$

where we have used Equations (3.20) and the fact that the cross product between parallel vectors is zero (e.g. $\mathbf{i} \times \mathbf{i} = \mathbf{0}$).

Info Division of vectors

Note that it is not possible to define the operation of division between vectors in an unique way. First of all the division of two vectors should be a vector (one says that the vector space should be closed with respect to the division operation). When we say "division," we really mean the inverse operation of multiplication, so that $\mathbf{x}/\mathbf{y} = \mathbf{c}$ just means that $\mathbf{c}$ is the **unique** vector with the property $\mathbf{y}\mathbf{c} = \mathbf{x}$. With this idea, we could think that we could use the cross-product just defined to also define the division. But consider the following two examples

$$\begin{aligned} (1,0,0) \times (0,1,0) = (1,0,0) \\ (1,0,0) \times (1,1,0) = (1,0,0) \end{aligned} \tag{3.23}$$

As you can see the two vectors $(0,1,0)$ and $(1,1,0)$ have both the property of giving the same result when multiplied by $(1,0,0)$. Therefore there is not a unique vector $\mathbf{y}$ with the property that $(1,0,0) \times \mathbf{y} = (1,0,0)$ and therefore we cannot use the cross product to define the division.

Warning Division of vectors in `numpy`

When developing machine learning models in Python you will need to use the `numpy` library, as it is the *de facto* standard in numerical programming. If you have two numpy arrays `a` and `b` (of the same size) you can divide them by writing `a/b` or `numpy.divide(a,b)`. `numpy` will return an array of the same size as `a` and `b` with components that are the ratios of the single components of `a` and `b`. Indicating the components of `a` and `b` with a_i and b_i respectively, the resulting vector from the numpy division will have compnents equal to a_i/b_i. This is helpful for normalising arrays or matrices and is used often in deep learning for various tasks. Just keep in mind that this is just a practical workaround to do things efficiently, but has nothing to do with an hypothetical division operation between vectors.

3.3 Matrices

Matrices are, in their simplest form, 2-dimensional arrays of numbers. In other words is a set of numbers arranged in rows and columns that form a rectangular array. An example of 2×3 matrix is

$$\begin{pmatrix} 2 & 3 & 5 \\ 3 & 8 & 11 \end{pmatrix} \tag{3.24}$$

A generic $m \times n$ matrix A can be written as

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{pmatrix} \tag{3.25}$$

where $a_{i,j}$ is the generic element at row i and columns j. In this book we will indicate matrices with uppercase letters. In this book, and in machine learning in general, only matrices with real numbers as elements will be considered.

Info Tensors

In Machine Learning you will deal often with arrays that have more than two dimensions. For example a dataset composed of M coloured images will have 4 dimensions: two will indicate the horizontal and vertical resolution, one the colour channel (if you have RGB images for example you will need 3 numbers to identify the colour of each pixel) and one the index of the image. In general terms an array of numbers that have more than 2 dimensions is called a **Tensor** and its elements are indicated, similarly to matrices, with subscripts. For example a Tensor T with three dimensions will have elements indicated with $T_{i,j,k}$ where $i, j, k \in \mathbb{N}$.

One property of matrices that is important is their *shape*, or in other words how many elements the matrix has along each of its dimensions. For example a matrix with 10 rows and 25 columns will have a shape given by $(10, 25)$. Typically the shape of an array is indicated as a tuple of numbers. A generic matrix with m rows and n columns will have a shape (m, n).

3.3.1 Sum, Subtraction and Transpose

The most basic operation you can perform on a matrix is the **transpose**. The operation simply mirror the matrix with respect to the diagonal. In more mathematical terms the transpose of A is indicated with $A^\top$ and has elements given by

$$(A^\top)_{i,j} = A_{j,i} \tag{3.26}$$

Matrices can be added and subtracted, but only if they have the same shape. The operation happen element-wise. So for example if we have two generic matrices X and Y, then the elements of $B = X + Y$ and $C = X - Y$ will be given by

$$\begin{cases} B_{i,j} & = X_{i,j} + Y_{i,j} \\ C_{i,j} & = X_{i,j} - Y_{i,j} \end{cases} \tag{3.27}$$

Thanks to Equation (3.27) is easy to see that sum and subtraction of matrices satisfy the commutative and associative properties[4].

3.3.2 Multiplication of Matrices and Vectors

Probably the most important operation that you can do with matrices (and the one most used in Machine Learning) is the multiplication. Note that the multiplication is only possible if the the shapes of the two matrices you want to multiply satisfy a specific relationship. Suppose we consider two matrices X with a shape of (n, m) and Y with a shape (p, q). The multiplication $C = XY$ is defined by

$$C_{i,j} = \sum_{k=1}^{m} X_{i,k} Y_{k,j} \tag{3.28}$$

and is only possible if $m = p$. For example to calculate the element $C_{2,2}$ you will need to use the second row of the matrix X (you will need all elements $X_{2,j}$) and the second column of matrix Y (all elements $Y_{j,2}$). In Figure 3.6 you can see a graphical

[4] The commutative property says that changing the order of the elements you are summing does not change the result, or in other words $a + b = b + a$. The associative property says that if you are summing, for example, three elements a, b and c that $(a + b) + c = a + (b + c)$.

representation of the multiplication process between the two matrices X (with shape (n, m)) and Y (with shape (m, q)). To obtain the element $C_{2,2}$ one has to multiply the 2nd row elements from X (marked in blue in Figure 3.6) **element-wise** with the second column elements of Y (marked in orange in Figure 3.6) as explained by the red marked elements in Figure 3.6 and then sum the results of the element-wise multiplications.

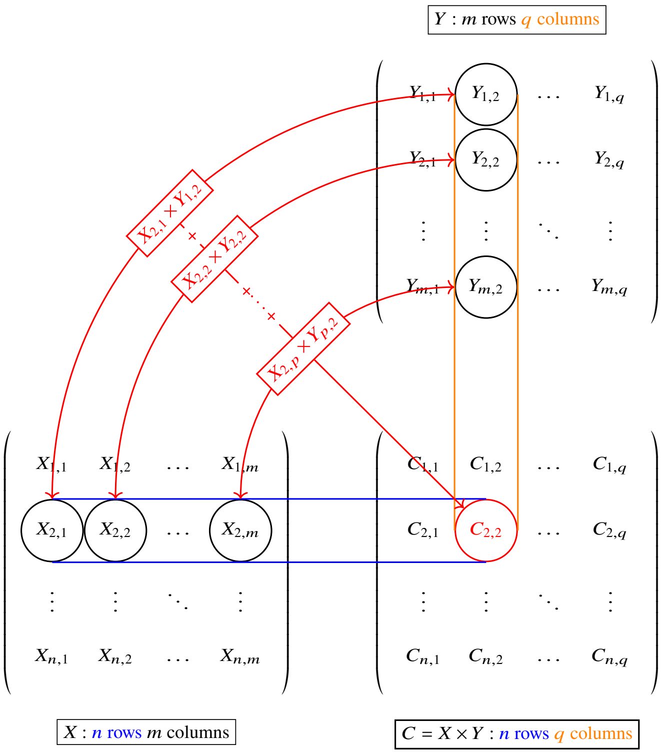
Y : m rows q columns
$Y_{1,1}$ $Y_{1,2}$ … $Y_{1,q}$
$Y_{2,1}$ $Y_{2,2}$ … $Y_{2,q}$
$Y_{m,1}$ $Y_{m,2}$ … $Y_{m,q}$
$X_{2,1} \times Y_{1,2}$
+
$X_{2,2} \times Y_{2,2}$
+ ⋯ +
$X_{2,p} \times Y_{p,2}$
$X_{1,1}$ $X_{1,2}$ … $X_{1,m}$
$X_{2,1}$ $X_{2,2}$ … $X_{2,m}$
$X_{n,1}$ $X_{n,2}$ … $X_{n,m}$
$C_{1,1}$ $C_{1,2}$ … $C_{1,q}$
$C_{2,1}$ $C_{2,2}$ … $C_{2,q}$
$C_{n,1}$ $C_{n,2}$ … $C_{n,q}$
X : n rows m columns
$C = X \times Y$: n rows q columns

Fig. 3.6 A graphical representation of the multiplication of two matrices X and Y. The matrix C is given by $C = XY$. To obtain the element $C_{2,2}$ one has to multiply the 2nd row elements from X (marked in blue) **element-wise** with the second column elements of Y (marked in orange) as explained by the red marked elements and then sum the results of the element-wise multiplications.

Multiplication is distributive

$$X(Y + Z) = XY + XZ \tag{3.29}$$

and associative

$$X(YZ) = (XY)Z \tag{3.30}$$

It is important to note that matrix multiplication is **not** commutative. So in general

$$XY \neq YZ \tag{3.31}$$

The transpose of the product of two matrices XY can be calculated with the formula

$$(XY)^\top = Y^\top X^\top \tag{3.32}$$

Matrices can be multiplied with vectors of course. We can use Equation (3.28) by writing a vector $\mathbf{y} = (y_1, ..., y_m)$ as a matrix with 1 column and 1 row (shape $(m, 1)$) (or intuitively in vertical notation)

$$\mathbf{y} = \begin{pmatrix} y_1 \\ \vdots \\ y_m \end{pmatrix} \tag{3.33}$$

by doing that, if we have a matrix A with a shape of (n, m) we can easily calculate the product by using Equation (3.28) obtaining

$$A\mathbf{y} = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,m} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,m} \end{pmatrix} \begin{pmatrix} y_1 \\ \vdots \\ y_m \end{pmatrix} = \begin{pmatrix} a_{1,1}y_1 + \cdots a_{1,m}y_m \\ a_{2,1}y_1 + \cdots a_{2,m}y_m \\ \cdots \\ a_{n,1}y_1 + \cdots a_{n,m}y_m \end{pmatrix} \tag{3.34}$$

3.3.3 Inverse and Trace

The inverse of a matrix X is indicated with X^{-1} and is the matrix that satisfy the equation

$$X^{-1}X = I \tag{3.35}$$

where I is the identify matrix of shape (n, n) as the matrix that satisfy the condition

$$\forall \mathbf{x} \in \mathbb{R}^n \; I\mathbf{x} = x \tag{3.36}$$

This is a convoluted way of defininig a matrix that has all the diagonal elements equal to 1 and all other equal to 0. For example a $(3, 3)$ identify matrix looks like this

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{3.37}$$

Note that a matrix X has an inverse X^{-1} only if it is **square** and all columns are **linearly independent**. We will not spend time here in discussing those conditions, but is good to know that not all matrices can be inverted.

Info System of linear equations and inverse of matrices

In general a linear set of equations

$$\begin{cases} a_{1,1}x_1 + \cdots a_{1,m}x_m = b_1 \\ a_{2,1}x_1 + \cdots a_{2,m}x_m = b_2 \\ \ldots \\ a_{m,1}x_1 + \cdots a_{m,m}x_m = b_m \end{cases} \tag{3.38}$$

can be written in what is called **matrix form** as

$$A\mathbf{x} = \mathbf{b} \tag{3.39}$$

where

$$\begin{cases} A_{i,j} & = a_{i,j} \\ (\mathbf{x})_i & = x_i \\ (\mathbf{b})_i & = b_i \end{cases} \tag{3.40}$$

By looking at Equation (3.39) it is to see that a solution **x** is simply given by

$$\mathbf{x} = A^{-1}\mathbf{b} \tag{3.41}$$

under the assumption that A^{-1} exists. Solving a system of linear equation with this approach is much faster than by using the substitution method (deriving an expression for x_1 from one of the equations and substituting it in the others, then deriving x_1 and so on). If the matrix A^{-1} does not exist, the system has no solution and viceversa (the discussion about existence of the inverse goes beyond the scope of this book). For example the system

$$\begin{cases} x_1 - x_2 = 0 \\ -x1 + x2 = 3 \end{cases} \tag{3.42}$$

has clearly no solutions (you should verify it). Analogously the matrix

$$A = \begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix} \tag{3.43}$$

is not invertible. We will see in the next section how to realise if a matrix is invertible.

The trace of a matrix X is defined only for square matrices and is indicated with $\text{Tr}(X)$. It is simply the sum of the elements on the diagonal

$$\text{Tr}(X) = \sum_{i=1}^{n} X_{i,i} \tag{3.44}$$

The trace of matrix is rarely used in applied deep learning nonetheless is good to know its main properties.

$$\begin{cases} \mathrm{Tr}(A+B) = \mathrm{Tr}(A) + \mathrm{Tr}(B) \\ \mathrm{Tr}(cA) = c\mathrm{Tr}(A); \quad c \in \mathbb{R} \\ \mathrm{Tr}(A^\top B) = \mathrm{Tr}(AB^\top) = \mathrm{Tr}(B^\top A) = \mathrm{Tr}(BA^\top) \\ \mathrm{Tr}(AB) = \mathrm{Tr}(BA) \\ \mathrm{Tr}(ABCD) = \mathrm{Tr}(BCDA) = \mathrm{Tr}(CDAB) = \mathrm{Tr}(DABC) \ \ \text{(cyclic property)} \end{cases} \tag{3.45}$$

3.3.4 ★ Determinant

For a square matrix X we can define the **determinant**, that is a function that maps matrices to scalars. It is indicated with $\det(X)$ or $|X|$. Before looking at how to calculate it and giving a formula for it, it is very educative to try to understand its meaning. Here I will provide a geometrical interpretation of it. Let us suppose we have a matrix

$$\begin{pmatrix} \text{——} \ \mathbf{x}_1^\top \ \text{——} \\ \vdots \\ \text{——} \ \mathbf{x}_n^\top \ \text{——} \end{pmatrix} \tag{3.46}$$

where $\mathbf{x}_i^\top$ indicates the i^{th} matrix row. Now consider the set S of all points formed by taking all possible linear combinations of the vectors $\mathbf{x}_i^\top$ for $i = 1, ..., n$ with coefficients $\alpha_i \in]0,1]$. This can be expressed in more mathematical form

$$S = \{p \in \mathbb{R}^n | p = \sum_{i=1}^{n} \alpha_i \mathbf{x}_i^\top \text{ where } \alpha_i \in]0,1], i = 1, ..., n\} \tag{3.47}$$

The absolute value of $\det(X)$ is the measure of the volume of the set S. We have not defined formally what volume means in this context, as this would go beyond the scope of this book, but an example in two dimensions will give the student an adequate intuitive understanding of this concept. Let us consider a generic matrix $X \in \mathbb{R}^{2\times 2}$.

$$X = \begin{pmatrix} X_{1,1} & X_{1,2} \\ X_{2,1} & X_{2,2} \end{pmatrix} \tag{3.48}$$

in this case we will have

$$\begin{cases} \mathbf{x}_1^\top & = (X_{1,1} \ \ X_{1,2}) \\ \mathbf{x}_2^\top & = (X_{2,1} \ \ X_{2,2}) \end{cases} \tag{3.49}$$

In Figure (3.7) you can see two vectors in a Cartesian space marked in blue ($\mathbf{x}_1^\top$) and in red ($\mathbf{x}_2^\top$). The Figure will help us in understanding visually the calculations below.

Note that all the formulas we will write are completely general, and do not depend on the particular vectors in Figure 3.7.

The set S that we have defined in Equation (3.47) is, for the vectors in Figure 3.7, the shaded region in Figure (3.7). Now let us calculate the area A of the shaded area in Figure (3.7) for our generic matrix X. In general A is given by the formula[5]

$$A = |\mathbf{x}_1^\top|_2 h = |\mathbf{x}_1^\top|_2 |\mathbf{x}_2^\top|_2 \sin\theta \tag{3.50}$$

where h is the segment indicated with a dashed line in Figure 3.7. We now only need to derive a formula for the angle θ. We can do this by using Equation (3.16)

$$\cos\theta = \frac{\mathbf{x}_1^\top \cdot \mathbf{x}_2^\top}{|\mathbf{x}_1^\top|_2 |\mathbf{x}_2^\top|_2} \tag{3.51}$$

Now we have all ingredients to derive a formula for A. In fact

$$\begin{aligned} A &= |\mathbf{x}_1^\top|_2 |\mathbf{x}_2^\top|_2 \sin\theta = |\mathbf{x}_1^\top|_2 |\mathbf{x}_2^\top|_2 \sqrt{1-\cos\theta^2} = \\ &= |\mathbf{x}_1^\top|_2 |\mathbf{x}_2^\top|_2 \frac{\sqrt{|\mathbf{x}_1^\top|_2^2 |\mathbf{x}_2^\top|_2^2 - (\mathbf{x}_1^\top \cdot \mathbf{x}_2^\top)^2}}{|\mathbf{x}_1^\top|_2 |\mathbf{x}_2^\top|_2} = \\ &= \sqrt{(X_{1,1}^2 + X_{1,2}^2)(X_{2,1}^2 + X_{2,2}^2) - (X_{1,1}X_{2,1} + X_{1,2}X_{2,2})^2} = \\ &= \{\text{With some work}\} = \\ &= |X_{1,1}X_{2,2} - X_{1,2}X_{2,1}| \end{aligned} \tag{3.52}$$

Note that technically speaking the square root of a square can be positive or negative, but since we are looking at the area the result must be positive. As you will see in what follows, the last line in Equation (3.52) is the absolute value of the determinant of X.

Let us now give a formal definition of the determinant for a 2×2 and a 3×3 matrix.

Definition 3.2 The determinant of a generic matrix $X \in \mathbb{R}^{2\times 2}$ is given by

$$\det(X) = \det\begin{pmatrix} X_{1,1} & X_{1,2} \\ X_{2,1} & X_{2,2} \end{pmatrix} = X_{1,1}X_{2,2} - X_{1,2}X_{2,1} \tag{3.53}$$

Definition 3.3 For a 3×3 matrix the determinant is given by this formula

[5] You should remember how to calculate the area of a parallelogram.

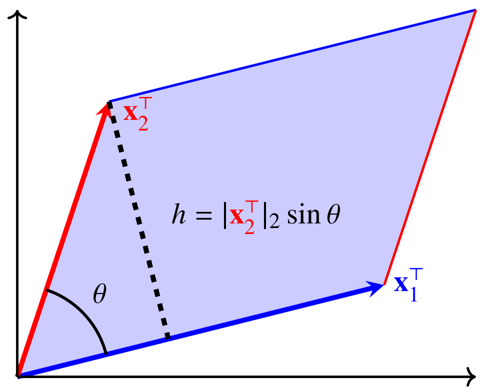

Fig. 3.7 The vectors $\mathbf{x}_1^\top$ and $\mathbf{x}_2^\top$ in a cartesian space marked in blue and in red respectively. The shaded area contains all the elements of the set S that we have defined in Equation (3.47). h indicates the length of the component of the vector $\mathbf{x}_2^\top$ along the direction perpendicular to $\mathbf{x}_1^\top$.

$$\det(X) = \det\begin{pmatrix} X_{1,1} & X_{1,2} & X_{1,3} \\ X_{2,1} & X_{2,2} & X_{2,3} \\ X_{3,1} & X_{3,2} & X_{3,3} \end{pmatrix} = X_{1,1}\det\begin{pmatrix} X_{2,2} & X_{2,3} \\ X_{3,2} & X_{3,3} \end{pmatrix} + \\ + X_{1,2}\det\begin{pmatrix} X_{2,1} & X_{2,3} \\ X_{3,1} & X_{3,3} \end{pmatrix} + X_{1,3}\det\begin{pmatrix} X_{2,1} & X_{2,2} \\ X_{3,1} & X_{3,2} \end{pmatrix} \tag{3.54}$$

the final result can then be calculated by evaluating the determinants of the 2×2 matrices. Each of those determinants is called a *minor* of the matrix X.

The determinants for matrices that larger than 3×3 is defined analogously in a recursive fashion. This is called the Laplace expansion, and the general recursive formula for the determinants of an arbitrary matrix X of shape (n, n) is given by

$$\det(X) = \sum_{j=1}^{n} (-1)^{i+j} X_{i,j} M_{i,j} \tag{3.55}$$

where $M_{i,j}$ is the determinant of the submatrix obtained by removing the i^{th} row and the j^{th} column of X. In the formula one has to choose one value for i, for example $i = 1$. Note that we report this formula just for completeness and for large matrices numerical approaches are almost always used in practice, but this expansion is often used in mathematical proofs and formulas.

Info ★ The role of the determinant in the inversion of a matrix

We have discussed briefly before that not all matrices can be inverted. In general the inverse of a matrix X^{-1} is proportional to the inverse of the determinant

$$X^{-1} \propto \frac{1}{\det(X)} \tag{3.56}$$

and therefore the inverse of a matrix exists if and only if the determinant is not zero. This is an easy way of veryfing if one matrix can be inverted. In the example we have looked at before when we tried to solve the system of linear equations (3.42) we had to invert the matrix

$$A = \begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix} \tag{3.57}$$

You may remember that we said this matrix is not invertible. We can verify this statement easily by calculating the determinant

$$\det(A) = a_{1,1}a_{2,2} - a_{1,2}a_{2,1} = (1)(1) - (-1)(-1) = 0 \tag{3.58}$$

and as expected, since the determinant is equal to zero, the matrix A^{-1} does not exist.

3.3.5 ★ Matrix Calculus and Linear Regression

Matrix calculus is the study of doing calculus with matrices. Is not obvious how to perform a derivative, for example, of a matrix or even less obvious how to perform it with respect to a matrix (or a vector for that matter). This short section intend to give the reader an overview of the most important definitions and show with an example (linear regression) how powerful these techniques are when used correctly. Let us start with a few definitions.

Definition 3.4 If we consider a matrix X of shape (n, m) that depends on some scalar variable y, we define its derivative with respect to y with

$$\frac{dX}{dy} = \begin{pmatrix} dX_{1,1}/dy & \cdots & dX_{n,1}/dy \\ \vdots & \ddots & \vdots \\ dX_{n,1}/dy & \cdots & dX_{n,m}/dy \end{pmatrix} \tag{3.59}$$

Analogously we can define the derivative of a vector $\mathbf{x}$ that depends on some scalar y.

Definition 3.5 The derivative of a vector $\mathbf{x}$ of shape $(n, 1)$ with respect to some scalar y is defined by

$$\frac{d\mathbf{x}}{dy} = \begin{pmatrix} dx_1/dy \\ \vdots \\ dx_n/dy \end{pmatrix} \tag{3.60}$$

Now let us turn our attention to the derivative of a function with respect to a matrix or a vector. Consider a function $f(X)$ and that X is a matrix of shape (n, m). The following definitions are common.

Definition 3.6 The derivative of a scalar function with respect to a matrix X and a vector $\mathbf{x}$ are given by

$$\frac{df}{dX} = \begin{pmatrix} df/dX_{1,1} & \cdots & df/dX_{n,1} \\ \vdots & \ddots & \vdots \\ df/dX_{n,1} & \cdots & df/dX_{n,m} \end{pmatrix} \tag{3.61}$$

and

$$\frac{df}{d\mathbf{x}} = (df/dx_1, \cdots, df/dx_n) \tag{3.62}$$

respectively. Note that $df/d\mathbf{x}$ is nothing else than the gradient of a function that we have already seen (we have indicated with $\nabla_{\mathbf{x}} f$ so far). The gradient of a function is one of the key and important elements when training neural networks and was discussed at length in 2.6.

All this new notation may seem an unnecessary complication, but it is extremely powerful and can be used in many practical cases. To give the student an idea of its power let us discuss the problem of linear regression with least squares. Let us consider M tuples of real numbers $(x^{(i)}, y^{(i)})$ for $i = 1, ..., M, x^{(i)} \in \mathbb{R}^n$ and $y^{(i)} \in \mathbb{R}$. Our goal is to find the best parameters θ_i for $i = 0, ..., n$ that minimise the mean square error (MSE) between the formula

$$\hat{y}^{(j)} = \sum_{i=1}^{n} \theta_i x_i^{(j)} + \theta_0 \tag{3.63}$$

and the expected value $y^{(i)}$. The MSE in this case can be written as

$$J = \frac{1}{n} \sum_{i=1}^{n} (y^{(i)} - \hat{y}^{(i)})^2 = \frac{1}{n} \sum_{i=1}^{n} \left(y^{(i)} - \sum_{j=1}^{n} \theta_i x_i^{(j)} - \theta_0 \right)^2 \tag{3.64}$$

To solve this problem we need to minimise J with respect to all θ_i. This can be done directly by using Equation (3.64) and calculating its derivatives with respect to θ_i of course, but their form will be quite complicated to handle. But luckily matrices are here to save us. Let us first rewrite Equation (3.63) in matrix form by defining first the vector $\boldsymbol{\theta} = (\theta_0, \theta_1, ..., \theta_n)$ (with shape $(n+1, 1)$). Since we have a constant term in Equation (3.63), we define the matrix

$$X = \begin{pmatrix} 1 & x_1^{(1)} & \cdots & x_n^{(1)} \\ 1 & x_1^{(2)} & \cdots & x_n^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_1^{(m)} & \cdots & x_n^{(m)} \end{pmatrix} \tag{3.65}$$

by adding a column with all 1s to the matrix obtained by putting all $x^{(i)}$ together. Note that X have a shape equal to $(m, n+1)$. With those definitions we can finally rewrite Equation (3.63) in matrix form

$$\hat{\mathbf{y}} = X\boldsymbol{\theta} \tag{3.66}$$

where $\hat{\mathbf{y}} = (\hat{y}_1, ..., \hat{y}_n)^\top$. Now let us rewrite J from Equation (3.64) with matrices

$$J = \frac{1}{m}(X\boldsymbol{\theta} - \mathbf{y})^\top (X\boldsymbol{\theta} - \mathbf{y}) \tag{3.67}$$

This equation is much more easy to handle than Equation (3.64). Let's expand it

$$\begin{aligned} J =& \frac{1}{m}(\mathbf{y} - X\boldsymbol{\theta})^\top(\mathbf{y} - X\boldsymbol{\theta}) = \frac{1}{m}(\mathbf{y}^\top - \boldsymbol{\theta}^\top X^\top)(\mathbf{y} - X\boldsymbol{\theta}) = \\ =& \frac{1}{m}\left(\boldsymbol{\theta}^\top X^\top X\boldsymbol{\theta} - \boldsymbol{\theta}^\top X^\top \mathbf{y} - \mathbf{y}^\top X\boldsymbol{\theta} + \mathbf{y}^\top\mathbf{y}\right) \end{aligned} \tag{3.68}$$

all the elements that do not contain the matrix $\boldsymbol{\theta}$ are not relevant when we will calculate $\partial J / \partial \boldsymbol{\theta}$ and therefore will be ignored in the next calculations. Now before proceeding I will cheat and give you some formulas without derivation (all the formulas can be found in [7]).

$$\frac{\partial \boldsymbol{\theta}^\top A \boldsymbol{\theta}}{\partial \boldsymbol{\theta}} = (A + A^\top)\boldsymbol{\theta} \tag{3.69}$$

$$\frac{\partial \boldsymbol{\theta}^\top \mathbf{a}}{\partial \boldsymbol{\theta}} = \mathbf{a} \tag{3.70}$$

$$\frac{\partial \mathbf{a}^\top \boldsymbol{\theta}}{\partial \boldsymbol{\theta}} = \mathbf{a} \tag{3.71}$$

by using Equations (3.69), (3.70) and (3.71) in Equation (3.68) it is possible to derive the formula

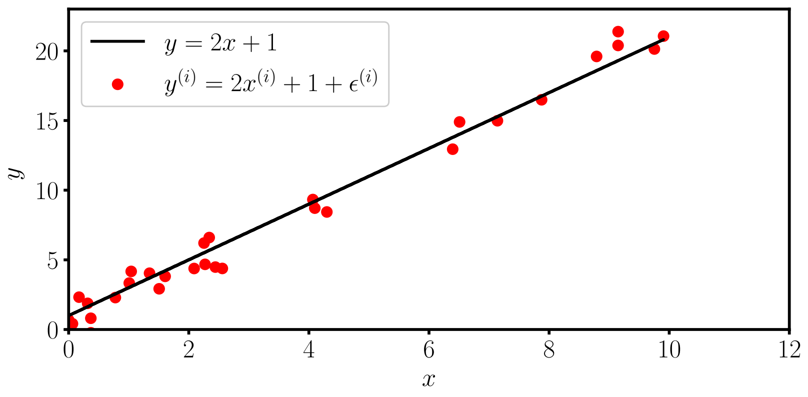

Fig. 3.8 Sample data obtained by adding noise (red points) to a linear function $2x + 1$.

$$\frac{\partial J}{\partial \boldsymbol{\theta}} = \frac{1}{m}\frac{\partial}{\partial \boldsymbol{\theta}}\left(\underbrace{\boldsymbol{\theta}^\top X^\top X\boldsymbol{\theta}}_{\text{use } 3.69} - \underbrace{\boldsymbol{\theta}^\top X^\top \mathbf{y}}_{\text{use } 3.70} - \underbrace{\mathbf{y}^\top X\boldsymbol{\theta}}_{\text{use } 3.71} + \mathbf{y}^\top \mathbf{y}\right) = \\ = \frac{2}{m}(X^\top X\boldsymbol{\theta} - X^\top \mathbf{y}) \tag{3.72}$$

Now we want to minise J therefore we have to impose

$$\frac{\partial J}{\partial \boldsymbol{\theta}} = 0 \tag{3.73}$$

with that and Equation (3.72) we finally get the result

$$\boldsymbol{\theta} = (XX^\top)^{-1}X^\top \mathbf{y} \tag{3.74}$$

Equation (3.74) gives the values of $\boldsymbol{\theta}$ that minimise the MSE.

Let us now use this formula and compare it to the numerical solution obtained by minimising the MSE. Let us generate 30 tuples $(x^{(i)}, y^{(i)})$ with $i = 1, ..., 30$ with the equation

$$y^{(i)} = 2x^{(i)} + 1 + \epsilon^{(i)} \tag{3.75}$$

by choosing $x^{(i)}$ randomly between 0 and 10 (from a uniform distribution) and where $\epsilon^{(i)}$ is chosen randomly by sampling from a normal distribution with average equal to zero and variance equal to 1. The data is shown in Figure 3.8. The red points have been obtained as explained, while the black lines has been obtained with Equation (3.75) without $\epsilon^{(i)}$. Note that if you decide to recreate this dataset, your points $y^{(i)}$ will be different due to a different random seed in your numerical routines. So don't be surprised if this happen. If we use normal numerical routines[6] to minimise the

MSE we get, for the data in Figure 3.8, the value for the slope of 2.096 and for the intercept of 0.520. Now if we use our exact formula we get the results

$$\boldsymbol{\theta} = (XX^\top)^{-1}X^\top \mathbf{y} = \begin{pmatrix} 0.520 \\ 2.096 \end{pmatrix} \tag{3.76}$$

that are exactly the same results we obtained with the numerical approach. By using matrix formalism we can simplify the notation extremely and gets formula that are much easier to handle than summation over large number of elements.

3.4 Relevance for Deep Learning

Matrices are used extensively in deep learning. The weights in FFNNs or the filters in CNNs are always matrices (typically very large) and often they are tensors since they have more than 2 dimensions. We will see in Part II in more details where matrices appear in neural networks, but they are one of the fundamental building blocks of deep learning. The input data is also always saved as a tensor. Imagine you have M images, each having a resolution of 1024×1024 in RGB (3 channels). Then the entire dataset can be saved as a huge tensor that have 4 dimensions: index of the image (M values), pixel index along the x axis (1024 values), pixel index along the y axis (1024 values), 3 values for the color in RGB (Red value, Green value and Blue value, so 3 values). The dataset tensor has 4 dimensions and shape (M, 1024, 1024, 3).

This is why the Python `numpy` library is so commonly used. It was developed exactly with the goal of performing operations on arrays of numbers in the most efficient way possible.

Info GPUs, TPUs and Matrices

GPUs (Graphical Processing Units) are very good at multiplying matrices. In 3-dimensional graphics, matrices are fundamentals since they are used to (for example) rotate objects in space. Graphic cards were built to be very fast at that: multiplying matrices. This is the reason why they are so useful in deep learning. At its very core training a deep learning model, consists in many matrix multiplication operations. The fastest you can multiply tensors, the fastest you can train a model. At the time of writing, practically the only GPUs that can be used for deep learning are NVIDIA ones. Since 2007 the NVIDIA CUDA framework provides access to GPU resources. CUDA is based on C and provides an API that can be used to access GPUs for machine learning tasks. Since 2021, apple with their M1 chip laptops, managed to make their GPUs available for deep learning with the Metal framework (for

[6] If you are interested, to obtain these results I have used Python and scikit-learn.

example with TensorFlow), but the NVIDIA cards are the ones that you will find in all servers and deep learning workstations. Note also that CUDA works in Linux and is not available for MacOs or Windows. An interesting review of methods to optimise code for GPUs can be found in [8].
An additional advantage of using GPUs is the fact that they can scale when used in parallel (meaning you can use multiple GPUs to speed up your training) and that very often comes with a large amount of memory for very large datasets. 48 Gb or more of memory for each graphic card are not uncommon in the more high end segment. Note that such cards can cost up to (and more than) 10'000 USD. Setting up a professional deep learning infrastructure can be very expensive.
If you models are large but you do not want to invest in an expensive deep learning infrastructure, you can always use cloud solutions like Google Cloud, Microsoft Azure or AWS (Amazon Web Services). They will provide a pay-by-use infrastructure that can be dynamically changed according to your needs.
TPUs (Tensor Processing Units) are special chips developed by Google that can do one thing, and one thing only: multiply matrices. They cannot run a word processor or a text editor, but they can multiply matrices at a very fast speed [9, 10]. They are available on Google Cloud and they provide quite a speed up to model training. Those are not available to buy for workstations, but are used in Google Linux Clusters.

3.5 Principal Component Analysis

If you are a researcher in any natural science, you will encounter sooner or later (much sooner than later) principal component analysis (PCA). A very well known dimensionality reduction techniques. It is a good idea to understand it well, to be able to know when you can apply it, when not and what its limitations are.

PCA is a widely used method to extract relevant and sometime hidden information in complex datasets. Very often, depending on how data collection is done (we will make an example later), we end up with a lot more variables and information than what is really relevant to the problem. PCA is callled a *dimensionality reduction* technique, because by extracting only relevant information, effectively reduce the number of dimensions of a problem ignoring the redundant and irrelevant information.

Suppose that we want to study the movement of an object attached to a spring[7]. Assuming the right initial conditions, we know from physics that the movement will be approximately (the spring may not be ideal, the surface on which the experiment is done not completely flat, the initial condition not perfect, etc.) along a line. Let us suppose that we do not know much about the problem, and we decide to study

[7] Example adapted from [11]

it with three cameras positioned randomly around the system (spring plus object). Let us also suppose that each camera measures the position of the object at regular intervals. We know already that three cameras are too much, and one would be more than enough. But by setting up our measurement system redundantly, we generate data with a much higher dimensionality than necessary. Very often, as experimenter, we often have no idea about which measurement setup reflect in the best possible way how a specific system works.

The question we are trying to address is *can we remove the redundant information and extract the really physical relevant data from our measurements*?

An additional complications that make our life more difficult is that in real life, systems and measurement systems are imperfect. This means that in our example, we will not measure a perfect line of points. As mentioned the spring may be imperfect, the object not completely symmetric, friction on the surface may be irregular and make the spring slow down with time and so on. This means that we will not observe a perfect line of points, but a cloud (more or less spread) depending on a variety of factors that we cannot always control.

3.5.1 Change of Basis

Let us first discuss briefly some necessary mathematical formalism. In vectorial notation, a **basis** is describe by a set of unit vectors. If the axis are orthonormal the vectors could be, for example, $(1, 0)$, $(0, 1)$ in two dimensions. In general a basis can be expressed by putting each of the unit vectors as row vectors in a matrix, getting the identity matrix.

$$I = \begin{pmatrix} 1 & & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{pmatrix} \tag{3.77}$$

How can we express a change of basis in matrix formalism? Note that we will consider here only a *linear* basis transformation by considering only a linear combination of the existing unit vectors that describe our original basis.

This is highly relevant to our discussion since **PCA is nothing does nothing else than expressing the original data on a new basis that has been obtained by a linear combination of the original basis vectors.**

In general, to make a *linear* change of basis, it suffices to multiply our identity matrix (our original basis) by a **transformation matrix** T. Let us give an example, to make the discussion more concrete. Let us consider the Euclidean space $\mathbb{R}^2$ with basis vectors $v_1 = (1, 0)$ and $v_2 = (0, 1)$. Now let us suppose that we want to rotate the axis of an angle α. It is easy to see, with some trigonometry, that the new vectors u_1 and u_2, obtained by rotating v_1 and v_2, are

$$u_1 = (\cos\alpha, \sin\alpha) \tag{3.78}$$
$$v_1 = (-\sin\alpha, \cos\alpha) \tag{3.79}$$

The formula can be easily understood by looking at Figure 3.9. It is then easy to

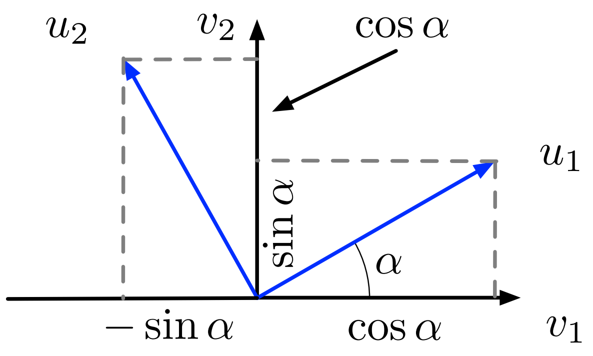

Fig. 3.9 A change of basis from (v_2, v_2) to (u_1, u_2) obtained by rotating the axis by an angle α.

express the transformation as a matrix operation. Let us define the basis as a matrix (with each vector v_1 and v_2 as column vectors).

$$V = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \tag{3.80}$$

we can define the transformation matrix T as

$$T = \begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix} \tag{3.81}$$

Then the new basis matrix U (where we have indicated with $u_{1,x}$ the component of the vector u_1 along v_1, etc.) with the new basis vectors as columns.

$$U = \begin{pmatrix} u_{1,x} & u_{2,x} \\ u_{1,y} & u_{2,y} \end{pmatrix} \tag{3.82}$$

can be calculated with

$$U = TV \tag{3.83}$$

It is easy to check that this is correct. Let's now consider a point $P = p_1 v_1 + p_2 v_2$ (we will consider the basis vectors in matrix formalism, as column vectors), that can be written in vector form as

$$P = p_1 v_1 + p_2 v_2 = p_1 \begin{pmatrix} 1 \\ 0 \end{pmatrix} + p_2 \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} p_1 \\ p_2 \end{pmatrix} \tag{3.84}$$

How can we express P in the new basis (that we will indicate with P_V)? This can now be done easily using the transformation matrix T. In fact we have

$$P_V = TP = \begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix} \begin{pmatrix} p_1 \\ p_2 \end{pmatrix} = \begin{pmatrix} p_1 \cos\alpha - p_2 \sin\alpha \\ p_1 \sin\alpha + p_2 \cos\alpha \end{pmatrix} \tag{3.85}$$

This can be easily verified by drawing a diagram similar to Figure 3.9. In general, given a dataset X where each column is a single sample of our data, when we have a linear transformation T, the new representation Y in the new basis will be given by

$$Y = TX \tag{3.86}$$

that is the natural expansion of Equation 3.85. In fact in Equation 3.85 we just calculated the new coordinates of a single point, in Equation 3.86 we changed the coordinates of **all** points at the same time (since each is a column in the matrix X).

Equation 3.85 represents a change of basis. In general T is a rotation and a stretch of the space. The rows of T represents the new basis vectors. Note that the matrix T cam have a higher dimension that 2×2 as in our example. In fact X is our datasets, and that means that the vertical dimension is given by the number of features that we have and that can be very high. This can be easily seen by writing the equation in the following form (assuming the matrix T has m rows and our dataset has n data sampels, or number of columns).

$$TX = \begin{pmatrix} — \boldsymbol{t}_1 — \\ \vdots \\ — \boldsymbol{t}_m — \end{pmatrix} \begin{pmatrix} | & & | \\ \boldsymbol{x_1} & \cdots & \boldsymbol{x_n} \\ | & & | \end{pmatrix} = \begin{pmatrix} \boldsymbol{t}_1 \cdot \boldsymbol{x}_1 & \cdots & \boldsymbol{t}_1 \cdot \boldsymbol{x}_n \\ \vdots & \ddots & \vdots \\ \boldsymbol{t}_m \cdot \boldsymbol{x}_1 & \cdots & \boldsymbol{t}_m \cdot \boldsymbol{x}_n \end{pmatrix} \tag{3.87}$$

And as you can see, the first row of the matrix TX will be the project of the the dataset along the vector $\boldsymbol{t}_1$, the second along $\boldsymbol{t}_2$ and so on.

3.5.2 Digression on Linear Transformations

Let us define what is a linear transformation.

Definition 3.7 A linear transformation is a function $T : \mathbb{R}^n \longrightarrow \mathbb{R}^m$ that satisfies the following two properties:

1. $T(x + y) = T(x) + T(y)$ for all $x, y \in \mathbb{R}^n$
2. $T(cx) = cT(x)$ for all $x \in \mathbb{R}^m$ and $c \in \mathbb{R}$.

It can be shown (it is quite easy) that if $T : \mathbb{R}^n \longrightarrow \mathbb{R}^m$ is a linear transformation, then the function T is just a matrix-vector multiplication: $T(x) = Ax$ for some matrix A.

In general, any transformation is described, in geometrix terms, by some property as rotation, a stretch or some other gemotrical modification [12]. In Table 3.1 a list of transformations [12] in two dimensions is reported.

Geometrical Transformation	**Transformation Matrix**
Rotation of an angle α	$\begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix}$
Reflection through the x-axis	$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$
Reflection through the y-axis	$\begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix}$
Reflection through the origin	$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$

Table 3.1 List of transformation matrices for specific geometrical modifications [12].

3.6 PCA

Now that we have the formalism out of the way, let us go back to the original question. With this formalism we still have to answer two main questions.

- What is the best way to re-write our dataset X? Or in other words, how can we decide what is a good and what is a bad representation?
- What is then a good choice for T?

PCA is based on one important assumption: **the directions along which the data show the largest variance are the ones that contain the most relevant and interesting information**.

This can be understood by thinking that, if along a certain direction there is a very small variance, that means that along that direction the data does not change and therefore cannot contain much interesting information. In our example of the spring, the direction perpendicular to the spring is not relevant for the description of the phenomena, and movement of the objects attached to the spring will be minimal (or not existing at all) along that direction (so the data will have a very low variance along the direction perpendicular to the spring). So our goal would be to try to find a new basis along whose (at least some of) directions the variance is maximized.

3.6.1 Redundancy

Let us spend some words about one aspect of having more features that are really needed. This is called *redundancy*. For example, let us consider our spring example. Suppose, as explained, that we measure with three cameras the position of the objects in a plane at regular interval. That means that at each time point we will have 6 values (the coordinates of the object as seen from each camera). Now we know that the motion is uni-dimensional (when ignoring noise, imperfections, etc.), thus 5 values are useless, or *redundant*. The idea about PCA, and in general dimensionality reduction, is that it is able to identify redundancies and identify relevant (new) features.

3.6.2 Covariance Matrix

But how to measure when between two variables (for example) one is redundant? Well, it is simply a matter of checking if they are correlated or not. For example let us suppose that we have a data set given by two features $D = \{x_i, x_i\}_{i=1}^N$. It is evident that we do not need two features to describe D, one would certainly suffice.

To explain this let us consider two set of measurements: $X = \{x_i\}_{i=1}^N$ and $Y = \{y_i\}_{i=1}^N$. To simplify the equations, let us consider that they both have zero mean. In this case the *covariance* of X and Y, given by the equation

$$\sigma_{XY}^2 = \frac{1}{N}\sum_{i=1}^{N} x_i y_i \tag{3.88}$$

measures the degree of relationship between the two variables. A large positive value indicates positively correlated data (if one grows, so does the other). A negative value indicates negatively correlated data (if one grows, the other decreases). By writing X and Y as matrices $\boldsymbol{X}$ and $\boldsymbol{Y}$ with dimensions $(1, N)$ we can re-write the covariance as a matrix product

$$\sigma_{XY}^2 = \frac{1}{N}\boldsymbol{X}\boldsymbol{Y}^T \tag{3.89}$$

In general if we have a large dataset we can generalize the definition. Let us consider a dataset $\boldsymbol{X}$

$$\boldsymbol{X} = \begin{pmatrix} \text{—}\, \boldsymbol{x}_1 \,\text{—} \\ \vdots \\ \text{—}\, \boldsymbol{x}_m \,\text{—} \end{pmatrix} \tag{3.90}$$

Now in this case each row contains a complete measurement (all the fatures), while each column all measurements of a specific feature[8]. In this case we can write the

[8] This is done in the opposite way as before, since in this case we are interested in the covariance between features and not measurements.

covariance matrix $\boldsymbol{C}$ as

$$\boldsymbol{C} = \frac{1}{N}\boldsymbol{X}\boldsymbol{X}^T \tag{3.91}$$

Note that $\boldsymbol{C}$ is a square matrix, its diagonal terms are the variances of particular features and the off-diagonal terms the covariance between different features.

Remember our hypothesis, that what is interesting happens along directions that have large variance and small co-variance (low redundancy) we can make the following statements:

1. Large diagonal elements indicate interesting features.
2. large off-diagonal terms indicate large redundancy (so we do not want that, remember?).

You may, at this point, see where we are going with this. Our goal is to **find a new basis (diagonalise the co-variance matrix) to (1) maximise the variance (so the diagonal elements) and (2) minimise redundancy (the off-diagonal elements)**. This is achieved by simply diagonalising the co-variance matrix. This all PCA does really.

To make PCA easy to use and calculate PCA assumes that the new basis will be an **orthonormal matrix**, or in other words that the vectors of the new vasis are or orthogonal to each other.

3.6.2.1 Overview of Assumptions

Let us review all the assumptions we have made in this paper to use PCA.

1. *Linearity*: the change of basis we are searching for is a linear transformation (there are other approaches that lift this assumption, like t-SNE (t-distributed Stochastic Neighbour Embedding) [13]).
2. *Large variance is important*: this assumption is often safe to make, but it is a strong assumption that is not always correct. For example, if you are studying the transverse oscillations of a spring due to imperfections in your system, PCA may simply ignore those effects since the variance is minimal along the transverse direction.
3. *The new basis is orthogonal*: this makes using PCA fast and easy, but it does not work every time. It may well be that there are directions where the interesting phenomena occur that are not orthogonal to each other.

3.6.3 PCA with Eigenvectors and Eigenvalues

The problem that PCA solves can be stated as follows.

Problem 3.1 (PCA Problem Statement) Find an orthonormal matrix $\boldsymbol{T}$, such that, given a dataset $\boldsymbol{X}$ and $\boldsymbol{Y} = \boldsymbol{T}\boldsymbol{X}$ the matrix $\boldsymbol{C} = (1/N)\boldsymbol{Y}\boldsymbol{Y}^T$ is a diagonal matrix.

Let us start by writing $\boldsymbol{C}$.

$$
\begin{aligned}
\boldsymbol{C} &= \frac{1}{N}\boldsymbol{Y}\boldsymbol{Y}^T \\
&= \frac{1}{N}(\boldsymbol{T}\boldsymbol{X})(\boldsymbol{T}\boldsymbol{X})^T \\
&= \frac{1}{N}\boldsymbol{T}\boldsymbol{X}\boldsymbol{X}^{\boldsymbol{T}}\boldsymbol{T}\boldsymbol{T} \\
&= \boldsymbol{T}\left(\frac{1}{N}\boldsymbol{X}\boldsymbol{X}^T\right)\boldsymbol{T}^T \\
&= \boldsymbol{T}\boldsymbol{C}_{\boldsymbol{X}}\boldsymbol{T}^T
\end{aligned}
\tag{3.92}
$$

where with $\boldsymbol{C}_{\boldsymbol{X}}$ we have indicated the covariance matrix of $\boldsymbol{X}$. Now you should know that any symmetric matrix $\mathbf{M}$ is diagonalized by a matrix composed of its eigenvectors organized as columns. The trick now is to choose $\boldsymbol{T}$ to be a matrix where each row is an eigenvector of $\left(\frac{1}{N}\boldsymbol{X}\boldsymbol{X}^T\right)$. Let us indicate this matrix with $\boldsymbol{E}^T$, and let us indicated with $\boldsymbol{D}$ the diagonalised version of $\left(\frac{1}{N}\boldsymbol{X}\boldsymbol{X}^T\right)$.

$$
\begin{aligned}
\boldsymbol{C} &= \boldsymbol{T}\boldsymbol{C}_{\boldsymbol{X}}\boldsymbol{T}^T \\
&= \boldsymbol{T}(\boldsymbol{E}\boldsymbol{D}\boldsymbol{E}^T)\boldsymbol{T}^T \\
&= \{\text{Remember } \boldsymbol{E}^T = \boldsymbol{T}\} \\
&= (\boldsymbol{T}\boldsymbol{T}^T)\boldsymbol{D}(\boldsymbol{T}\boldsymbol{T}^T) \\
&= \{\text{Since the matrix } \boldsymbol{T} \text{ is orthonormal then } \boldsymbol{T}^T = \boldsymbol{T}^{-1}\} \\
&= (\boldsymbol{T}\boldsymbol{T}^{-1})\boldsymbol{D}(\boldsymbol{T}\boldsymbol{T}^{-1}) \\
&= \boldsymbol{D}
\end{aligned}
\tag{3.93}
$$

So this choice of $\boldsymbol{T}$ diagonalizes $\boldsymbol{C}$.

3.6.3.1 One Implementation Limitation

All we discussed sound good, but one limitation of PCA is that when you want to diagonalise numerically the matrix $\left(\frac{1}{N}\boldsymbol{X}\boldsymbol{X}^T\right)$, this has to fit completely in memory. With large dataset, this may be a problem (often is). This is why you may encunter difficulties in doing this. One variant of PCA that you may consider, is IPCA (Incremental Principal Component), that goes beyond the scope of this paper, but that is available in the Python library scikt-learn [14].

3.6.4 Summary

This short paper describes briefly what PCA is, what its assumptions are, and shows one analytical approach to solve Problem 3.1. The reader should be aware that there are different ways of solving Problem 3.1, the most notable one being single value decomposition (and that is how it is solved in the Python implementation of scikit-learn [15].

Exercises

3.1 Write a Python program to multiply two arrays each having 10^7 numbers. implement it first as a loop and by using the `numpy.multiply()` function. Measure how much time each of the two methods takes.

3.2 Given three matrices X, Y and Z, prove that the following property is valid

$$X(Y+Z) = XY + YZ$$

3.3 Given three matrices X, Y and Z, prove that the following property is valid

$$X(YZ) = (XY)Z$$

3.4 Given two matrices X, Y prove that the transpose of the product is given by

$$(XY)^\top = Y^\top X^\top$$

3.5 Prove the properties of the trace in Equation (3.45).

3.6 If you repeat the linear regression as explained at the end of Section 3.3.5 but you use $x^{(i)} = i/3$ what will happen when you are trying to use the exact solution? Can you explain where is the problem?

3.7 Write a Python function that can multiply two matrices X and Y. Then check if the `numpy` version is faster, slower or as fast as your function.

References

1. Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

2. Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.
3. Lloyd N Trefethen and David Bau III. *Numerical linear algebra*, volume 50. Siam, 1997.
4. Peter D. Lax. *Linear Algebra and Its Applications*. Wiley-Interscience, Hoboken, NJ, second edition, 2007.
5. Harry Dym. *Linear algebra in action*, volume 78. American Mathematical Soc., 2013.
6. Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.
7. Kaare Brandt Petersen, Michael Syskind Pedersen, et al. The matrix cookbook. *Technical University of Denmark*, 7(15):510, 2008.
8. Sparsh Mittal and Shraiysh Vaishay. A survey of techniques for optimizing deep learning on gpus. *Journal of Systems Architecture*, 99:101635, 2019.
9. Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pages 1–12, 2017.
10. Amna Shahid and Malaika Mushtaq. A survey comparing specialized hardware and evolution in tpus for neural networks. In *2020 IEEE 23rd International Multitopic Conference (INMIC)*, pages 1–6. IEEE, 2020.
11. Lindsay I Smith. A tutorial on Principal Components Analysis.
12. John Gilbert. Linear Transformations. https://web.ma.utexas.edu/users/gilbert/. [Last accessed 1st Oct. 2023].
13. Laurens Van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(11), 2008.
14. Incremental PCA. https://scikit-learn/stable/auto_examples/decomposition/plot_incremental_pca.html. [Last accessed 1st Oct. 2023].
15. sklearn.decomposition.PCA. https://scikit-learn/stable/modules/generated/sklearn.decomposition.PCA.html. [Last accessed 1st Oct. 2023].