# Introduction to the Mathematics of Machine Learning

pre-release version 0.1

Kristof Neys
Department of Computer Science
Birkbeck, University of London

February 2022

# Contents


**1 Introduction** **2**

**2 Linear Algebra for Machine Learning** **4**
- 2.1 Preliminaries . . . 4
  - 2.1.1 Let's talk numbers... . . . 4
  - 2.1.2 Functions . . . 5
- 2.2 A Data Point is a Vector . . . 8
  - 2.2.1 In the beginning... . . . 8
  - 2.2.2 What are vectors? . . . 8
  - 2.2.3 Special vectors . . . 10
  - 2.2.4 Operations on vectors (vector algebra) . . . 11
  - 2.2.5 Vector spaces . . . 15
  - 2.2.6 Projections . . . 18
- 2.3 Matrices . . . 21
  - 2.3.1 Context . . . 21
  - 2.3.2 Types of Matrices and Matrix operations . . . 23
  - 2.3.3 Matrix subspaces . . . 30
  - 2.3.4 Tensors . . . 30
  - 2.3.5 Linear Transformations . . . 32
  - 2.3.6 Eigenvalues and Eigenvectors . . . 38
  - 2.3.7 Eigenspace and Eigenbasis . . . 42
  - 2.3.8 Matrices & Orthogonality . . . 42
- 2.4 Matrix Decomposition . . . 44
  - 2.4.1 Eigendecomposition . . . 44
  - 2.4.2 Singular Value Decomposition . . . 46
  - 2.4.3 Properties of SVD . . . 47
  - 2.4.4 Low Rank Matrix Approximation . . . 47
- 2.5 Example: Term-Document Matrices . . . 48

**3 Calculus for Machine Learning** **50**
- 3.1 Introduction . . . 50
  - 3.1.1 Why calculus? . . . 50
  - 3.1.2 Limits . . . 50
- 3.2 Differentiation . . . 51
- 3.3 Integration . . . 58

# Chapter 1

# Introduction

Updated versions can be found at:
`https://github.com/Kristof-Neys/Intro_Maths_4_ML`
Please email any errors or comments to:
cneyso1@dcs.bbk.ac.uk

**For who is this written?**
The purpose of this text is to focus on providing the mathematical foundations for understanding these algorithms. Put differently, since algorithms are written in the language of mathematics, to work with algorithms on a professional level, you have to have a solid understanding of this language.
As such, the context of this text is to provide non-mathematicians an introduction to this language and this from an angle of machine learning. The lack of rigour will make most mathematicians object strongly and possibly even call it entirely incorrect, but it is this 'cutting corners' approach that is precisely there to remain focused on the machine learning use cases and thereby disregarding the full context and subtleties of various mathematical concepts.
Whilst doing the research for this text I have found many books, papers and texts explaining all mathematical methods and concepts that underlie the various machine learning algorithms to the $n^{th}$ degree. However, none of them accompanied the student from high-school maths to the core mathematical concepts. This document is entirely written for those students of machine learning who either did not study some of the basic mathematical concepts that are used in machine learning or are coming back to the field after some time away from formal study, i.e. the mature student. As such, this is very much an introductory text to the subject of mathematics of machine learning.

### Broad outline

In broad terms, we can identify three core pillars that provide the foundation of the mathematics of machine learning:

1. Linear Algebra
2. Mathematical Analysis
3. Probability theory & Calculus

In the first chapter we will cover linear algebra in detail, illustrating how we can represent and manipulate data, as well as explain mathematical analysis. The second chapter illustrates the use of calculus, with the main goal to provide a solid understanding of *gradient descent.*

### With the practitioner in mind

Furthermore, the document is very much targeted to the machine learning practitioner, and as such aims to illustrate how to implement the various computations using Python. Most examples will be illustrated through the use of such code snippets. Finally, each important concept will be summarised and highlighted using a *Heuristic*. The intention is that these memory aides will assist in retaining these concepts in short-term memory for further processing.

**Heuristic**

*"A heuristic, or heuristic technique, is any approach to problem solving or* ***self-discovery that employs a practical method*** *that is not guaranteed to be optimal, perfect, or rational, but is nevertheless sufficient for reaching an immediate, short-term goal or approximation".*

### Notation conventions

We have aimed to follow the notation conventions which are as general as possible. Vectors are denoted as bold face lower case letters, e.g., $\mathbf{v}$. Matrices are denoted by bold-face capital letters, e.g., $\mathbf{A}$ and tensors are written as caligraphic letters, e.g., $\mathcal{T}$. We have used the standard notation for vector and matrix elements; the $i$th entry of a vector $\mathbf{v}$ is denoted by $v_i$, element $(i, j)$ of a matrix $\mathbf{A}$ is denoted by $a_{ij}$ and element $(i, j, k)$ of tensor $\mathcal{T}$ is denoted by $t_{ijk}$. For the Euclidean norm we use $= \|\cdot\|$ (and not $\|\cdot\|_2$).

# Chapter 2

# Linear Algebra for Machine Learning

## 2.1 Preliminaries

### 2.1.1 Let's talk numbers...

Numbers come in all different shapes and sizes and are generally categorised over five classes:

1. Natural numbers - $\mathbb{N}$: also called positive integers, counting numbers, or natural numbers: $\{1, 2, 3, 4, 5, \dots\}$
2. Integers - $\mathbb{Z}$: The set of all natural numbers plus zero and all the negatives of the natural numbers: $\{\dots, -2, -1, 0, 1, 2, \dots\}$
3. Rational numbers - $\mathbb{Q}$: fractions where the top and bottom numbers are integers; i.e. $\frac{1}{2}$, $\frac{3}{4}$, $\frac{7}{2}$, $\frac{-4}{3}$, $\frac{4}{1}$
4. Real numbers - $\mathbb{R}$: all numbers that can be written as a decimal. This includes fractions written in decimal form such as: $0.5, 0.752.35, -0.073, 0.3333, 2.142857$. It also includes all the irrational numbers such as $\pi$ and $\sqrt{2}$.
5. Complex numbers - $\mathbb{C}$: A number which can be written in the form $a + bi$ where a and b are real numbers and i is the square root of $-1$.

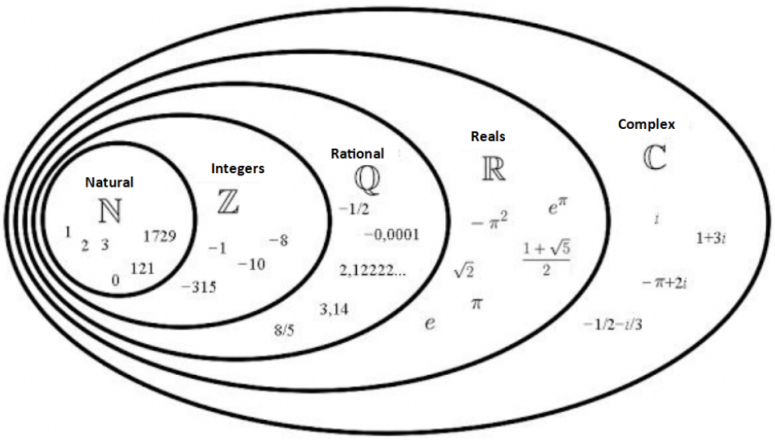


Figure 2.1: Number classification

### 2.1.2 Functions

Functions are to linear algebra and calculus what atoms are to physics. Without a clear grasp of the meaning and behaviour of functions the rest will make little sense. Books have been written on functions, but for our purpose we will limit ourselves to their definition, the input & output of functions and their main properties.
In essence, a functions purpose is to describe the relationship between variables, and more specifically how one variable depends on another. For instance, the amount of profit that Facebook makes depends on the number of users on its platform. In Q4 2021 Facebook made a profit of $10.3 billion, less than the profit in Q4 2020, and lost about a million daily users. Profit is a function of daily users. Tough times at Facebook indeed...
Formally, we define a function to be a *mathematical object* that takes elements from a set as inputs and returns elements of another set as outputs, which we denote as:

$$f : X \to Y$$

A mathematical object is anything that has been (or could be) formally defined

A function is also described as a *mapping* from elements of set $X$ to exactly one element of set $Y$. Should there be a map to more than one element in $Y$ than we can no longer call it a function. As such, we say that "$f$ maps $x$ to $f(x)$". Often the input and output data will be numbers, but as we will see later on, these numbers can be expanded to *vectors*.

Functions come with some specific and important terminology:

- $X$ is the **source** set of the function, and is the set of all the allowed input values for the function. For instance, we could have a function that maps the real numbers to another set of the real numbers, as follows: $f : \mathbb{R} \longrightarrow \mathbb{R}$, then in this example the source set are the real numbers, $\mathbb{R}$. However, that does not mean these are all the allowed input values to the function. For that we need the *domain* of a function

- Dom($f$) is the **domain** and consist of all the values that this specific function can take. Suppose we have $f : \mathbb{R} \longrightarrow \mathbb{R}$ defined as $f(x) = \frac{1}{x}$, then since we cannot divide by 0 the domain is of course all the real numbers excluding 0: $\mathbb{R} \setminus \{0\}$
- $Y$ is the **codomain** of the function and describes the type of outputs the function has. The codomain is also often referred to as the *target set.*
- Im($f$) is the **image** of the function and contains the set of possible output values of the function. The image is also called the *range.*

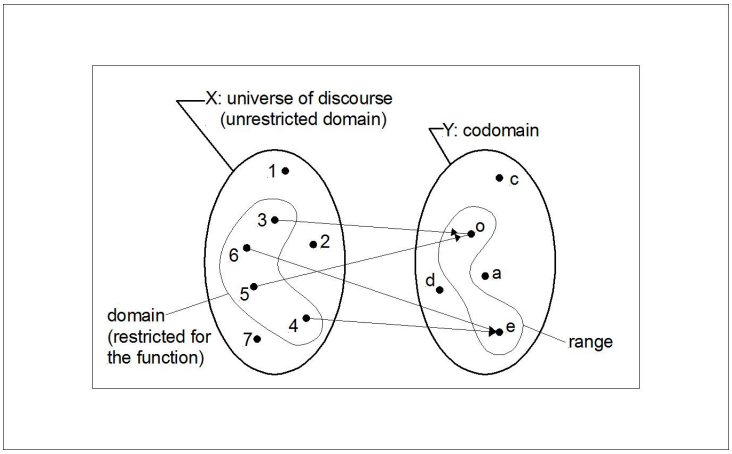


Figure 2.2: Image

**Set & Function notation**
Often function notation can be confusing, but it not need be. Let $f : \mathbb{R} \longrightarrow \mathbb{R}$ and suppose $f$ is defined as: $f(x) = x^2$, which is simply a function that takes in a real number and squares it, then $f(x) = \{x \in \mathbb{R} \mid x \geq 0\}$, which we read as: *"$f(x)$ is defined as the set of real numbers $x$"* (which is the part before the pipe operator, |), the pipe operator is read as "such that" , or "where", and then the part from the pipe operator is: *"such that $x$ is greater than or equal to zero".*

**Properties of functions**

As stated, every function is a mapping, and we can distinguish three important types of mappings:

- **Injective:** There exist a unique-input-for-each-output such that an injective function maps two different inputs to two different outputs. That is, if $x_1 \neq x_2$ in the input space, then $f(x_1) \neq f(x_2)$ in the output space. Injective functions are also called **one-to-one** functions.
- **Surjective** For each element in the codomain there exists **at least** one input in the domain. Hence, in a surjective function, the input does **not** need to be unique. It follows that the image (or range) of $f$ is equal to its domain. Surjective functions are also called **onto** functions.

- **Bijective:** When a function is both injective and surjective we have a bijective function. As such a bijective function defines an exact mapping between each element in the input set and an element in the output set. Consequently, there are no unpaired elements.

**Examples**

- $f : \mathbb{R} \longrightarrow \mathbb{R}, f(x) = 2x$ is injective.
- $f : \mathbb{R} \longrightarrow \mathbb{R}, f(x) = x^2$ is **not** injective, since it maps both $-x$ and $x$ to the same output. Neither is it surjective since it's image is only the nonnegative numbers in $\mathbb{R}$.
- $f : \mathbb{R} \longrightarrow \mathbb{R}, f(x) = x^3 - x$ is surjective but not injective. *Hint:* try it with $x = 0$ and $x = 1$.
- $f : \mathbb{R} \longrightarrow \mathbb{R}, f(x) = x^3$ is surjective, since $\text{Im}f(x) = \mathbb{R}$, that is for each element in the target, i can find at least one element in the domain **and** injective, hence a bijective function.

**Function composition**

As with so many things in life, it's better together, and that applies to functions as well, where we can combine two, or more, simple functions by *chaining* them together. As we will see in chapter 3, function composition also allows us to chain together derivative functions, which in turn play an essential role in machine learning.
*Function composition* comes with its special operator: $\circ$ such that $f \circ g$ denotes first applying function $g$ upon its output we apply function $f$, as follows: $f \circ g(x) = f(g(x))$. Some caution however, function compositions are, naturally, not commutative, and $f(g(x)) \neq g(f(x))$.
As a simple example, if we have $f, g : \mathbb{R} \to \mathbb{R}$, defined as $f(x) = x - 1$, and $g(x) = x^2$, then the function composition: $f(g(x)) = x^2 - 1$, that is the output of $g$ becomes the input to $f$. Reversing the composition, $g(f(x))$, will give you a different result - try it!

## 2.2 A Data Point is a Vector

### 2.2.1 In the beginning...

In the beginning there was no coordinate system to describe an object in space. So humans had to construct (or invent) the concept of a coordinate system. That is, space has no intrinsic grid. Whether we overlay space with a grid of horizontal and vertical lines, or instead with diagonal lines, is entirely arbitrary. What is universal in any coordinate system, however, is the origin. Since, as we will see later on with vectors, we can always backtrack on ourselves from where we started. But the direction of the axis and spacing of the grid lines are different depending on our choice of *basis vectors*, which we will explain in detail later on.
In general terms, we use the *Cartesian plane* to describe a point in a two-dimensional space. In linear algebra, coordinate systems are commonly specified using vectors rather than coordinate axes. This then raises another important item; how to translate between coordinate systems? A coordinate system is like a language. We are all looking at the same object in space (which we can describe as a vector), but in English, a flying bird in metal is called *a plane*, in french it is called une avion. Similarly a grid with horizontal and vertical lines can also be one with diagonal lines.

Not all data is numerical, indeed most of it is not. Think of text, images and so forth. However, all data can be represented *numerically* and thereby suitable for reading into a computer program.

**Heuristic 1**

**Think of data as vectors**

### 2.2.2 What are vectors?

For our, limited purpose, we consider vectors in only two forms, which are interlinked:

1. Geometric vectors
2. Elements of $\mathbb{R}$

**Geometric vectors** Geometric vectors are oriented segments. Therse are the kind of vectors you may have learned about in high-school physics and geometry. Many linear algebra concepts come from the geometric point of view of vectors: space, plane, distance, etc.

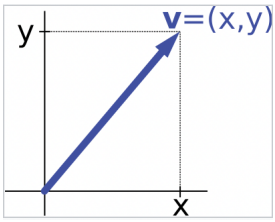


Figure 2.3: Image taken from Wikipedia - ref to follow

**Elements of $\mathbb{R}$** A vector is a finite ordered list of elements, where for our purpose, these elements are real numbers.
A *tuple* is defined as such a finite ordered list of elements. Then, an *n-tuple* is a sequence (or ordered list) of $n$ elements, where $n$ is a positive integer. For instance, $(2, 7, 4, 1, 7)$ is a 5-tuple.
More concise: **A vector is a tuple of one or more values.** The individual elements of a vector are called *components*.
Put differently, vectors are built from components, which are ordinary numbers, and ordinary numbers are called *scalars*

**Heuristic 2**

**Think of a vector as a list of numbers**

Then: $\mathbf{v} = (v_1, v_2, v_3, v_4, v_5)$, such that we call $\mathbf{v}$ a vector and $v_1, v_2, ...$ its components, or *scalar values*. Representing a vector by its components is called, unsurprisingly, the *component notation*.

Crucially, vector components depend on the *coordinate system* in which the vectors are represented. Without the coordinate system, the coordinate values are meaningless. Vectors are often represented in column format and indeed this is the convention we will maintain:

$$\mathbf{a} = \begin{bmatrix} 2 \\ 7 \\ 4 \\ 1 \\ 7 \end{bmatrix} \in \mathbb{R}^5 \tag{2.1}$$

Hence, a row vector is denoted by $\mathbf{a}^T$, the *transpose* of $\mathbf{a}$.

The **dimension of a vector** (or also called *length* or *size*) is the number of elements it contains. Vector $\mathbf{a}$ here above has dimension 5, hence $\mathbf{a} \in \mathbb{R}^5$, and its third entry is 4. Then, a vector of size $n$ is called and *n-vector*.

The raw material for machine learning computations are high-dimensional vectors that represents "rich data" such as text, images, etc. The actual process of converting or transforming a data set into a set of vectors is called *vectorization*.

**Example of vectorization** Consider:

d1: Julie loves me more than Linda loves me
d2: Jane likes me more than Julie loves me

To 'vectorize' we construct a list of the unique words in all the sentences, called the *corpus*, being the following:

$$[me, Jane, Julie, Linda, likes, loves, more, than]$$

Now simply record the frequency of the words in each document as such:

d1: [2, 0, 1, 1, 0, 2, 1, 1]
d2: [2, 1, 1, 0, 1, 1, 1, 1]

The two sentences are now *encoded* as vectors.

### 2.2.3 Special vectors

**Zero vector**

Zero vectors are simply vectors with only zeroes as components and usually they are denoted as 0, regardless of their dimension. Hence, you may see a 3-dimensional or 10-dimensional vector with all entries equal to 0, referred as the 0 vector.

$$0 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{2.2}$$

The zero vector plays an important role in the **kernel** of a linear map, also known as the null space, which we will see later on in Linear Transformations. For now we leave it by the working definition that the **kernel** of a linear transformation is the set of vectors mapped to the zero vector by this linear transformation.

**Unit vector**

Unit vectors are vectors composed of a single element equal to one, and the rest to zero, i.e:

$$\mathbf{x}_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \mathbf{x}_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \mathbf{x}_3 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \tag{2.3}$$

Unit vectors are important both in terms of *unit vector notation* as well as in the area of *vector spaces*. They feature in the formation of a basis of a vector space, which we will work out in detail in the following sections.

Earlier we saw the *component notation* of a vector. In the *unit vector notation* a vector is expressed as a combination of the unit vectors, that is, let:

$$\hat{i} = (1, 0) \text{ and } \hat{j} = (0, 1), \text{ then } \mathbf{v} = \mathbf{v}_1\hat{i} + \mathbf{v}_2\hat{j}$$

**Sparse vector**

Sparse vectors, are vectors with most of its elements equal to zero. We denote the number of nonzero elements of a vector x as **nnz(x)**. The sparsest possible vector is the zero vector. Sparse vectors are common in machine learning applications and often require some type of method to deal with them effectively.

### 2.2.4 Operations on vectors (vector algebra)

The vector operations we will investigate and use are:

1. Addition
2. Subtraction
3. Scalar multiplication
4. Dot product
5. Length
6. Linear combinations

**Addition & Subtraction**

Addition and subtraction is a simple *elementwise* operation such that the inputs are pairs of vectors and the operation produces vectors as outputs, as follows:

$$\mathbf{x} + \mathbf{y} = \begin{bmatrix} 2 \\ 3 \\ 4 \end{bmatrix} + \begin{bmatrix} 5 \\ 6 \\ 7 \end{bmatrix} = \begin{bmatrix} 7 \\ 9 \\ 11 \end{bmatrix} \tag{2.4}$$

Similarly:

$$\mathbf{x} - \mathbf{y} = \begin{bmatrix} 2 \\ 3 \\ 4 \end{bmatrix} + \begin{bmatrix} 5 \\ 6 \\ 7 \end{bmatrix} = \begin{bmatrix} -3 \\ -3 \\ -3 \end{bmatrix} \tag{2.5}$$

Hence, vectors must be of the same *size* (dimension, length) before they can be added together or subtracted and produce a vector of that size/length/dimension. This may look trivial, but often in machine learning code this is where things trip up!

**Properties:**

- Vector addition is *Commutative*, i.e: $\mathbf{x} + \mathbf{y} = \mathbf{y} + \mathbf{x}$
- Vector addition is *Associative*, i.e.: $(\mathbf{x} + \mathbf{y}) + \mathbf{z} = \mathbf{x} + (\mathbf{y} + \mathbf{z})$
- Adding the zero vector has no effect: $\mathbf{x} + 0 = 0 + \mathbf{x} = \mathbf{x}$, notice how the 0 here denotes the *zero vector*
- Subtracting a vector from itself returns the zero vector, i.e.: $\mathbf{x} - \mathbf{x} = 0$

**Numpy code** In `NumPy`, we can compute the addition of vectors inverse of a matrix with the method:

```
x = np.array([[1],[2],[3]])
y = np.array([[4],[5],[6]])

np.add(x,y)
```

Results in:

```
array([[5],
       [7],
       [9]])
```

### Scalar multiplication

(a.k.a *scalar-vector multiplication*)
A vector is multiplied by a scalar (simply a number) which is done by multiplying each component of the vector by the scalar, as follows:

$$(-2)\begin{bmatrix}2\\3\\4\end{bmatrix} = \begin{bmatrix}-4\\-6\\-8\end{bmatrix} \tag{2.6}$$

**Properties of scalar multiplication:** Let $\alpha, \beta$ be scalars, then:

- Associativity: $(\alpha\beta)\mathbf{x} = \alpha(\beta\mathbf{x})$
- Left-distributive property: $(\alpha + \beta)\mathbf{x} = \alpha\mathbf{x} + \beta\mathbf{x}$
- Right-distributive property: $\mathbf{x}(\alpha + \beta) = \mathbf{x}\alpha + \mathbf{x}\beta$
- Right-distributive property for vector addition: $\alpha(\mathbf{x} + \mathbf{y}) = \alpha\mathbf{x} + \alpha\mathbf{y}$

### Dot product (a.k.a Inner product)

The *dot product* takes two vectors as inputs and produces a single, real number as an output (i.e. a scalar), as follows:

$$\mathbf{a} \cdot \mathbf{b} = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$

Hence, it is the sum of the products of the corresponding components, and the following are equivalent:

$$\mathbf{a} \cdot \mathbf{b} = \mathbf{a}^T \mathbf{b} = \mathbf{b}^T \mathbf{a}$$

since by property of *commutativity* we indeed have that: $\mathbf{a}^T \mathbf{b} = \mathbf{b}^T \mathbf{a}$.

The dot product plays a central role in many algorithms and machine learning tasks, and it is the key method for calculating vector projections, vector decompositions as well as to determine orthogonality, which we'll cover later on.

**Numpy code** - using the $x, y$ vector from earlier, we can use the `numpy.dot` method, however, notice how it requires the first vector to be transposed `x.T`:

```
np.dot(x.T,y)
```

Which results in:

```
array([[32]])
```

### Norms (a.k.a Length, Magnitude)

The norm of a vector is a nonnegative number that represents the extent of the vector in space. That is: for geometric vectors that start at the origin, it's the distance to the 'end' of this vector, hence it's also referred to as the length or magnitude of the vector. For our purposes, the *Euclidean norm* is of essential use and its algebraic formula is defined as:

For $\mathbf{u} \in \mathbb{R}^n$:

$$||\mathbf{u}||_2 = \sqrt{u_1^2 + u_1^2 + u_1^2 \cdots + u_n^2}$$

Now, since $u_1^2 + u_1^2 + u_1^2 \cdots + u_n^2$ is simply the dot product of $\mathbf{u}$ with itself, i.e.:

$$\mathbf{u} \cdot \mathbf{u} = u_1^2 + u_1^2 + u_1^2 \cdots + u_n^2$$

it follows that the norm of $\mathbf{u}$ is the square root of its dot product:

$$||\mathbf{u}||_2 = \sqrt{\mathbf{u} \cdot \mathbf{u}}$$

Sometimes the Euclidean norm is also called the $L_2$ norm.

**Geometric representation of the dot product:** Having defined the norm of a vector, we can now define the geometric formula of the dot product as follows:

$$\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\|\|\mathbf{b}\| \cos \alpha$$

where $\alpha$ is the angle between the two vectors. We have now defined the dot product as a function of the product of the vectors' length and their angle.
Rearranging this formula we can construct the angle between two vectors as:

$$\cos \alpha = \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\|\|\mathbf{b}\|}$$

Now, from the geometric factor, $\cos \alpha$, we can derive the following:

- if the vectors point in the same direction, we have:

$$\cos \alpha = \cos 0^\circ = 1, \text{then } \mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\|\|\mathbf{b}\|$$

- if the vectors are perpendicular to each other we have that:

$$\cos \alpha = \cos 90^\circ = 0, \text{then } \mathbf{a} \cdot \mathbf{b} = 0$$

From here we can easily see that when two vectors are close, or indeed on top of each other, that their cosine must be close or equal to one. Consequently; $\mathbf{u}$ and $\mathbf{v}$ are *orthogonal* if the angle between them is $\pi/2$, i.e. $= 0$. Moreover, an important interpretation of $\theta$ is that it is the **geodesic distance** on the unit sphere between the unit vectors $\mathbf{u}/\|\mathbf{u}\|$ and $\mathbf{v}/\|\mathbf{v}\|$.

> **Definition 2.2.1** (**Orthogonality**)**.** Two vectors, $\mathbf{a}$ and $\mathbf{b}$ are **orthogonal** if and only if $\mathbf{a} \cdot \mathbf{b} = 0$. Furthermore, if $\|\mathbf{a}\| = 1 = \|\mathbf{b}\|$, that is both are unit vectors, then $\mathbf{a}$ and $\mathbf{b}$ are **orthonormal**.

Orthogonality of vectors and matrices forms the basis of many machine learning operations.

The norm that we saw earlier now also comes in handy to work out a unit vector, since a unit vector of course has length one, then normalising a vector by dividing by its norm achieves just that:

$$\mathbf{u} = \frac{\mathbf{u}}{\|\mathbf{u}\|}$$

### 2.2.5 Vector spaces

What is even better than one vector? A set of vectors!

Let's start with an example: suppose **a** and **b** are both three-dimensional vectors, than adding them produces another vector, **c** which also has a dimension of three, i.e. $(c_1, c_2, c_3)$.

In essence, a vector space consists of a set of vectors and all possible linear combinations (see below) of these vectors. Vector spaces form part of the mathematical field of *Group Theory*, where a key property of what constitutes a group is the concept of *closure*. The *"all possible combinations"* part of a vector space captures this notion of closure. With respect to the vector spaces this means first of all *closed under addition*, meaning the sum of two vectors is also a vector in that vector space. Now, going back to our example, **c** cannot have a higher dimension, for instance of six as this would of course mean that we are no longer in $\mathbb{R}^3$. Secondly, vector spaces are *closed under scalar multiplication*, when we multiply a vector by a scalar (recall, this is simply a number in $\mathbb{R}$), the resulting 'scaled-up' vector remains in the vector space.

Formally, we write these closure properties for a Vector Space as:

1. $\forall \mathbf{v}, \mathbf{u} \in \mathbf{V}, \mathbf{v} + \mathbf{u} \in \mathbf{V}$
2. $\forall \alpha \in \mathbb{R}$ and $\mathbf{v} \in \mathbf{V}, \alpha\mathbf{v} \in \mathbf{V}$

**Linear combination** Before we move on to define further characteristics of vector space, we must define what a *linear combination* is. This is simply applying the two operations allowed that satisfy the vector space requirements, namely addition and scalar multiplication. Formally:

> **Definition 2.2.2.** Let $\mathbf{V}$ be a vector space with a finite number of vectors: $\mathbf{x}_1, \ldots, \mathbf{x}_k \in \mathbf{V}$. Then, every $\mathbf{v} \in \mathbf{V}$ that is of the form:
>
> $$\mathbf{v} = \lambda_1\mathbf{x}_1 + \cdots + \lambda_k\mathbf{x}_k = \sum_{i=1}^{k} \lambda_i\mathbf{x}_i \in \mathbf{V} \tag{2.7}$$
>
> is a **linear combination** of the vectors $\mathbf{x}_1, \ldots, \mathbf{x}_k$, where $\lambda_1 + \cdots + \lambda_k \in \mathbb{R}$

#### Span & Generating Set

The *span* operator can at times lead to some confusion, but it's best considered in its proper meaning of *"The full extent of something from end to end; the amount of space that something covers."* Applied to vector spaces the span then means the set of all linear combinations of some set of vectors that covers either the entire or a subspace of a vector space.

> **Definition 2.2.3.** Let $\mathbf{V}$ be a vector space and $\mathbf{A}$ a set of vectors such that $\mathbf{A} = \{\mathbf{x}_1, \mathbf{x}_2, ...\mathbf{x}_k\} \subseteq \mathbf{V}$. Then, if every vector $\in \mathbf{V}$ can be expressed as a linear combination of $\mathbf{x}_1, \mathbf{x}_2, ...\mathbf{x}_k$, then we say that $\mathbf{A}$ is a **generating set** of $\mathbf{V}$. Furthermore, the set of all linear combinations of vectors in $\mathbf{A}$ is called the **span** of $\mathbf{A}$. Finally, if $\mathbf{A}$ spans the vector space $\mathbf{V}$ (i.e. covers the whole area/plane), then we write $\mathbf{V} = \text{span}[\mathbf{A}]$.

Hence, *generating sets* are sets of vectors that span vector (sub)spaces. That is, every vector can be represented as a linear combination of the vectors in the generating set. This now poses the

question, can we find the smallest generating set that spans a particular vector space? Since of course, on the other side of the spectrum, the largest generating set would be the set of vectors of that particular vector space. This now brings us to the essential concept in vector algebra of *basis*. However, to define this new concept, we first need to explain another cornerstone of linear algebra, the notion of *linear independence*.

### Linear independent

One of the core principles in Mathematics is the aim to do more with less. Now, suppose we are given a set of vectors; $\{\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3\}$ then naturally we want to know whether one of the vectors in this set can be expressed as a linear combination of any or all the other vectors. For instance, if $\mathbf{v}_3 = \mathbf{v}_1 + \mathbf{v}_2$ then the set is not independent, as one vector depends on the others. Conversely, if none of the vectors in the set can be expressed as some linear combination of the other members in that set, then we have found a set that is *linearly independent*. Formally we define this concept as:

> **Definition 2.2.4.** A set of vectors $\{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n\} \in \mathbb{R}^n$ is said to be **linearly independent** if the vector equation
>
> $$k_1\mathbf{v}_1 + k_2\mathbf{v}_2 + \cdots + k_n\mathbf{v}_n = 0$$
>
> only has the trivial solution: $k_1 = k_2 = \cdots = k_n = 0$. Otherwise the set is called **linearly dependent**

In its most basic form, the set of standard unit vectors is linearly independent, since, suppose we are in $\mathbb{R}^3$, then we have:

$$\mathbf{e}_1 = (1, 0, 0), \mathbf{e}_2 = (0, 1, 0), \mathbf{e}_3 = (0, 0, 1)$$

Then the only way we can have:

$$k_1\mathbf{e}_1 + k_2\mathbf{e}_2 + k_n\mathbf{e}_3 = \mathbf{0}$$

Is when $k_1 = k_2 = k_3 = 0$, i.e. the trivial solution. Hence, $\{\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3\}$ is independent.

### Basis

The concept of the basis of a vector space, and by extension the basis of a matrix, is core to the mathematics of machine learning, as transformations are built upon them and most machine learning operations, or certainly deep learning operations, consists of a sequence of transformations. Now, as we have shown, we can add vectors together and multiply them with scalars and thanks to the closure property the resulting vector still lives in the same vector space. Now, we can even find a set of vectors with which we can represent every vector in the vector space by adding them and scaling them. We call this set of vectors the *basis*.

> **Definition 2.2.5** (Basis)**.** We say that $\mathbf{B} = \{\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_n\}$ is a **basis** for the vector space $\mathbf{V}$ if $\mathbf{B}$ satisfies the *Spanning property* and *Linear independence property*.

Hence, for the spanning property to be satisfied we must have that any vector $\mathbf{v} \in \mathbf{V}$ can be expressed as a linear combination of the basis vectors:

$$\mathbf{v} = \mathbf{v}_1\mathbf{e}_1 + \mathbf{v}_2\mathbf{e}_2 + \cdots + \mathbf{v}_n\mathbf{e}_n$$

As such, this property guarantees that the vectors in the basis $\mathbf{B}$ are sufficient to represent any vector in $\mathbf{V}$.

The Linear independence property ensures that none of the vectors that form the basis, i.e. $\{\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_n\}$, are redundant. Another version of the definition is the following:

***The basis of a vector space is a set of linearly independent vectors that span the full space***

**Heuristic 3**

**Think of the basis as a pointer that allows you to reach any point in the space you want**

Also, another helpful interpretation is to recall that a vector's components describe how much of the vector lies in a given direction. Then, components, coordinates and coefficients are all ways to describe vectors with respect to a basis.

### Vector Subspaces

The core idea of *vector subspaces* is that they are sets contained in the original vector space with the property that when we perform vector space operations, such as addition, scalar multiplication etc, that we satisfy the closure property. Put differently, we never leave the vector subspace. Vector subspaces occur often in machine learning operations, such as dimensionality reduction. Vector subspaces are another one of these key concepts in machine learning which we will encounter many times, for instance when we work on dimensionality reduction methods. The intuition is that vector subspaces are simply sets contained in the original vector space. Now, since they are part of a vector space, they of course need to satisfy the conditions of a vector space, i.e. to satisfy *closure*. Let $\mathbf{V}$ be a vector space, then we use the notation $\mathbf{W} \subseteq \mathbf{V}$ to name $\mathbf{W}$ as a *subspace* of $\mathbf{V}$, *if* the following conditions are satisfied:

1. $\mathbf{W}$ is contained in $\mathbf{V}$: for all $\mathbf{w}$, if $\mathbf{w} \in \mathbf{W}$, then $\mathbf{w} \in \mathbf{V}$
2. $\mathbf{W}$ is closed under addition and multiplication:
   for all $\mathbf{w}_1, \mathbf{w}_2 \in \mathbf{W}, \mathbf{w}_1 + \mathbf{w}_2 \in \mathbf{W}$, and
   for all $\mathbf{w} \in \mathbf{W}, \alpha\mathbf{w} \in \mathbf{W}$, where $\alpha \in \mathbb{R}$

The last condition implies that the zero vector, $\mathbf{0}$ is always contained in the subspace, since the scalar, $\alpha$, can of course be 0.

### 2.2.6 Projections

Linear transformations are a cornerstone of machine learning, with *projections* being an important class of these linear transformations. Concepts like "embeddings", "low-dimensional representation", or "dimensionality reduction", are all examples of projections.
In our endeavours we will often, if not always, deal with high dimensional data, which by its nature is hard to visualize and analyze, not to mention of the memory impact etc. However, high-dimensional data is also often characterised by the property that only a few dimensions contain most information, with the other dimensions being less important to describe the key properties of the data. Hence, reducing this high-dimensional data to lower dimensions allows us to analyze, visualize and indeed work more efficiently with high-dimensional data. This process of dimensionality reduction is what 'projecting' high-dimensional data onto a lower dimensional feature space is all about. We will also see that projections can be represented as matrices acting on vectors.

Put simply, projections are mappings from a space onto a subspace, or from a set of vectors onto a subset of vectors.
As an heuristic, a projection is best viewed as when having two vectors, both from the origin, overlaying one vector onto the other using the shortest distance.

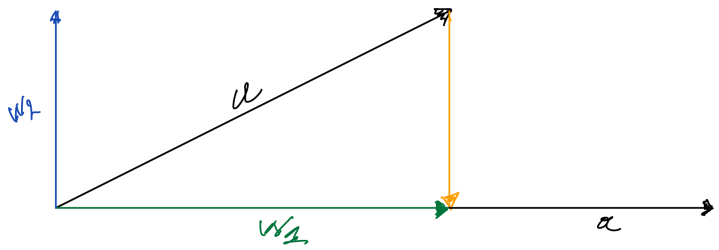


Figure 2.4: by hand...

As an example, and within the context of dimensionality reduction, we can *decompose* a vector $\mathbf{u}$ into a sum of two terms:

1. term 1: a scalar multiple of a nonzero vector, say $\mathbf{a}$ - this is the *scalar projection*
2. term 2: a term orthogonal to $\mathbf{a}$ - this is the *orthogonal projection*

As in image 2.4, suppose we have $\mathbf{u}$ and $\mathbf{a} \in \mathbb{R}^2$, and both have their initial point at $\mathbf{Q}$, where we want to 'project', $\mathbf{u}$ onto $\mathbf{a}$. Then we go about as follows:

- We start by going to the tip of $\mathbf{u}$ and dropping a perpendicular line onto $\mathbf{a}$.
- construct a vector $\mathbf{w}_1$ from $\mathbf{Q}$ to the point where the perpendicular line crosses $\mathbf{a}$
- construct the vector $\mathbf{w}_2 = \mathbf{u} - \mathbf{w}_1$

We have now shown that we can decompose $\mathbf{u}$ into a simple linear sum of two orthogonal vectors: $\mathbf{w}_1$ which is a scalar multiple of $\mathbf{a}$ and $\mathbf{w}_2$, a vector that is orthogonal to $\mathbf{a}$.

$$\mathbf{u} = \mathbf{w}_1 + \mathbf{w}_2 = \mathbf{w}_1 + (\mathbf{u} - \mathbf{w}_1)$$

This now allows us to derive the projection formulas as follows:

1. We know $\mathbf{w}_1$ is a scalar of $\mathbf{a}$, then we must have that:

$$\mathbf{w}_1 = k\mathbf{a}, \text{ where } k \in \mathbb{R}$$

In this example k is around 0.6.

2. Since $\mathbf{u} = \mathbf{w}_1 + \mathbf{w}_2$, then $\mathbf{u} = k\mathbf{a} + \mathbf{w}_2$

3. We compute the dot product on both sides, by $\mathbf{a}$:

$$\begin{aligned}\mathbf{u} \cdot \mathbf{a} &= (k\mathbf{a} + \mathbf{w}_2) \cdot \mathbf{a} \\ \mathbf{u} \cdot \mathbf{a} &= (k\mathbf{a} \cdot \mathbf{a}) + (\mathbf{w}_2 \cdot \mathbf{a}) \\ \mathbf{u} \cdot \mathbf{a} &= k\|\mathbf{a}\|^2 + (\mathbf{w}_2 \cdot \mathbf{a})\end{aligned}$$

4. Now, since we have that $\mathbf{w}_2$ is to be orthogonal to $\mathbf{a}$, it must be that the term $(\mathbf{w}_2 \cdot \mathbf{a})$ is zero, hence:

$$\mathbf{u} \cdot \mathbf{a} = k\|\mathbf{a}\|^2$$

5. Re-arranging terms we get:

$$k = \frac{\mathbf{u} \cdot \mathbf{a}}{\|\mathbf{a}\|^2}$$

6. Recall, we have that $\mathbf{w}_1 = k\mathbf{a}$ and $\mathbf{w}_2 = \mathbf{u} - k\mathbf{a}$, then substituting terms, we get the following projection formulas:

$$\text{vector component of u along a: } \mathbf{w}_1 = \frac{\mathbf{u} \cdot \mathbf{a}}{\|\mathbf{a}\|^2}\mathbf{a}$$

$$\text{orthogonal component of u along a: } \mathbf{w}_2 = \mathbf{u} - \frac{\mathbf{u} \cdot \mathbf{a}}{\|\mathbf{a}\|^2}\mathbf{a}$$

**Notation** The vector component projection is often referred to as the *scalar projection* and denoted by the symbol $\text{proj}_a\mathbf{u}$ for the projection of $\mathbf{u}$ on $\mathbf{a}$, i.e.

$$\text{proj}_a\mathbf{u} = \frac{\mathbf{u} \cdot \mathbf{a}}{\|\mathbf{a}\|^2}\mathbf{a}$$

As an example on how projection is used, and why this so important in machine learning applications, we will illustrate as follows. As we saw earlier, any coordinate system has a set of basis vectors. Often we will need to convert a vector that is expressed in one basis into another basis.

**Example** We are given vector:

$$\mathbf{v} = \begin{bmatrix} 5 \\ -1 \end{bmatrix} \in \mathbb{R}^2$$

written in the standard basis, i.e.:

$$5 \begin{bmatrix} 1 \\ 0 \end{bmatrix} + (-1) \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 5 \\ -1 \end{bmatrix}$$

Suppose, we are given two orthogonal vectors, $\mathbf{b}_1$ and $\mathbf{b}_2$, where:

$$\mathbf{b}_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \mathbf{b}_2 = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$$

Now the task is to find $\mathbf{v}$ expressed in the basis defined by $\mathbf{b}_1$ and $\mathbf{b}_2$. This can now easily be solved with our projection method, as what we are trying to find is the orthogonal projection of $\mathbf{v}$ along either basis vector $\mathbf{b}_1$ and $\mathbf{b}_2$. Hence, we simply apply:

$$\text{proj}_{b_1}\mathbf{v} = \frac{\mathbf{v} \cdot \mathbf{b}_1}{\|\mathbf{b}_1\|^2}\mathbf{b}_1$$

Where: $\mathbf{u} \cdot \mathbf{b}_1 = \begin{bmatrix} 5 \\ -1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 1 \end{bmatrix} = 4$ and $\|\mathbf{b}_1\|^2 = 2$ and so we get $\frac{4}{2} = 2 \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 2 \\ 2 \end{bmatrix}$

Doing the same for projecting $\mathbf{v}$ onto $\mathbf{b}_2$ gets us: $3 \begin{bmatrix} 1 \\ -1 \end{bmatrix} = \begin{bmatrix} 3 \\ -3 \end{bmatrix}$

As such we have found $\mathbf{v}$ expressed in the new basis of $\mathbf{b}_1$ and $\mathbf{b}_2$ as $\begin{bmatrix} 2 \\ 3 \end{bmatrix}$.

We can do a quick check:

$$2 \begin{bmatrix} 1 \\ 1 \end{bmatrix} + 3 \begin{bmatrix} 1 \\ -1 \end{bmatrix} = \begin{bmatrix} 5 \\ -1 \end{bmatrix}$$

As we will illustrate, projecting vectors forms the core method upon which change of basis is build. And a change of basis in turn is one of those fundamental geometric functions, upon which many machine learning, and indeed deep learning applications are built

Another view on projections is how one vector will 'collapse' onto another vector. By determining such a method, this now allows us to project (collapse if you like) any vector onto the basis vectors of a vector space.

## 2.3 Matrices

### 2.3.1 Context

Matrices and vectors are the cornerstones of machine learning. If with vectors we can represent a data point, such as a pixel, then matrices allow us to represent sets of data points, such as an image. But matrices can do lots more. The format and context of matrices that most students are familiar with is as a solution to solving simultaneous equations. For instance, suppose we want to find the price of apples and bananas from two recent shopping trips, where our first trip we bought 2 apples and 3 bananas at a cost of 8 and our second trip we bought 1 apple and 2 bananas at a cost of 5, then we can write this as a simultaneous equation:

$$\begin{aligned} 2a + 3b &= 8 \\ a + 2b &= 5 \end{aligned}$$

Then, we recall that this can be written in matrix format as:

$$\begin{bmatrix} 2 & 3 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} 8 \\ 5 \end{bmatrix}$$

As such, we can see that matrices compactly represent systems of linear equations.

But again, matrices are soo much more, and indeed represent a mathematical *object* in its own right. They even stretch beyond the realm of linear algebra into graph theory, where for instance an adjacency matrix can represent a graph, as well as geometry and other fields. Furthermore, as we will see, matrices also represent linear functions, another corner stone in machine learning.
With this as background, we can define matrices in its broadest form as:

> **Definition 2.3.1.** A matrix $A \in \mathbb{R}^{m \times n}$ is a rectangular array of real numbers with m rows and n columns

Where the numbers in the array are called the ***entries*** and the ***size*** of a matrix is defined in terms of the number or rows and columns it contains. For example, a $3 \times 2$ matrix is defined as:

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{bmatrix}$$

Then, to specify the matrix $A$, we need to determine the values of the six entries: $a_{11}, a_{12}, a_{21}, a_{22}, a_{31}, a_{32}$

Now that we have a good understanding of vectors, we can consider a matrix simply as an ordered collection of vectors. Recall that we can treat a vector either as a *column* vector, or as a row vector. As per our convention which we agreed earlier, $\mathbf{v} \in \mathbb{R}$ denotes a column vector, hence it's transpose, $\mathbf{v}^T \in \mathbb{R}$ represents a row vector. Now, $(1, n)$-matrices and $(n, 1)$-matrices are special matrices that are simply *row/column vectors*.

**Heuristic 4**

**Think of matrices as a (horizontal) stack of column vectors**

Treating a matrix as a collection of vectors will prove to be of great help in our understanding and deployment of machine learning methods. Furthermore, the concept of an ordered collection of vectors representing a matrix can be extended to an ordered collection of matrices, which brings us to the concept of *tensors* and into the realm of multi-linear algebra. In practice, most inputs in a machine learning workflow consists of tensors, which is where the neural network engine *TensorFlow* draws its name from.

### Matrices as functions

How vectors are transformed by matrices is at the heart of linear algebra. Matrices can be thought of as a function that transforms the basis vectors of a space. As such matrices are objects that rotate and stretch vectors, which in turn are representations of Data. And in deep learning, the model has learned through the construction of learned weights. Indeed, the main 'use case' of matrices for our purposes is their role in *linear transformations*. We will deal with linear transformations in detail later on, but for now, in its most intuitive way, linear transformations can be viewed as a method to move around space, preserving the origin and ensuring that grid lines remain parallel and evenly spaced. Matrices then give us a 'language' to describe these transformations.

**Heuristic 5**

**Matrices can transform space**

### 2.3.2 Types of Matrices and Matrix operations

#### Types of Matrices

**Zero Matrix** A zero matrix is a matrix with all elements equal too zero. Typically, the zero matrix is simply denoted by 0.

**Square Matrix** As the name suggests, the dimension of the rows needs to equal the dimensions of the columns (put differently, row and column vectors that make up the matrix have to be of same length).

**Identity Matrix** The identity matrix performs an essential role in matrix operations and has a higher theoretical purpose within *Group Theory*. It is always square and its *diagonal* elements are all equal to one, and it's off-diagonal elements are zero. We denote them by $\mathbf{I}$ and define them formally as:

> **Definition 2.3.2.** We define the $n \times n$ matrix $\mathbf{I}$ with 1's on the diagonal and 0's elsewhere as the **identity** matrix.

For example:

$$\mathbf{I} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Which is easily called in `Numpy`:

```
np.identity(3)

array([[1.,  0.,  0.],
       [0.,  1.,  0.],
       [0.,  0.,  1.]])
```

**Sparse Matrix** A matrix $\mathbf{A}$ is said to be *sparse* if many of its entries are zero. Many common matrices are sparse. An $n \times n$ identity matrix is sparse, since it has only $n$ nonzeros. We will encounter sparse matrices, and sparse vectors, often in our workflow. For instance, a typical customer purchase history matrix is sparse, since each customer has likely only purchased a small fraction of all the products available.

**Diagonal Matrix** A diagonal matrix is nothing more than a matrix where all the off-diagonal entries are zero. An identity matrix is a prime example. Often we denote this as, for instance, $\text{diag}(0.2, -3, 1.2)$, which is:

The main diagonal is also called "principal, primary or leading" diagonal.

$$\text{diag}(0.2, -3, 1.2) = \begin{bmatrix} 0.2 & 0 & 0 \\ 0 & -3 & 0 \\ 0 & 0 & 1.2 \end{bmatrix}$$

**Matrix transpose** Now that we know what a diagonal matrix is, we can use this concept and "flip" matrix $\mathbf{A}$ around its diagonal to obtain the *transpose* of $\mathbf{A}$, denoted as $\mathbf{A}^T$. Or equivalently, we can obtain $\mathbf{A}^T$ by writing the columns of $\mathbf{A}$ as the rows of $\mathbf{A}^T$. Formally, we define this as:

> **Definition 2.3.3.** If $\mathbf{A}$ is any $m \times n$ matrix, then the **transpose of A**, denoted as $\mathbf{A}^T$, is defined to be the $n \times m$ matrix that results by interchanging the rows and columns of $\mathbf{A}$.

We can call the transpose in `Numpy` with the simple `T` method:

```
A = np.array([[1, 2],
              [3, 4],
              [5, 6]])
A.T
```

Results in:

```
array([[1, 3, 5],
       [2, 4, 6]])
```

Note that the entries on the diagonal of the matrix do not change when we apply the transpose operation.

The transpose of a matrix will feature heavily when we are working with matrix decomposition, and we will make use of its properties.

**Matrix transpose - properties**

Let $\mathbf{A}$ and $\mathbf{B}$ be matrices, then:

$$(\mathbf{A}+\mathbf{B})^T = \mathbf{A}^T + \mathbf{B}^T$$
$$\left(\mathbf{A}^T\right)^T = \mathbf{A}$$
$$(\mathbf{A}+\mathbf{B})^T = \mathbf{A}^T + \mathbf{B}^T$$
$$\left(\mathbf{A}^T\right)^T = \mathbf{A}$$

**Symmetric Matrix** A square matrix $\mathbf{A}$ is said to be *symmetric* if $\mathbf{A} = \mathbf{A}^T$.
We can neatly combine the properties of transpose and inverse matrix. If $\mathbf{A}$ is invertible, then so is $\mathbf{A}^T$, and:

$$\left(\mathbf{A}^{-1}\right)^T = \left(\mathbf{A}^T\right)^{-1} =: \mathbf{A}^{-T}$$

**Triangular Matrix** Square matrices are said to be *upper triangular* when the elements below the main diagonal are zero and are said to be *lower triangular* when the elements above the main diagonal are zero.

> **Definition 2.3.4.** We say that $\mathbf{U} \in \mathbb{R}^{m \times n}$ is an **upper triangular** matrix if $u_{ij} = 0$ when $i > j$, that is, the elements below the diagonal are zero. Similarly, we call $\mathbf{U}$ a **lower triangular** matrix if $u_{ij} = 0$ when $i < j$

**Matrix inverse:** Some would say, that mathematics is like an unreliable boy-friend, always looking to go back on a commitment. Well, most operations in mathematics have an inverse operation, and matrices are not different.
Recall, when working with real numbers, the reciprocal of a number $a$ is the number $x$ such that when we multiply $a$ and $x$ we get 1, i.e. $ax = 1$. Then of course it must be that $x = \frac{1}{a}$, also often denoted as $a^{-1}$. Matrices, broadly, follow the same principle and the inverse of a matrix is formally defined as:.

> **Definition 2.3.5.** If $\mathbf{A}$ and $\mathbf{B}$ are square matrices, such that $\mathbf{AB} = \mathbf{BA} = \mathbf{I}$ then we say that $\mathbf{A}$ is invertible (nonsingular) and $\mathbf{B}$ is the **inverse** of $\mathbf{A}$.

A key difference with the inverse of numbers is that not every matrix is invertible, there is no "undo" operation for them. Also, invertible matrices **must** be square matrices. It will not work for rectangular matrices or vectors (try it out).

**Properties of matrix inverse operations**:

- $(\mathbf{AB})^{-1} = \mathbf{B}^{-1}\mathbf{A}^{-1}$
- $(\mathbf{ABC})^{-1} = \mathbf{C}^{-1}\mathbf{B}^{-1}\mathbf{A}^{-1}$
- $\left(\mathbf{A}^T\right)^{-1} = \left(\mathbf{A}^{-1}\right)^T$
- $\left(\mathbf{A}^{-1}\right)^{-1} = \mathbf{A}$

In `NumPy`, we can compute the inverse of a matrix with the `.linalg.inv` method:

```
A = np.array([[1, 2, 1],
              [4, 4, 5],
              [6, 7, 7]])

A_i = np.linalg.inv(A)
print(f'A inverse:\n{A_i}')
```

results in:

```
A inverse:
[[-7. -7.  6.]
 [ 2.  1. -1.]
 [ 4.  5. -4.]]
```

Matrix inverses are important as they allow us to solve systems of linear equations.

### Matrix Trace & Determinant

The operations of *determinant* and *trace* will allow us to *summarize* a matrix. In addition, they are key components to determine the *characteristic polynomial*, which we need in order to work with the Queen and King in Matrix land, namely *eigenvalues* and *eigenvectors*.

**Matrix trace:** A trace of a square matrix is the sum of the values on the main diagonal of the matrix (top-left to bottom-right).

$$tr(\mathbf{A}) = \sum_{i=1}^{n} a_{ii}$$

**Matrix determinant:** The determinant method is very useful as it allows us to capture all the entries of a square matrix and distill it into one single number. It can do all kinds of other tricks as well. It can tell us whether a matrix is *singular* or invertible. For instance, for a square matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$, if $\det(\mathbf{A}) = 0$, then $\mathbf{A}$ is singular, that is $\mathbf{A}$ does not have an inverse (noninvertible). Conversely, when $\det(\mathbf{A}) = 1$, then $\mathbf{A}$ does have an inverse, and all kinds of great stuff can be done with $\mathbf{A}$.
The mathematical perspective of determinants is a geometric one. Determinants indicate the sign area of a parallelogram (e.g., a rectangular area) and the sign volume of the parallelepiped, for a matrix whose columns consist of the basis vectors in Euclidean space. For our purposes this is overkill. We will be primarily be interested in the *scaling factor* property that determinants possess as well as their use in determining eigenvalues and eigenvectors.

The name "determinant" refers to the property of "determining" if the matrix is singular or not.

Formally, we represent this as:

$$\det : \mathbb{R}^{n \times n} \to \mathbb{R}$$

The issue with determinants is that they do not scale. In a typical machine learning production workflow we will be dealing with matrices of dimensions up to a billion (yes, that would be $\mathbb{R}^{1,000,000,000 \times 1,000,000,000}$) and computing the determinant on such large matrices becomes infeasible. Nevertheless, determinants allow us to understand conceptually some of the important aspects of matrix decomposition and allow us to work through them.

The determinant of a matrix is a single number that tells whether a matrix is invertible or singular, this is, whether its columns are linearly independent or not, which is one of the most important things you can learn about a matrix. Actually, the name "determinant" refers to the property of "determining" if the matrix is singular or not. Specifically, for an square matrix An×n, a determinant equal to 0, denoted as det(A=0), implies the matrix is singular (i.e., noninvertible), whereas a determinant equal to 1, denoted as det(A)=1
, implies the matrix is not singular (i.e., invertible). Although determinants can reveal if matrices are singular with a single number, it's not used for large matrices as Gaussian Elimination is faster. Recall that matrices can be thought of as function action on vectors or other matrices. Thus, the determinant can also be considered a linear mapping of a matrix A onto a single number. But, what does that number mean? So far, we have defined determinants based on their utility of determining matrix invertibility. Before going into the calculation of determinants, let's examine determinants from a geometrical perspective to gain insight into the meaning of determinants.
The determinant of a square matrix is a scalar representation of the volume of the matrix.
The determinant describes the relative geometry of the vectors that make up the rows of the matrix. More specifically, the determinant of a matrix A tells you the volume of a box with sides given by rows of A.

A much more important interpretation of the determinant is within the context of a linear transformation, as we will see later on.

### Matrix operations

The matrix operations we need to get under our belt are:

- Addition & subtraction: $\mathbf{A} + \mathbf{B}, \mathbf{A} - \mathbf{B}$
- Scaling by a constant $\alpha$: $\alpha\mathbf{A}$
- Matrix-vector product: $\mathbf{Av}$
- Matrix-matrix multiplication: $\mathbf{AB}$
- Matrix inverse: $\mathbf{A}^{-1}$
- Trace of a matrix: $\text{Tr}(\mathbf{A})$
- Determinant: $\det(\mathbf{A})$

**Matrix Addition & subtraction** The sum of two matrices $\mathbf{A} + \mathbf{B} \in \mathbb{R}^{m \times n}$ is simply adding up the entries, *component-wise*, provided $\mathbf{A}$ and $\mathbf{B}$ are matrices of the same size of course, as follows:

$$\mathbf{A} + \mathbf{B} = \begin{bmatrix} a_{11} + b_{11} & \cdots & a_{1n} + b_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} + b_{m1} & \cdots & a_{mn} + b_{mn} \end{bmatrix}$$

With an example in Numpy (the `add` operator also works):

```
A = np.array([[0,2],
              [1,4]])
B = np.array([[3,1],
              [-3,2]])
```

Then, $\mathbf{A} + \mathbf{B}$, returns:

```
array([[ 3,  3],
       [-2,  6]])
```

**Matrix-scalar product** Recall, a *scalar*, denoted $\alpha$ is simply a real number, $\alpha \in \mathbb{R}$, and therefore *scaling* is just another word for multiplication by a constant. For instance, let $\mathbf{A} \in \mathbb{R}^{2 \times 2}$, then with our $\alpha$, we *scale* $\mathbf{A}$ as follows:

$$\alpha\mathbf{A} = \alpha \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{bmatrix} = \begin{bmatrix} \alpha a_{11} & \alpha a_{12} \\ \alpha a_{21} & \alpha a_{22} \\ \alpha a_{31} & \alpha a_{32} \end{bmatrix}$$

The properties of associativity and distributivity hold, i.e. let $\lambda, \alpha \in \mathbb{R}$, then:

**Associativity**:

$$(\lambda\alpha)\,\mathbf{A} = \lambda\,(\alpha\mathbf{A})$$
$$\alpha\,(\mathbf{BA}) = (\alpha\mathbf{B})\,\mathbf{A} = \mathbf{B}\,(\alpha\mathbf{A}) = (\mathbf{BA})\,\alpha, \mathbf{A} \in \mathbb{R}^{m\times n}, \mathbf{B} \in \mathbb{R}^{n\times k}$$

**Distributivity**:

$$(\lambda + \alpha)\,\mathbf{A} = \lambda\mathbf{A} + \alpha\mathbf{A}$$
$$\lambda\,(\mathbf{A} + \mathbf{B}) = \lambda\mathbf{A} + \lambda\mathbf{B}$$

In `NumPy`, we compute matrix-scalar multiplication with the `*` operator or `multiply` method:

```
alpha = 2
A = np.array([[1,2],
              [3,4]])
alpha * A
```

results in:

```
array([[2, 4],
       [6, 8]])
```

**Matrix-vector product** Recall our bananas and apples that we saw earlier? And now recall that we stated that a matrix is simply a collection of vectors. Well, add the dot product operation into the mix and we can do a matrix-vector operation, as follows:

$$\begin{bmatrix} 2 & 3 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} 5 \\ 6 \end{bmatrix} = \begin{bmatrix} 8 \\ 5 \end{bmatrix}$$

We take the dot product of the row vector with the column vector, i.e. $r_i^T \cdot c_i$ (r for row, c for column):

$$2 \times 5 + 3 \times 6 = 28$$
$$1 \times 5 + 2 \times 6 = 17$$

In `Numpy`, we compute the matrix-vector product with the `dot` method (the @ operator also works):

```
A = np.array([[0,2],
              [1,4]])
x = np.array([[1],
              [2]])
np.dot(A, x)
```

Results in:

```
array([[4],
       [9]])
```

**Matrix-Matrix multiplication:** Now, descending down further into the engine room of machine learning, most of the magic happens with matrix multiplication, defined formally as:

> **Definition 2.3.6.** If $\mathbf{A} \in \mathbb{R}^{m \times p}$ and $\mathbf{B} \in \mathbb{R}^{p \times n}$, then their matrix product $\mathbf{AB} \in \mathbb{R}^{m \times n}$ is given by
>
> $$e_{ij} = (AB)_{ij} = \sum_{k=1}^{q} a_{ik} b_{kj}, \qquad \text{for } 1 \leq i \leq p \text{ and } 1 \leq j \leq r.$$

Now, where have we seen the $e_{ij}$ expression before? Indeed, this is a dot product! Since, to obtain $e_{ij}$ we multiply the elements of the $i$th row of $\mathbf{A}$ with the $j$th column of $\mathbf{B}$ and sum them up, which, again, is equivalent to $r_i^T \cdot c_i$. But if it's a dot product, that means that it only works if the vectors are of the same length. In matrix terms the matrix on the left must have the same column length as the row length of the matrix it is multiplying to its right. Put differently, matrices can only be multiplied if their neighboring dimensions match, that is:

$$\mathbf{A} \in \mathbb{R}^{m \times p} \mathbf{B} \in \mathbb{R}^{p \times n} = C \in \mathbb{R}^{m \times n}$$

Notice how the column dimension of $\mathbf{A}, p$, matches the row dimension of $\mathbf{B}, p$, and the resulting matrix $C$ has the dimension of the "outer" indices, i.e. $m$ and $n$.
Python example

```
A = np.array([[0,2],
              [1,4]])
B = np.array([[1,3],
              [2,1]])
np.dot(A, B)
```

results in:

```
array([[4, 2],
       [9, 7]])
```

**Properties of Matrix multiplication**

- Is **not** commutative: $AB \neq AB$
- Associativity: $(AB)C = A(BC)$
- Associativity with scalar multiplication: $\alpha(AB) = (\alpha A)B$
- Distributivity with addition: $A(B + C) = A + B + AC$
- Transpose of product: $(AB)T = B^T A^T$

All texts on linear algebra would now explain and demonstrate how to solve systems of linear equations using matrices and detail the (tedious) use of Gaussian elimination to solve such systems. However, for our purposes, a detailed understanding of this, although very important, area is not essential and so we refer the interested reader to any text on linear algebra for a detailed explanation.

### 2.3.3 Matrix subspaces

Since matrices are simply collections of vectors, and we know that we can create vector subspaces, then it must be that there also are subspaces of matrices. Stated equivalently, the row and column vectors that make up a matrix form vector subspaces, and therefore matrix subspaces. This of course implies that they satisfy the definition of a subspace (that is the *axioms*), of closure under multiplication, addition and that the subspace contains the zero vector.

We can define three **fundamental subspaces** associated with a matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$:

In effect there are six but for our purposes we do not include the transpose of a matrix

**The column space**
The column space of a matrix $\mathbf{A}$ is composed of all linear combinations of the columns of $A$. We denote the column space as $C(A)$. In other words, $C(A)$ equals the span of the columns of $\mathbf{A}$.
**The row space**
The row space of a matrix A is composed of all linear combinations of the rows of a matrix. We denote the row space as $R(A)$. In other words, $R(A)$ equals to the span of the rows of $\mathbf{A}$.
**The null space**
The null space of a matrix $\mathbf{A}$, denoted as $\mathcal{N}(\mathbf{A})$, is composed of all vectors that the matrix $\mathbf{A}$ sends, i.e maps/transforms, to the zero vector:

$$\mathcal{N}(\mathbf{A}) = \{\mathbf{v} \in \mathbb{R}^m | \mathbf{A}\mathbf{v} = 0\}$$

The *kernel* of a matrix is the null space and the terms are used interchangeably. The kernel term is predominantly used when we talk about functions, and intuitively, the kernel is the set of vectors that a function, say, $\phi$, maps onto the neutral element in the *codomain* of $\phi$.

### 2.3.4 Tensors

In many applications data is organized in more than two categories. Mathematically, such data can often be analysed using multilinear algebra, that is, the algebra of tensors.
Pretty much all deep learning models use tensors as their basic data structure. Tensors are a generalization of matrices to an arbitrary number of dimensions, which in the context of tensors is also called an *axis*. Then, the number of axes of a tensor is also called its *rank*
A *tensor*, or $m$-way array, is a generalization of a vector and a matrix. As such, an $M$th order tensor is an element of the tensor product of M vector spaces, each with its own coordinate system. Hence; a first-order tensor is a vector and a second-order tensor is a matrix. Many examples are restricted to using arrays of data with three dimensions; that is a tensor being $\mathcal{T} \in \mathbb{R}^{l \times m \times n}$. We refer to the different dimensions of the array as *modes* or also as *ways*. Formally we define a tensor as:

> **Definition 2.3.7.** Tensors are multilinear mappings over a set of vector spaces. The order of a tensor $\mathcal{T} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_M}$ is M. An element of $\mathcal{T}$ is denoted as $\mathcal{T}_{i_1 \ldots i_m \ldots i_M}$ where $1 \leq i_m \leq I_m$.

**Example of Tensor representation**

We take a collection of images of $n_p$ persons, where each image is an $m_{i_1} \times m_{i_2}$ array of pixels and we set $n_i := m_{i_1} m_{i_2}$. Next, we stack the columns such that we can represent each image by a vector in $\mathbb{R}^{n_i}$. For instance, say we have a facial image which we cropped to $112 \times 78$ pixels then we would get a vector of length 8736. Now, we assume that each person has been photographed with $n_e$ different facial expressions (say three different expressions; neutral, smiling and sad). Hence, we can now represent the collection of images as a tensor:

$$\mathcal{T} \in \mathbb{R}^{n_i \times n_e \times n_p}$$

that is: the different modes are $i$ for image mode, $e$ for expression mode and $p$ for person mode.

Figure 2.5 below illustrates the tensor representation of a database of faces where we have $n_p$ persons photographed, represented by $n_p$ matrices $\mathbf{T}_{n_p} \in \mathbb{R}^{n_i \times n_e}$. The expressions $n_e$ are along the columns and the rows represent the images; $n_i$. Hence, a tensor representation of a facial database can be represented as $\mathcal{T} \in \mathbb{R}^{n_i \times n_e \times n_p}$

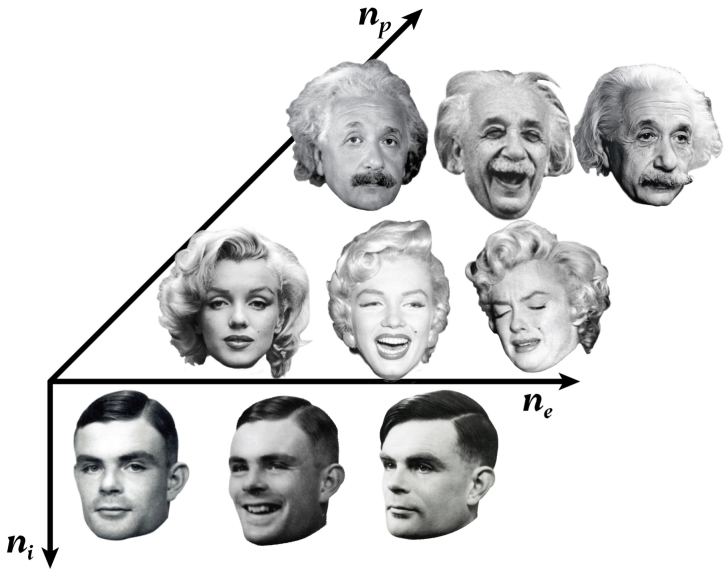


Figure 2.5: Tensor representation of a database of faces, $e$: neutral, smiling, sad

### 2.3.5 Linear Transformations

Linear transformations is another one of those central ideas of linear algebra and forms a key node in the network connecting the various concepts.
Now that we have learned about inner products, orthogonality, and orthogonal/orthonormal bases, we know everything about the structure of our *feature spaces*. However, in machine learning, many of the tasks we will perform revolves around *transforming* the data. For instance, when performing facial recognition, we will want to do stretches of faces, rotations to provide different angles etc.

Recall, a *feature space* are our inputs, such as pixels or text etc.

The broader context is as follows: recall that a **function** is a rule that associates with each element of a set A one and only one element in a set B. For instance, we let $f$ be such a function that associates element $b$ with element $a$, then we write this as

$$b = f(a)$$

Recall also that we said that $b$ is the **image** of $a$ under $f$. Then we said that the set A is called the **domain** of $f$ and the set B the **codomain** of $f$. Finally, we said that the subset of the codomain that consists of all images of points in the domain is called the **range**. Now, all of this we defined for where sets were real numbers, i.e. $a, b \in \mathbb{R}$. A linear transformation, however, is concerned with the functions for which the domain and codomain are **vector spaces**. Indeed, the following heuristic applies:

**Heuristic 6**

**Think of a (linear) transformation as a function**

OK - but where does the "linear" aspect come into it? A transformation is **linear** when two properties are satisfied:
1) All lines (that make up the grid) must remain lines without getting curved.
2) The origin must remain in place.
As such, a linear transformation ensures that grid lines, representing the coordinate system, remain parallel and evenly spaced.

As a first example, we consider transformations from $\mathbb{R}^n$ to $\mathbb{R}^m$ Suppose we have the following set of equations:

$$\begin{aligned} w_1 &= a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \\ w_2 &= a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \\ &\vdots \\ w_m &= a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n \end{aligned}$$

Then of course, in matrix notation this becomes:

$$\begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_m \end{bmatrix} = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ a_{12} & \cdots & a_{2n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

But notice how we have now gone from a vector in $n-$dimensions, the $\mathbf{x}$-vector, to a vector of $m-$dimensions, the $\mathbf{w}$-vector. Continuing with the matrix notation, we can write it more concisely as:

$$\mathbf{w} = \mathbf{A}\mathbf{x}$$

And we have now *transformed* the column vector $\mathbf{x} \in \mathbb{R}^n$ to the column vector $\mathbf{w} \in \mathbb{R}^m$ by multiplying $\mathbf{x}$ on the left by $\mathbf{A}$, which we call a **matrix transformation**, and denote this as:

$$T_A : \mathbb{R}^n \to \mathbb{R}^m$$

As an example, suppose we have the following equations:

$$\begin{aligned} w_1 &= 2x_1 - 3x_2 + x_3 - 5x_4 \\ w_2 &= 4x_1 + x_2 - 2x_3 + x_4 \\ w_3 &= 5x_1 - x_2 + x_4 \end{aligned}$$

Now, suppose that $(x_1, x_2, x_3, x_4) = (1, -3, 0, 2)$ , then in matrix form, using `NumPy`, we get:

```
A = np.array([[2, -3, 1, -5], [4, 1, -2, 1], [5, -1, 4, 0]])
x = np.array([1, -3, 0, 2])

sol = A.dot(x)
print(sol)
```

Resulting in:

```
[1 3 8]
```

That is (and verify):

$$\begin{bmatrix} 1 \\ 3 \\ 8 \end{bmatrix} = \begin{bmatrix} 2 & -3 & 1 & -5 \\ 4 & 1 & -2 & 1 \\ 5 & -1 & 4 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ -3 \\ 0 \\ 2 \end{bmatrix}$$

That is, we went from a vector with dimension 4 to a vector of dimension 3, *dimensionality reduction!* In essence, a linear transformation is a mapping (or a function) on vector spaces that preserves their structure. Formally, we define this as follows:

**Definition 2.3.8.** For vector spaces $V, W$, a mapping $T : V \to W$ is called a linear transformation if:

$$\forall \mathbf{x}, \mathbf{y} \in V, \forall \lambda, \phi \in \mathbb{R} : T(\lambda \mathbf{x} + \phi \mathbf{y}) = \lambda T(\mathbf{x}) + \phi T(\mathbf{y})$$

**Heuristic 7**

**Think of linear transformations as a method for manipulating data to the shape you want it in.**

Now, as by magic, we can represent linear transformations as matrices!

**Key concepts**

- $T : \mathbb{R}^n \to \mathbb{R}^m$: is a linear transformation that takes inputs in $\mathbb{R}^n$ and produces outputs in $\mathbb{R}^m$
- $\mathbf{M}_T \in \mathbb{R}^{m \times n}$: the matrix representation of $T$

Then, multiplying a vector $\mathbf{v}$ by the matrix $\mathbf{M}_T$ is equivalent to the linear transformation of $\mathbf{v}$ by $T$. That is:

$$\mathbf{w} = \mathbf{M}_T\mathbf{v} \qquad \Longleftrightarrow \qquad \mathbf{w} = T(\mathbf{v})$$

Important to observe here is that the columns of a transformation matrix are the axes of the new basis vectors of the mapping in the coordinate system

**Main types of Linear transformations**

- Stretchings
- Shear
- Projections
- Reflections
- Rotations

It is hard to overstate how integral linear transformations are in the computation of any machine learning workflow. Indeed, they perform a lot of the heavy lifting in image processing, computer vision, and other linear applications. As a simple example, every time you rotate an image on screen, you are performing a matrix transformation. Furthermore, combinations of linear and nonlinear mappings are what neural networks do to learn mappings from inputs to outputs, so it is critical to get a good understanding of these.

**Stretchings** Stretching is done by using *scalars* in our identity matrix, $\mathbf{I}$. We have our standard form: $\mathbf{y} = \mathbf{A}\mathbf{x}$, then $\mathbf{A}$ is simply $\alpha\mathbf{I}$, i.e.

$$\begin{bmatrix} \alpha_1 & 0 \\ 0 & \alpha_2 \end{bmatrix}$$

Then such *scaling* stretches a vector, $\mathbf{x}$ by $|\alpha|$ if $\alpha > 1$, it shrinks $\mathbf{x}$ when $\alpha < 1$ and reverses it when when $\alpha < 0$.

**Shear** In a shear mapping points on one axis remain unchanged, while all other points are shifted parallel to that axis.

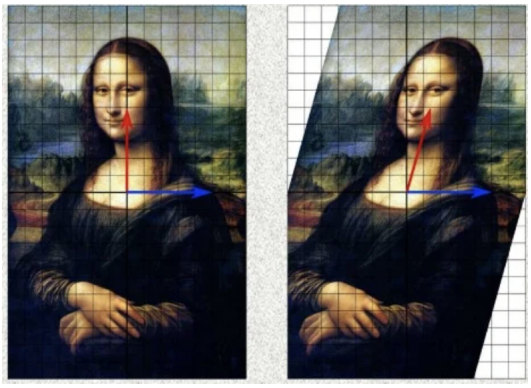

Figure 2.6: The red arrow changes direction but the blue one remains fixed

**Projections** Earlier we defined projections to be mappings from one space to a subspace. Well, this operation can be achieved by representing projections as matrices acting on vectors. Projections are what makes neural networks possible, where in a neural network a projection layer is a matrix multiplication, known as a (dense) layer, without the non-linear activation in the end (sigmoid/tanh/relu/etc.). As such the model projects for instance a 100K-dimensions discrete vector into a 600-dimensions continuous vector.

**Reflections** Reflection is the mirror image of an object in Euclidean space. For the general case, reflection of a vector **x** through a line that passes through the origin is obtained as:

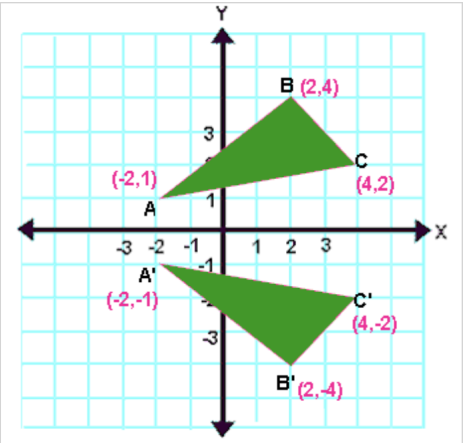


Figure 2.7: Reflection around the x-axis

**Rotations** Rotation mappings do exactly what it says on the tin: they move objects - by convection - counterclockwise in Euclidean space. For the general case in $\mathbb{R}^2$, counterclockwise of vector x by radiants rotations is obtained as:] [**IMAGE TO FOLLOW**]

### Basis change for Transformation Matrix

One of the key applications of linear transformations is that it can transform one coordinate system to a new coordinate system, since, recall, a coordinate system is defined by its basis vectors. Then, a transformation does what it says: it "transforms" (or converts) one set of basis vectors to another set of basis vectors.
By way of example: suppose we have two basis vectors, $\hat{i}$ and $\hat{j}$, which are unit basis vectors in a grid. Now, suppose we 'transform' the grid by rotating this grid 90° counterclock. Hence, $\hat{i}$ goes from $\left[\begin{smallmatrix}1\\0\end{smallmatrix}\right]$ to $\left[\begin{smallmatrix}0\\1\end{smallmatrix}\right]$, and $\hat{j}$ goes from $\left[\begin{smallmatrix}0\\1\end{smallmatrix}\right]$ to $\left[\begin{smallmatrix}-1\\0\end{smallmatrix}\right]$

[INSERT IMAGE]

We can write these new basis vectors as a matrix: $\left[\begin{smallmatrix}0 & -1\\1 & 0\end{smallmatrix}\right]$. Let's call this matrix **A**. Furthermore, we can consider the columns of **A** as the axes of the new coordinate system, the one that was just transformed by 90°. Put differently, these columns are the "transformed version" of the original basis vectors. Now, also recall that any vector can be expressed as a combination of the unit vectors, where the components of the vector form the coefficients of the combination, as follows:

$$\mathbf{v} = v_1\hat{i} + v_2\hat{j}$$

OK - now, let $\mathbf{v} = \left[\begin{smallmatrix}2\\3\end{smallmatrix}\right]$, and continuing with our example above, then we have:

$$\mathbf{v} = 2\begin{bmatrix}1\\0\end{bmatrix} + 3\begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}2\\3\end{bmatrix}$$

Then, we want to know where $\left[\begin{smallmatrix}2\\3\end{smallmatrix}\right]$ 'lands' in our new grid (the one we rotated by 90°), which we can easily do with our transformation matrix, as follows:

$$\begin{bmatrix}0 & -1\\1 & 0\end{bmatrix}\begin{bmatrix}2\\3\end{bmatrix} = \begin{bmatrix}-3\\2\end{bmatrix}$$

Hence, by rotating our grid by 90°, our vector $\left[\begin{smallmatrix}2\\3\end{smallmatrix}\right]$ gets 'transformed' to $\left[\begin{smallmatrix}-3\\2\end{smallmatrix}\right]$, by means of the *transformation matrix*, $\mathbf{A}$.

**Heuristic 8**

**Consider a matrix as a transformation of space**

### Determinant of a transformation

As we have seen, linear transformations are able to stretch an area in space or conversely compress, i.e. squish, an area. However, to understand these transformations well, we need to have an understanding of the magnitude, that is a *factor*, by which these transformations are being *scaled*. Put differently, we want to be able to derive such a *scaling factor*, which is a measure by which a given area increases or decreases when a linear transformation is applied. More formally, suppose we have a linear transformation $T : \mathbb{R}^2 \to \mathbb{R}^2$ where $\mathbf{A}_T$ denotes the transformation matrix. Then,

> The determinant of $\mathbf{A}_T$ is the *scale factor* associated with the linear transformation $T$ and describes how the area of a unit square (a $1 \times 1$ square) is transformed by $T$.

Should $\det(\mathbf{A}_T) < 1$, then T "shrinks" the area, with $\det(\mathbf{A}_T) > 1$ then T "expands" the area. An important case is when $\det(\mathbf{A}_T) = 0$, where the two axes of the area come to lie on top of each other, or indeed can converge to a single point.
Hence, determining how the area of one transformation changes will allow us to determine how **any** area in space changes, since we are changing the coordinate system and grid lines remain parallel and evenly spread.

### 2.3.6 Eigenvalues and Eigenvectors

Now that we are familiar with matrices and the linear transformations that they can represent, as *transformation matrices*, we are able to learn a new method to represent these linear transformations and associated transformation matrices by performing an *eigen* analysis. That is, we can learn a new method to represent linear transformations by investigating the specific properties of a transformation matrix. As such, we will find a new way to characterize a matrix and its associated linear mapping.
To explain eigenvalues, we must first explain eigenvectors. Suppose we have some matrix $\mathbf{A}$, now, unless $\mathbf{A}$ is the identity matrix, or some scalar version of it, then all vectors change direction when they are multiplied by $\mathbf{A}$. i.e. $\mathbf{Az} = \mathbf{y}$ such that the direction of $\mathbf{z}$ and $\mathbf{y}$ are different. However, certain exceptional vectors, $\mathbf{x}$, are in the same direction as $\mathbf{Ax}$. Those are the "eigenvectors". Multiply an eigenvector by $\mathbf{A}$, and the result is a number, $\lambda$, times the original $\mathbf{x}$, as follows: $\mathbf{Ax} = \mathbf{y} = \lambda\mathbf{x}$. This scalar value, $\lambda$, is called the *eigenvalue* and tells us whether the eigenvector, $\mathbf{x}$, is stretched, shrunk, reversed or left unchanged, when multiplied by $\mathbf{A}$. Note, however, that if $\mathbf{A}$ is the identity matrix, $\mathbf{I}$, then **every** vector is an eigenvector of $\mathbf{I}$. Eigenvectors and eigenvalues are ubiquitous, and indeed, the Google search algorithm (PageRank) is largely based on it. For our purposes, within the machine learning arena, the main use case of eigenvectors and eigenvalues is their application in Matrix decomposition, which we will explore in detail later on.

The formal definition goes as follows:

*"Eigen"* is german for "self", "characteristic"

**Definition 2.3.9.** Let $\mathbf{A} \in \mathbb{R}^{n\times n}$. Then, $\lambda \in \mathbb{R}$ is an **eigenvalue** of $\mathbf{A}$ if there is a nonzero vector $\mathbf{x} \in \mathbb{R}^n$ such that $\mathbf{Ax} = \lambda\mathbf{x}$. We say that $\mathbf{x}$ is the **eigenvector** corresponding to the eigenvalue $\lambda$.

#### Properties of Eigenvalues and Eigenvectors

- If $\mathbf{A}$ is an $n \times n$ triangular matrix (upper, lower or diagonal), then the eigenvalues of $\mathbf{A}$ are the entries on the main diagonal of $\mathbf{A}$.
- If $\lambda$ is an eigenvalue of $\mathbf{A}$, then $\lambda$ is an eigenvalue of $\mathbf{A}^T$
- If $\lambda$ is an eigenvalue of $\mathbf{A}$ with eigenvector $\mathbf{x}$, then $\frac{1}{\lambda}$ is an eigenvalue of $\mathbf{A}^{-1}$, with eigenvector $\mathbf{x}$
- The sum of the eigenvalues of $\mathbf{A}$ equals the trace of $\mathbf{A}$
- The product of the eigenvalues of $\mathbf{A}$ equals the determinant of $\mathbf{A}$
- The eigenvectors $\mathbf{x}_1, \cdots, \mathbf{x}_n$ of a matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ with $n$ distinct eigenvalues $\lambda_1 \cdots \lambda_n$ are **linearly independent**

### Computing Eigenvalues and Eigenvectors

All very well, but how do we derive the eigenvectors and eigenvalues of a matrix?

### Eigenvalues

We just learned the eigenvalue equation, as: $\mathbf{Ax} = \lambda\mathbf{x}$, then to find the eigenvalues, we first insert the identity matrix, $\mathbf{I}$ and rewrite the equation to solve for 0, that is as a *null-space problem*:

$$\mathbf{Ax} = \lambda\mathbf{Ix} \implies (\mathbf{A} - \lambda\mathbf{I})\,\mathbf{x} = 0$$

Now, if $(\mathbf{A} - \lambda\mathbf{I}) = 0$ has a non-zero solution (non-trivial solution), then this implies that $\mathbf{A} - \lambda\mathbf{I}$ is not invertible, and therefore $\mathbf{A} - \lambda\mathbf{I}$ must be *singular*, hence $\det(\mathbf{A} - \lambda\mathbf{I}) = 0$

**How does that work?**

Suppose $\mathbf{A}$ did have an inverse, then:

$$\mathbf{A}^{-1}\mathbf{Ax} = \mathbf{A}^{-1}0 \implies \mathbf{Ix} = 0 \implies \mathbf{x} = 0,$$

but $\mathbf{x}$ cannot be a zero vector, hence $\mathbf{A}$ does not have an inverse, hence is singular and $\det(\mathbf{A})$ must be zero.

Okay, great, so now we must solve for $\det(\mathbf{A} - \lambda\mathbf{I}) = 0$ , which results in solving the *characteristic polynomial* such to find the values for $\lambda$, which are indeed the eigenvalues of $\mathbf{A}$.

For our purposes here, we let $\mathbf{A} \in \mathbb{R}^2$, such that:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$$

Now, recall from our work with determinants, computing the determinant of a $2 \times 2$ matrix is simply: $a_{11}a_{22} - a_{21}a_{12}$. The expression we need to solve is $\mathbf{A} - \lambda\mathbf{I}$, which we write out as:

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} - \begin{bmatrix} \lambda & 0 \\ 0 & \lambda \end{bmatrix} = \begin{bmatrix} a_{11} - \lambda & a_{12} \\ a_{21} & a_{22} - \lambda \end{bmatrix}$$

Then,

$$\begin{aligned} \det\left(\begin{bmatrix} a_{11} - \lambda & a_{12} \\ a_{21} & a_{22} - \lambda \end{bmatrix}\right) &= (a_{11} - \lambda)(a_{22} - \lambda) - a_{21}a_{22} \\ &= \lambda^2 - (a_{22} + a_{11})\lambda + a_{11}a_{22} - a_{21}a_{22} \ \textit{(characteristic polynomial)} \end{aligned}$$

The roots of the characteristic polynomial represent the eigenvalues of $\mathbf{A}$. Since the degree of the polynomial is 2, we will of course find two eigenvalues; $\lambda_1$ and $\lambda_2$. Now that we have found the eigenvalues, we can determine the eigenvectors.

**Eigenvectors**

The eigenvectors are the "axes" around which an object gets transformed and do not change in direction. The other vectors of the object do change direction, but these other vectors are combinations of the two eigenvectors (or whatever the dimension of the object)
We recall the *eigenequation*: $\mathbf{Ax} = \lambda\mathbf{x}$. Suppose $\mathbf{x}$ is in $\mathbb{R}^2$, and we let $\mathbf{x} = (x_{\lambda_1}, x_{\lambda_2})$, then to find the eigenvectors associated with the eigenvalues $\lambda_1$ and $\lambda_2$, we need to solve for the components $x_{\lambda_1}$ and $x_{\lambda_2}$. That is:

$$\begin{bmatrix} a_{11} - \lambda & a_{12} \\ a_{21} & a_{22} - \lambda \end{bmatrix} \begin{bmatrix} x_{\lambda_1} \\ x_{\lambda_2} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

**Example**

Suppose we want to find the eigenvalues and eigenvectors of $\mathbf{A} = \begin{bmatrix} -5 & 2 \\ -9 & 6 \end{bmatrix}$

We kick it off by finding the eigenvalues $\lambda_1$ and $\lambda_2$ by setting up the characteristic polynomial by solving $\det(\mathbf{A} - \lambda\mathbf{I}) = 0$, that is:

$$\begin{aligned} \det\left(\begin{bmatrix} -5 & 2 \\ -9 & 6 \end{bmatrix} - \begin{bmatrix} \lambda & 0 \\ \lambda & 0 \end{bmatrix}\right) &= \begin{vmatrix} -5 - \lambda & 2 \\ -9 & 6 - \lambda \end{vmatrix} \\ &= (-5 - \lambda)(6 - \lambda) - (-9)(2) \\ &= -30 - \lambda + \lambda^2 + 18 \\ &= \lambda^2 - \lambda - 12 \\ &= (\lambda + 3)(\lambda - 4) = 0 \text{ if } \lambda = -3 \text{ or } 4 \end{aligned}$$

Then, the roots of the characteristic equation, $(\lambda + 3)(\lambda - 4) = 0$ are $\lambda = -3$ and $\lambda = 4$ and these are the eigenvalues of $\mathbf{A}$.
Next, we find the eigenvectors of $\mathbf{A}$ by substituting the eigenvalues in the two equations given by $(\mathbf{A} - \lambda\mathbf{I})$

For $\lambda = -3$:

$$\begin{bmatrix} -5 - (-3) & 2 \\ -9 & 6 - (-3) \end{bmatrix} \begin{bmatrix} x_{\lambda_1} \\ x_{\lambda_2} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Giving us:

$$\begin{aligned} -2x_{\lambda_1} + 2x_{\lambda_2} &= 0 \\ -9x_{\lambda_1} + 9x_{\lambda_2} &= 0 \end{aligned}$$

Solving for this homogeneous system will give us infinite solutions, where $x_{\lambda_1} = x_{\lambda_2}$, and we choose a convenient value for $x_{\lambda_1} = 1$, which gives $x_{\lambda_2} = 1$ as well, and obtain the solution space, or

*eigenspace*: $E_{(-3)} = \text{span}[\begin{bmatrix}1\\1\end{bmatrix}]$

For $\lambda = 4$, we solve equivalently and obtain the system of equations:

$$-9x_{\lambda_1} + 2x_{\lambda_2} = 0$$
$$-9x_{\lambda_1} + 2x_{\lambda_2} = 0$$

And choose $x_{\lambda_1} = 2$, giving us $x_{\lambda_2} = 9$, obtaining the eigenvector for $\lambda = 4$ as $\begin{bmatrix}2\\9\end{bmatrix}$

We have now found the two eigenvalues with their corresponding eigenvectors. If we had a $3 \times 3$ matrix then we would have found 3 eigenvalues and 3 corresponding eigenvectors. And so in general, a $n \times n$ matrix will produce $n$ eigenvalues and $n$ corresponding eigenvectors.

**Eigenvalues & Eigenvectors in NumPy**
Computing all of this in `NumPy` could not be easier:

```
A = np.array([[-5, 2], [-9, 6]])
eigvals, eigvecs = np.linalg.eig(A)


print("lambda_1  = ", eigvals[0], "\nlambda_2  = ", eigvals[1])
print("Eigenvector corresponding to lambda_1: ",'\n', eigvecs[:,0].reshape(2,1))
print("Eigenvector corresponding to lambda_2: ",'\n', eigvecs[:,1].reshape(2,1))
```

results in:

```
lambda_1  =  -3.0
lambda_2  =  4.0
Eigenvector corresponding to lambda_1:
 [[-0.70710678]
 [-0.70710678]]
Eigenvector corresponding to lambda_2:
 [[-0.21693046]
 [-0.97618706]]
```

**Eigenvalues and eigenvectors have some interesting properties.**

- If $\mathbf{A}$ is an $n \times n$ triangular matrix (upper, lower or diagonal), then the eigenvalues of $\mathbf{A}$ are the entries on the main diagonal of $\mathbf{A}$.
- If $\mathbf{A}$ is an $n \times n$ matrix, the following statements are equivalent:
  1. $\lambda$ is an eigenvalue of $\mathbf{A}$.
  2. the system of equations $(\lambda\mathbf{I} - \mathbf{A})\,\mathbf{x} = 0$ has nontrivial solutions.
  3. $\lambda$ is a solution of the characteristic equation $\det(\lambda\mathbf{I} - \mathbf{A})$

### 2.3.7 Eigenspace and Eigenbasis

**Eigenspace**
Once we have found the set of all eigenvectors of a matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ which are associated with an eigenvalue $\lambda$, then we say that this set spans a subspace of $\mathbb{R}^n$ which is called the *eigenspace* of $\mathbf{A}$, and is denoted by $E_\lambda$. In the example above, the eigenspace associated with $\lambda = 4$, is:

$$E_{\lambda=4} = \texttt{span}[\begin{bmatrix} -0.21693046 \\ -0.97618706 \end{bmatrix}]$$

Now, recall our work on the *null space* of a matrix? It is a set of vectors which a matrix maps onto the zero vector. Well, also recall that we had to solve for $(\mathbf{A} - \lambda_i \mathbf{I})\mathbf{x} = 0$, and called this the *null-space problem*. Then, the set of vectors, $E_{\lambda i}$ which the matrix $(\mathbf{A} - \lambda_i \mathbf{I})$ maps onto the zero vector is its *null space*. Hence, the eigenspace is the null space.

**Eigenbasis**

An eigenbasis is simply a basis for a particular vector space where every vector is an eigenvector. Formally, Let A be an $n \times n$ matrix. Recall that a nonzero vector $\mathbf{v}$ is an eigenvector for A with eigenvalue $\lambda$ if $\mathbf{A}\mathbf{v} = \lambda\mathbf{v}$. Then, an **eigenbasis** is a basis of $\mathbb{R}^n$ consisting of eigenvectors of A. For our purposes, however, the key role that the eigenbasis will play is it will allow us to *diagonalize* a matrix, which will open up all kinds of applications in Machine Learning.

### 2.3.8 Matrices & Orthogonality

Orthogonality of matrices forms an important method in Data Mining. As not every data set is born equal; we can generally divide data quality in linear algebra terms as *good basis vectors* and *bad basis vectors*, that is: close to orthogonal basis vectors versus basis vectors that are almost linearly dependent. Recall, two vectors are orthogonal when the cosine of their angle is zero. Some important definitions and results follow.

- Let $\mathbf{u}, \mathbf{v}$ be nonzero vectors then $\mathbf{u}$ and $\mathbf{v}$ are *orthogonal* if their Dot product is zero, then $\mathbf{u}^T\mathbf{v} = 0$.
- A set of orthogonal vectors is *orthonormal* if every vector has norm of 1.
- Let $\mathbf{u}, \mathbf{v}$ be nonzero orthogonal vectors. Then they are linearly independent

Then, the definition of *Orthogonal matrix* naturally follows from the vectors.

**Definition 2.3.10.** A square matrix $\mathbf{Q} \in \mathbb{R}^{m \times m}$ whose columns are orthonormal is called an **orthogonal matrix**. In other words, $\mathbf{Q} \in \mathbb{R}^{m \times m}$ is orthogonal if and only if $\mathbf{Q}^T\mathbf{Q} = \mathbf{I}$, or, equivalently, $\mathbf{Q}^T = \mathbf{Q}^{-1}$ Furthermore, the set of orthogonal $m \times m$ matrices is defined as

$$O_m = \{\mathbf{Q} \in \mathbb{R}^{m \times m} : \mathbf{Q}^T\mathbf{Q} = \mathbf{I}\}$$

Orthogonal matrices have a number of important properties:

1. The product of two orthogonal matrices is orthogonal

2. The Euclidean norm of a vector is invariant under an orthogonal transformation $\mathbf{Q}$. That is: $\| \mathbf{Qv} \| = \| \mathbf{v} \|$ for all $\mathbf{v} \in \mathbb{R}^n$ if and only if $\mathbf{Q}$ is orthogonal.

3. Every orthogonal matrix is invertible

**How does that work?**

$$\begin{aligned} \text{as per the definition} &\longrightarrow \mathbf{Q}^T = \mathbf{Q}^{-1} \\ \text{taking transpose} &\longrightarrow (\mathbf{Q}^T)^T = (\mathbf{Q}^{-1})^T \\ \text{by properties of transpose} &\longrightarrow \mathbf{Q} = (\mathbf{Q}^T)^{-1} \end{aligned}$$

Hence, there exists an invertible for every orthogonal matrix

## 2.4 Matrix Decomposition

As we saw earlier, one of the key pillars of machine learning is dimensionality reduction. In the following section we will study the core principles and techniques that lie at this foundation, namely *matrix decomposition* and in the process learn one of the most fundamental theorems in linear algebra: *singular value decomposition.*

Similar to numbers that can be broken up into their 'factors', such as $6 = 2 \times 3$, we can express a matrix as the product of simpler matrices. Hence, matrix decomposition is often referred to as matrix factorisation. This in turn will often allow for greatly simpler, and indeed less memory consuming, computations.
We will limit the text to the essential decompositions:

1. Eigendecomposition
2. Singular value decomposition

Which are both of essence when performing dimensionality reduction.

### 2.4.1 Eigendecomposition

Eigendecomposition, or sometimes also referred to as "Spectral decomposition", recasts a matrix in terms of its eigenvalues and eigenvectors.

#### Diagonalization

On our path to matrix decompositions we define the concept of *diagonalization.* The specific information gained from eigenvalues and eigenvectors of a matrix allows us to determine a factorisation of this particular matrix such that one of the elements of this product will be a diagonal matrix. Put differently, we can transform $\mathbf{A}$ into a diagonal matrix: $\mathbf{D}$. Diagonal matrices are the equivalent of Lego blocks: easy to work and build with. Furthermore, once we have established the similarity between $\mathbf{A}$ and $\mathbf{D}$ we can make use of the *similarity invariant.* We define two matrices to be similar as follows:

> **Definition 2.4.1.** Let $\mathbf{A}$ and $\mathbf{B}$ be square matrices, then we say that **$\mathbf{B}$ is similar to $\mathbf{A}$** if there exists an invertible matrix $\mathbf{P}$ such that $\mathbf{B} = \mathbf{P}^{-1}\mathbf{A}\mathbf{P}$.

Then the similarity invariant tells us that any property of $\mathbf{A}$ is shared by $\mathbf{D}$. We need to state an important theorem:

> **Theorem 2.4.2. The Diagonalization Theorem**. An $n \times n$ matrix $\mathbf{A}$ is diagonalizable if and only if $\mathbf{A}$ has $n$ linearly independent eigenvectors. Furthermore, $\mathbf{A} = \mathbf{P}\mathbf{D}\mathbf{P}^{-1}$ where $\mathbf{D}$ is a diagonal matrix, if and only if the columns of $\mathbf{P}$ are $n$ linearly independent eigenvectors of $\mathbf{A}$. The diagonal entries of $\mathbf{D}$ are eigenvalues of $\mathbf{A}$ that correspond to the respective eigenvectors in $\mathbf{P}$.
> **Then** it follows that $\mathbf{A}$ is diagonalizable if and only if $\mathbf{A}$ has $n$ linearly independent eigenvectors. Equivalently; if $\mathbf{A}$ has $n$ distinct eigenvalues, then $\mathbf{A}$ is diagonalizable.

In other words: $\mathbf{A}$ is diagonalizable if and only if there are enough eigenvectors to form a basis of $\mathbb{R}^n$. Such a basis is called an **eigenvector basis** of $\mathbb{R}^n$.

As such we can prove that a real symmetric matrix $\mathbf{A}$ can be *orthogonally diagonalized*. Formally we define this by the following theorem:

**Theorem 2.4.3.** Let $\mathbf{A}$ be a real symmetric $n \times n$ matrix with eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_n$ and corresponding orthonormal eigenvectors

$$\mathbf{A} = \mathbf{QDQ}^T$$

Where $\mathbf{Q} = (\ \mathbf{q}_1\ \mathbf{q}_2\ \cdots\ \mathbf{q}_n)$ and $\mathbf{D}$ is a diagonal matrix with the eigenvalues of $\mathbf{A}$ along the diagonal of $\mathbf{D}$
By multiplying out we obtain:

$$\mathbf{A} = \lambda_1 \mathbf{q}_1 \mathbf{q}_1^T + \lambda_2 \mathbf{q}_2 \mathbf{q}_2^T + \cdots + \lambda_n \mathbf{q}_n \mathbf{q}_n^T = \sum_{i=1}^{n} \lambda_i \mathbf{q}_i \mathbf{q}_i^T,$$

which is the **spectral decompositon** of $\mathbf{A}$.

**How does that work?**

Since $\mathbf{Q}$ consists of orthonormal vectors $(\mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_n)$ then by the property of orthogonality we have that $\mathbf{Q}^T\mathbf{q}_i = \mathbf{e}_i$. Then for any $i$ we have that:

$$\mathbf{QDQ}^T\mathbf{q}_i = \mathbf{QDe}_i = \mathbf{Q}\lambda_i\mathbf{e}_i = \lambda_i\mathbf{Qe}_i = \lambda_i\mathbf{q}_i = \mathbf{Aq}_i$$

Hence: $\mathbf{QDQ}^T = \mathbf{A}$

We illustrate with a simple example; we let $\mathbf{A} = \mathbf{QDQ}^T$, with $\mathbf{A} = \begin{pmatrix} 3 & 2 \\ 2 & 3 \end{pmatrix}$. Then the characteristic equation for $\mathbf{A}$ is $\lambda^2 - 6\lambda + 5 = 0$, which factorizes to $(\lambda - 1)(\lambda - 5)$, hence, the eigenvalues of $\mathbf{A}$ are $\lambda_1 = 1, \lambda_2 = 5$. Solving for the eigenvectors we obtain: $\mathbf{v}_1 = \left(\begin{smallmatrix} 1 \\ -1 \end{smallmatrix}\right), \mathbf{v}_2 = \left(\begin{smallmatrix} 1 \\ 1 \end{smallmatrix}\right)$. Entries of $\mathbf{D}$ are the eigenvalues of $\mathbf{A}$, then $\mathbf{D} = \begin{pmatrix} 1 & 0 \\ 0 & 5 \end{pmatrix}$.
We form the orthonormal vectors, $\frac{1}{\|\mathbf{v}_i\|}\mathbf{v}_i$:

$$\tfrac{1}{\sqrt{2}}\begin{pmatrix} 1 \\ -1 \end{pmatrix}, \tfrac{1}{\sqrt{2}}\begin{pmatrix} 1 \\ 1 \end{pmatrix} \text{ and obtain } \mathbf{Q} = \begin{pmatrix} \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \\ \frac{-\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \end{pmatrix}.$$

Then the spectral decomposition of $\mathbf{A}$ is

$$\mathbf{A} = \begin{pmatrix} \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \\ \frac{-\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 5 \end{pmatrix} \begin{pmatrix} \frac{\sqrt{2}}{2} & \frac{-\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \end{pmatrix}$$

It follows from the definition that the linear transformation $\mathbf{A}$ maps vector $\mathbf{x}$ into the same direction.

### 2.4.2 Singular Value Decomposition

We have now come to the workhorse of Data Mining: Singular Value Decomposition (SVD). SVD is a method that allows us to build on the concept of matrix diagonalization to any matrix, square or not. In essence the SVD **factorizes a matrix into a product of two orthogonal matrices and a diagonal matrix with singular values as entries:**

$$\mathbf{A} = \mathbf{U}\Sigma\mathbf{V}^T$$

.
This is of course quite different from the 'standard' diagonalization; $\mathbf{D} = \mathbf{P^{-1}AP}$ since typically $\mathbf{U} \neq \mathbf{V}^T$. Whilst we can orthogonally diagonalize a matrix $\mathbf{A}$, there is no hope of doing so if $\mathbf{A}$ is not symmetric.
Furthermore, as we saw earlier, if we want to decompose a matrix in terms of its eigenvalues and eigenvectors, we again need to start with, or convert this matrix into, a diagonalizable one. None of these pesky constraints are applicable to SVD!
We can interpret the SVD from three mutually compatible perspectives:

1. We can see it as a method for transforming correlated variables into a set of uncorrelated ones that better expose the various relationships among the original data items.
2. SVD is a method for identifying and ordering the dimensions along which data points exhibit the most variation.
3. Once we have identified where the most variation is, it's possible to find the best approximation of the original data points using fewer dimensions

Hence, SVD can be seen as a method for dimensionality reduction. Indeed, an essential property of the Singular Value Decomposition is that it "orders" the information contained in the matrix such that the "dominating part" becomes visible. This is the property that makes the SVD so useful in data mining, as well as other areas.
Formally, we define SVD as follows:

**Theorem 2.4.4. Singular Value Decomposition**
Let $\mathbf{A} \in \mathbb{R}^{m\times n}$ be any matrix. Then there exist $\mathbf{U}$ as a $m \times m$ orthogonal matrix and $\mathbf{V}$ as a $n \times n$ orthogonal matrix and a diagonal matrix $\mathbf{\Sigma} \in \mathbb{R}^{m\times n}$ such that

$$\mathbf{A} = \mathbf{U\Sigma V}^T.$$

This is called the **singular value decomposition (SVD)** and the diagonal elements of $\mathbf{\Sigma}$ are called the **singular values** of $\mathbf{A}$.

Where the singular values are defined as

**Definition 2.4.5.** With $\mathbf{A} \in \mathbb{R}^{m\times n}$, then for $m > n$, the singular values, denoted by $\sigma_i$, are the positive square roots of the eigenvalues of the symmetric matrix $\mathbf{A}^T\mathbf{A}, \sigma_i = \sqrt{\lambda_i}$, since

$$\mathbf{QDQ}^T = \mathbf{A}^T\mathbf{A} = \left(\mathbf{U\Sigma V}^T\right)\mathbf{U\Sigma V}^T = \mathbf{V\Sigma}^T\mathbf{U}^T\mathbf{U\Sigma V}^T = \mathbf{V\Sigma}^T\mathbf{\Sigma V}^T.$$

Thus $\mathbf{Q} = \mathbf{V}$ and $\mathbf{D} = \mathbf{\Sigma}^T\mathbf{\Sigma}$.

An interesting perspective is to consider the singular values of $\mathbf{A}$ as the lengths of the vectors $\mathbf{Av}_i$ where $\mathbf{v}_i$ are the eigenvectors of $\mathbf{A}^T\mathbf{A}$.

### 2.4.3 Properties of SVD

We list a number of important corrolaries to the SVD. A more extensive list as well the proofs can be found in various textbooks on Linear Algebra.

Let $\mathbf{A} \in \mathbb{R}^{m \times n}$ with its SVD as $\mathbf{A} = \mathbf{U}\Sigma\mathbf{V}^T$, then the following are equivalent:

1. if $\mathbf{A} = \mathbf{U}\Sigma\mathbf{V}^T$ is the SVD of $\mathbf{A} \in \mathbb{R}^{m \times n}$, m $\geq$ n, then for $i = 1 \dots n$
$$\mathbf{A}\mathbf{v}_i = \sigma_i \mathbf{u}_i$$
2. if $\mathbf{A} \in \mathbb{R}^{m \times n}$, then
$$\|\mathbf{A}\| = \sigma_1, \|\mathbf{A}\|_F = \sqrt{\sigma_1^2 + \dots \sigma_p^2},$$
where $p = \min\{m, n\}$.
3. The SVD nicely characterizes the rank of a matrix and orthonormal bases for both its nullspace and its range; If $\mathbf{A}$ has $r$ positive singular values, then rank($\mathbf{A}$) $= r$ and
$$\text{null}(\mathbf{A}) = \text{span}\{\mathbf{v}_{r+1}, \dots, \mathbf{v}_n\}$$
$$\text{ran}(\mathbf{A}) = \text{span}\{\mathbf{u}_1, \dots, \mathbf{u}_r\}$$
4. If $\mathbf{A}$ has rank $r$, then it can be written as the sum of $r$ rank-1 matrices. If $\mathbf{A} \in \mathbb{R}^{m \times n}$ and rank($\mathbf{A}$) $= r$, then
$$\mathbf{A} = \sum_{i=1}^{r} \sigma_i \mathbf{u}_i \mathbf{v}_i^T$$

Finally, we note that the columns of $\mathbf{U}$ are called the *left singular vectors* of $\mathbf{A}$ and the columns of $\mathbf{V}$ as the *right singular vectors* of $\mathbf{A}$.

### 2.4.4 Low Rank Matrix Approximation

Often we will find that a data matrix contains a lot of data points which contribute little - they represent 'noise'. It follows that typically the evolution of the singular values shows a significant drop after a number of large singular values. Indeed, if the noise is sufficiently small the number of large singular values is often referred to as the *numercial rank* of the matrix. As such we can remove these data points and approximate $\mathbf{A}$ by a matrix of lower rank. This can be done by truncating the singular value expansion to the numerical rank of the matrix, say $k$. We have that the SVD can also be written as an expansion of the matrix multiplication, that is:

$$\mathbf{A} = \mathbf{U}\Sigma\mathbf{V}^T = \sum_{i=1}^{n} \sigma_i \mathbf{u}_i \mathbf{v}_i^T$$

then the truncated SVD:

$$\mathbf{A_k} = \sum_{i=1}^{k} \sigma_i \mathbf{u}_i \mathbf{v}_i^T$$

Finding the most optimal rank now becomes a minimization problem.
÷

## 2.5 Example: Term-Document Matrices

*Term-document matrices* are an example of a representation of data in matrix format: the rows are made up of key-words, called *terms* and the columns represent documents. Hence, the matrix describes the frequency of terms that occur in a set of documents.

Let's consider the following documents where the key words are in boldface.

Document 1: The **Google matrix P** is a model of the **Internet**.

Document 2: $\mathbf{P}_{ij}$ is nonzero if there is a **link** from **Web page** $j$ to $i$.

Document 3: The **Google matrix** is used to **rank** all **Web pages**.

Document 4: The **ranking** is done by solving a **matrix eigenvalue** problem.

Document 5: **England** dropped out of the top 10 in the **FIFA ranking**.

We can now count the frequency of the terms in each document to get the following matrix:

| Term | Doc 1 | Doc 2 | Doc 3 | Doc 4 | Doc 5 |
|---|---|---|---|---|---|
| eigenvalue | 0 | 0 | 0 | 1 | 0 |
| England | 0 | 0 | 0 | 0 | 1 |
| FIFA | 0 | 0 | 0 | 0 | 1 |
| Google | 1 | 0 | 1 | 0 | 0 |
| Internet | 1 | 0 | 0 | 0 | 0 |
| link | 0 | 1 | 0 | 0 | 0 |
| matrix | 1 | 0 | 1 | 1 | 0 |
| page | 0 | 1 | 1 | 0 | 0 |
| rank | 0 | 0 | 1 | 1 | 1 |
| Web | 0 | 1 | 1 | 0 | 0 |

As such, each document is represented by a vector in $\mathbb{R}^{10}$, and we organize all documents into a term-document matrix as follows:

$$\mathbf{A} = \begin{pmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 \end{pmatrix} \tag{2.8}$$

Such a matrix now conveniently allows us to search for terms in all the documents using a *query* vector (equivalent to search terms in an internet search), which is the process of finding the documents that are relevant to this particular query.
For instance, if we want to find all the documents that are relevant to the query **"ranking of Web pages"** we can construct the following vector:

$$\boldsymbol{q} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 1 \end{pmatrix} \in \mathbb{R}^{10} \tag{2.9}$$

Hence, the query is now also considered as a document, and we are considering this *information retrieval* problem as a which translated as a mathematical problem comes down to *'Find the columns of* $\mathbf{A}$ *that are close to the vector q'*. Hopefully, the "close to the vector" makes you think of cosine similarity and dot product! Putting it all together, we can now say that a document, say $\boldsymbol{a_j}$, is deemed relevant to our search if the angle between the query $\mathbf{q}$ and $\boldsymbol{a_j}$ is small enough, i.e. we get a high similarity score.

# Chapter 3

# Calculus for Machine Learning

## 3.1 Introduction

***'Events, dear boy, events'*** - Harold Macmillan, in reply when asked what had been the greatest influence on his administration when he was Prime Minister...

### 3.1.1 Why calculus?

Nothing is forever, stuff happens, things change, and it's the task of *calculus* to measure exactly that: change. More specifically, we observe that certain variables change as a result of a change in another, *independent*, variable. To understand and measure this change we need to know what this function does and looks like. Hence, **calculus is the study of functions and their properties.** The two main components of calculus are *differentiation* and *integration*, which, as usual in mathematics, are the two sides of the same coin.

Why do we care? We will deploy calculus in two main areas: optimization and probability. That is, many algorithms in machine learning, and indeed essential to *Deep Learning*, optimize an objective function (a.k.a loss function) with respect to a set of model parameters that control how well a model explains the data. Then to find these critical points we use *differential calculus*. As we will see, Gaussian distributions pop up in regression, dimensionality reduction and density estimation, where of course we need to measure probability distributions and the likes, for which we will deploy *integral calculus*.

**Heuristic 9**

**We use calculus to optimize loss functions and in probabilistic modeling.**

### 3.1.2 Limits

Essential in our understanding of derivatives, integrals and continuity, will be the concept of a *limit*. For a proper understanding of calculus it is critical to learn limits. However, as the purpose of this document is to focus only on the origin and working of the mathematics of machine learning, we

will only investigate the application of a limit with respect to the definition of a derivative.

As such, we use the concept of limits to describe what happens when a variable becomes very small, infinitely small even, or extremely large; infinitely large, and therefore we will not investigate what happens when a variable approaches a specific value.
Furthermore, whilst we will not go into explaining the concept of infinity, we do note that in a strict sense infinity is not a number, but instead it is a *process*. Think of any large number, $N$, well, you can always find a larger one by simply adding $+1$, and you can do this forever. In terms of notation, we denote infinitely large by $\infty$ (note that this can be either positive of negative). On the other side of the spectrum is an infinitely small number, which is usually denoted by either $\epsilon$ or $\delta$. By the similar reasoning, where $10^{-16}$ is a pretty small number, you can always find a $\epsilon < 10^{-16}$.

Then, to describe what happens when a variable becomes infinitely large, we write it as:

$$\lim_{n\to\infty} f(n)$$

Which is read as "in the limit as n goes to infinity, $f(n)$ goes to..."

Conversely, suppose a variable, say $\delta > 0$, becomes progressively smaller, we write:

$$\lim_{\delta\to 0} f(\delta)$$

And this is read as "in the limit as $\delta$ goes to zero, $f(\delta)$ goes to..."

It is this last version that we are interested in and will be using in difference quotient and the definition of a derivative.

## 3.2 Differentiation

An intuitive understanding of the concept of a derivative is by visualizing it as the measurement of the slope of a functions' graph.
Formally, we define a derivative as:

**Definition 3.2.1.** For $h > 0$, the *derivative* of $f$ at $x$ is defined as:

$$f'(x) \stackrel{\text{def}}{=} \lim_{h\to 0} \frac{f(x+h) - f(x)}{h}$$

Derivatives have been around for a while, and used a lot, and over the years many different notations have cropped up. The following are all equivalent:

$$Df(x) = f'(x) = \frac{d}{dx}f(x) = \frac{df}{dx} = \bigtriangledown f$$

As we have not gone into detail on how to do computations with limits we will not derive the derivatives from first principles and limits, but instead will restrict ourselves to the main derivative formula's and use these to construct the formulas for *composite* functions, with the work out of the *chain rule* as one of our main focus points.

**Derivative formulas**

**Power Rule**
Often, the function over which we want to compute the derivative is some polynomial, such as for instance $f(x) = 2x^{10} + 5$, as such we have that $x$ is being raised to some power, and the formula for computing derivatives of powers states that:

$$\text{if } f(x) = x^n \text{ then } f'(x) = nx^{n-1}$$

**Key derivative formulas**

| $f(x)$ | $\longrightarrow$ | $f'(x)$ |
|---|---|---|
| $a$ | $\longrightarrow$ | $0$ |
| $x$ | $\longrightarrow$ | $1$ |
| $x^n$ | $\longrightarrow$ | $nx^{n-1}$ |
| $\frac{1}{x} = x^{-1}$ | $\longrightarrow$ | $\frac{-1}{x^2} = -x^{-2}$ |
| $\sqrt{x}$ | $\longrightarrow$ | $\frac{1}{2\sqrt{x}}$ |

With these formulas in our backpocket, taking derivatives is a simple task: just apply the relevant formula from the table of derivatives to the specific problem at hand. However, obviously this only works for single functions, but what if we need to solve for some form of a combination of functions? In those circumstances we need to solve for *composite* functions, and apply the following rules:

**Linearity**
The derivative of a linear combination of functions is equal to the linear combination of the derivatives:

$$[\alpha f(x) + \beta g(x)]' = \alpha f'(x) + \beta g'(x)$$

**Product Rule**
The derivative of a product of two functions is the derivative of the first one multiplied by the second one plus the first one multiplied by the derivative of the second one, as follows:

$$[f(x)g(x)]' = f'(x)g(x) + f(x)g'(x)$$

**Quotient Rule**
The derivative of the quotient of two functions is obtained as follows:

$$\left[\frac{f(x)}{g(x)}\right]' = \frac{f'(x)g(x) - f(x)g'(x)}{g(x)^2}$$

**Chain Rule**
Many optimization tasks in machine learning will apply some form of a chain rule, which are expressions of the form where we have an inner and outer function, which we obtain as:

$$[f(g(x))] = f'(g(x))g'(x)$$

**Higher order derivatives**

We don't have to stop at computing only the derivative of a function, we can continue computing the derivative of the derivative of a function, and even the derivative of that derivative and so forth. Taking these derivatives of derivatives is what is called *higher order* derivatives.
For our purposes, we are foremost interested in these higher order derivatives such to compute the gradient of a function, and with that a *Hessian matrix*.

Let $f(x)$ be the original function, then the first derivative of that function, $f'(x)$ measures the *slope* of the function $f(x)$, whereas the second derivative of that function, denoted as $f''(x)$ contains information about the *curvature* of $f(x)$. As we will see later on when discussing the *backpropagation* algorithm, we are keen to understand the second derivatives of a set of functions. Where as we will see, computing the second derivative is simply a further application of the derivative rules that we saw earlier.

**Example**
Let $f(x) = x^2$, then $f'(x) = 2x$ and the second derivative of $f(x)$ is $f''(x) = 2$

**Gradient of functions**

So far we have studied the derivative of a function with only one variable, $x \in \mathbb{R}$. However, most models and applications in machine learning are determined by several variables. That is, a function $f$ depends on one or more variables $x \in \mathbb{R}^n$, i.e. $f(x) = f(x_1, x_2)$.
Then, the concept of the derivative of a function of several variables is called the *gradient* of a function. We obtain the gradient of a function, $f(x_1, x_2, \ldots, x_n)$, by varying one variable, i.e. $x_1, x_2, \ldots$ at a time and keeping the others constant. As such, the gradient is then a collection of the *partial derivatives* and by convention is noted as a row vector, which indeed introduces the area of *vector calculus*. Gradient of functions are essential to fascilitate learning in machine learning models as the gradient points in the direction of the steepest ascent, which we will show later on.

### Partial derivatives

Partial derivatives is like they suggest what they are: they only compute a part of the change in a function and is applicable to functions with multiple variables. Formally, we define partial derivatives as:

**Definition 3.2.2.** For $f : \mathbb{R}^n \to \mathbb{R}$ and $\mathbf{x} \in \mathbb{R}^n$ of $n$ variables $x_1, x_2, \ldots, x_n$, we define the *partial derivatives* as

note that $\mathbf{x}$ is a vector

$$\begin{aligned}\frac{\partial f}{\partial x_1} &= \lim_{h\to 0} \frac{f(x_1 + h, x_2, \ldots, x_n) - f(\mathbf{x})}{h} \\ &\vdots \\ \frac{\partial f}{\partial x_1} &= \lim_{h\to 0} \frac{f(x_1, x_2, \ldots, x_n + h) - f(\mathbf{x})}{h}\end{aligned}$$

Where we collect the partial derivatives in a row vector such that we obtain and use the following notation:

$$\nabla_x f = \text{grad} f = \frac{df}{d\mathbf{x}} = \begin{bmatrix}\frac{\partial f(\mathbf{x})}{\partial x_1} & \frac{\partial f(\mathbf{x})}{\partial x_2} & \cdots & \frac{\partial f(\mathbf{x})}{\partial x_n}\end{bmatrix}$$

And we call this row vector as the *gradient* of $f$. Hence, the gradient of say $f(x_1, x_2)$ is simply a (row) vector of its partial derivatives.

Often, the gradient vector will be written as a column vector. However, using notation of a row vector has the advantage that the gradient can be applied to vector-valued functions, and as such the gradient vector becomes a gradient matrix. Furthermore, a row vector notation allows for the immediate application of the multi-variate chain rule without paying attention to the dimension of the gradient.

### Vector-valued function

So far we have seen functions that map to real numbers, $f : \mathbb{R}^n \to \mathbb{R}$, where the elements in the domain can be *scalars* or vectors, but either way, the sausage that is being made is a real number. For instance, $f(\mathbf{x}) = f(x_1, x_2) = 3x_1^2 + x_2$, where $\mathbf{x} \in \mathbb{R}^2$, namely $\mathbf{x} = (x_1, x_2)$. We call these functions *scalar functions*. In other words, the co-domain of the function is the set of real numbers, $\mathbb{R}$. A *vector function*, however, is a function whose result or output is a vector quantity. The output is two-dimensional, three-dimensional or higher-dimensional. In other words, the co-domain of the function is $\mathbb{R}^2, \mathbb{R}^3, \mathbb{R}^n$.

Suppose now, that instead of a mapping to a real number, the function consists of a set of functions, i.e. a vector, where each function in that vector (i.e. the component in that vector) in turn maps onto a real number, as follows:

Let $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$ where $n \geqslant 1$ and $m > 1$ and a vector $\mathbf{x} = [x_1, x_2, \ldots, x_n]^T \in \mathbb{R}^n$, then the corresponding vector of function values is given by

$$\mathbf{f}(\mathbf{x}) = \begin{bmatrix} f_1(\mathbf{x}) \\ \vdots \\ f_n(\mathbf{x}) \end{bmatrix} \in \mathbb{R}^m$$

Hence, $\mathbf{f}$ is a vector of functions, $[f_1, f_2, \ldots, f_m]^T$, that map onto $\mathbb{R}$: $f_i : \mathbb{R}^n \to \mathbb{R}$.

Then naturally we will want to take the derivative of $\mathbf{f}$, which consists of taking the partial derivatives of each of the components of this vector of functions. Luckily for us, the differentiation rules for every $f_i$ are the same as the ones we have seen already. As such, we want to take the partial derivative of every function in that vector with respect to every $x_i \in \mathbb{R}$, where $i = 1$.

$$\frac{\partial \mathbf{f}(\mathbf{x})}{\partial x_i} = \begin{bmatrix} \frac{\partial f_i}{\partial x_i} \\ \vdots \\ \frac{\partial f_m}{\partial x_i} \end{bmatrix} = \begin{bmatrix} \lim_{h \to 0} \frac{f_1(x_1, x_2, \ldots, x_{i-1}, x_{i+h}, x_{i-1}, \ldots, x_n) - f_1(\mathbf{x})}{h} \\ \vdots \\ \lim_{h \to 0} \frac{f_m(x_1, x_2, \ldots, x_{i-1}, x_{i+h}, x_{i-1}, \ldots, x_n) - f_m(\mathbf{x})}{h} \end{bmatrix}$$

Note that for each component of $\mathbf{x}$, that is $x_i$, we obtain a **column** vector of the partial derivatives.

Now, computing the *gradient* of $\mathbf{f}$, that is $\nabla_{\mathbf{x}}\mathbf{f}$, with respect to a vector $\mathbf{x}$ (that is all the components, $x_i$) results in the **row** vector of the partial derivatives.

$$\nabla_{\mathbf{x}}\mathbf{f} = \begin{bmatrix} \frac{\partial \mathbf{f}(\mathbf{x})}{\partial x_i} \cdots \frac{\partial \mathbf{f}(\mathbf{x})}{\partial x_n} \end{bmatrix}$$

And of course, a row vector where the components are column vectors, gives us a matrix. Which brings us to the definition of a *Jacobian matrix*

**Definition 3.2.3.** The collection of all **first-order** partial derivatives of a vector-valued function $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$ is called the *Jacobian*, $\mathbb{J}$, which is an $m \times n$ matrix, defined as:

$$\begin{aligned} \mathbb{J} = \nabla_{\mathbf{x}}\mathbf{f} &= \begin{bmatrix} \frac{\partial \mathbf{f}(\mathbf{x})}{\partial x_i} \cdots \frac{\partial \mathbf{f}(\mathbf{x})}{\partial x_n} \end{bmatrix} \\ &= \begin{bmatrix} \frac{\partial f_1}{\partial x_i} & \cdots & \frac{\partial f_1}{\partial x_i} \\ \vdots & & \vdots \\ \frac{\partial f_m}{\partial x_n} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix} \end{aligned}$$

Should we want to take the second-order derivatives, then we will have computed the *Hessian matrix*

**Directional derivative**

Before we can continue to explain gradient descent, and it's critical role in machine learning, we first need to explain what directional derivatives are. Now that we have learned about partial derivatives, we can generalize this concept to the notion of directional derivative. As we have seen, a partial derivative of a function tells us what the instantaneous rates of change of that function is in the direction parallel to the coordinate axes, for instance $x$ and $y$. Suppose we take the partial derivatives of $f(x, y)$, then these tell us the rate of change of $f$ as we vary $x$, with $y$ constant and $y$, with $x$ constant, respectively. We then combine these partial derivatives in a vector and obtain the gradient. However, what to do if we allow both $x$ and $y$ to change simultaneously? How do we measure the rate of change of $f$ in that scenario? Well, *directional derivatives* allow us to compute the rates of change of a function in *any* direction. That is, the directional derivative takes into account the direction by which we change the variables. Furthermore, they allow us to find the direction where the change in the function is **maximised**. The directional derivative plays a critical role in our understanding of gradient descent (or ascent in this case).

Suppose we want to compute the instantaneous rate of change of a function $f(x, y)$ from a point $(x_0, y_0)$ in some direction. However, many vectors can describe such a direction and magnitude. So, to make our lives easier we will use a unit vector instead that has it's initial point at $(x_0, y_0)$ and that points in a particular direction.
Now, recall, our unit vector has a norm, or magnitude of 1. That is:

$$\|\mathbf{u}\| = \sqrt{u_1^2 + u_2^2} = 1$$

Then, the directional derivative of $f(x, y)$ in the direction of $\mathbf{u} = (u_1, u_2)$, is defined as:

$$D_u f(x_0, y_0) = \frac{\partial f}{\partial x_0} u_1 + \frac{\partial f}{\partial y_0} u_2$$

But wait! We have seen such an expression before, as it is simply the dot product of the partial derivatives with the components of the directional vector $\mathbf{u}$. Indeed, we can arrange terms and express this as follows:

$$D_u f(x_0, y_0) = \nabla_{\mathbf{x}} \mathbf{f} \cdot \mathbf{u}$$

And from this notation of the directional derivative, we can conclude the following theorem:

> **Theorem 3.2.4.** The maximum value of $D_u f(\mathbf{x})$ is given by the norm of the gradient of $f$, that is $\| \nabla f(\mathbf{x}) \|$, and which occurs when the gradient points in the direction of the steepest ascent.

**How does that work?**

We have just seen that the directional derivative of a function is given by the dot product of the gradient of that function and the unit vector that gives us the direction. Now, recall from our study of dot products that:

$$\mathbf{u} \cdot \mathbf{v} = \|\mathbf{u}\|\|\mathbf{v}\| \cos\theta$$

Then, applying this formula to our directional derivative expression we get:

$$\nabla_{\mathbf{x}}\mathbf{f} \cdot \mathbf{u} = \|\nabla_{\mathbf{x}}\mathbf{f}\|\|\mathbf{u}\| \cos\theta$$

And with the norm of a unit vector naturally being 1, we can rewrite as:

$$\nabla_{\mathbf{x}}\mathbf{f} \cdot \mathbf{u} = \|\nabla_{\mathbf{x}}\mathbf{f}\| \cos\theta$$

Now, $\theta$ is the angle between the gradient and $\mathbf{u}$, and recall: $\cos\theta$ ranges between $-1$ and 1. Then $\cos\theta = 1$ when $\theta = 0$. Hence, the maximum value occurs when the angle between the gradient and $\mathbf{u}$ is zero, or put differently, when $\mathbf{u}$ is pointing in the same direction as the gradient $\nabla\mathbf{f}(\mathbf{x})$.
As such, we have shown that the **gradient of a function points in the direction of steepest ascent.**

### Gradient descent

As stated at the beginning of this chapter, the two main use cases for calculus in machine learning are in optimization and probability. Where of course optimization is the task of either finding the minimum or maximum value of some function $f(\mathbf{x})$ by varying $\mathbf{x}$. Gradient descent is the optimization technique that powers modern neural networks and is tasked with finding the minimum of a defined loss function.
Since most objective functions are intended to be minimized, then finding the minimum value is equivalent to finding the valleys of the objective function. Now, we have just seen that gradients indeed point upwards, that is in the direction of the steepest ascent. Well, if instead, we now want to descent, all we have to do is to go into the opposite direction. Which, in terms of our dot product formula means the vectors of the gradient and the directional vector to lie in opposite directions. That is to minimize $\cos\theta$, which we'll obtain when $\cos\theta = -1$, that is $\theta = \pi$, which in turn gives us a negative gradient. This is the method known as **gradient descent or steepest descent**.

Almost all texts on machine learning give a detailed overview of the workings and applications of the gradient descent algorithm and we refer the interested reader to those texts.

## 3.3 Integration

Integral calculus finds many applications in machine learning, indeed, entire books consider machine learning from a probability perspective. In essence, data is often some noisy observation of a true underlying signal. Then, our aim is that by applying machine learning techniques we can identify the signal from the noise. For this to work, however, we need to develop a language that can quantify this "noise". That is, we want to find predictors that allow us to express this form of uncertainty, and it is this quantification of uncertainty that is mastered by *probability theory*. For instance, direct applications of probability in machine learning methods are *bayesian linear regression, density estimation* and many more methods.

In the first part of this Calculus chapter we investigated differentiation where we are given a function $f(x)$ and need to determine what the derivative of this function is. With Integral calculus, *integration*, we will turn the tables around such that when we are given a function, $g(x)$ and then ask which 'original' function $f(x)$ did we differentiate to get to $g(x)$?
The intuitive perspective that is often used is the tangent versus the area under the curve. Recall, with taking the derivative of a function, we construct a tangent line to that curve. Now, suppose we take several of these tangents, over say and interval from a to b. Then what is the area under the curve that this is equivalent to?

**By way of example:**
Suppose we have $f(x) = x^4 + 3x - 9$ as the result of a differentiation of the function $F(x)$. How did we get to this function? Well, we have three terms in $f(x)$, which we can unpack: the first term $x^4$ we got by differentiating where we drop the exponent by one and so we must have had $x^5$. This would imply we have $5x^4$, but instead we have a 1, so the 5 needs to cancel out when differentiating by multiplying by $\frac{1}{5}$. Hence, the first term of the 'original' function was $\frac{1}{5}x^5$. We apply the same approach to the other two terms and obtain:

recall the power rule of differentiation

$$F(x) = \frac{1}{5}x^5 + \frac{3}{2}x^2 - 9x$$

Except that it gets a bit tricky with constant terms. Recall, the derivative of a constant is zero. As such, $F(x)$ can have any constant term, since when differentiating $F(x)$ this term will disappear anyway, and therefore we must add a constant term, $c$ to $F(x)$ to get the final result:

$$F(x) = \frac{1}{5}x^5 + \frac{3}{2}x^2 - 9x + c$$

Now that we have seen an example, we can formalise a definition and terminology.

**Definition 3.3.1.** Given the function $f(x)$, we say that the function $F(x)$ is an **anti-derivative** of $f(x)$ such that

$$F'(x) = f(x)$$

We call the most general anti-derivative of $f(x)$ the **indefinite integral** and denote this as

$$\int f(x)dx = F(x) + c$$

We call $\int$ the **integral symbol**, $f(x)$ the **integrand**, $x$ the **integration variable** and $c$ being the constant of integration. As such, the process of finding the indefinite integral called **integration**.

When we are specific about the integration variable we will say that we are integrating $f(x)$ **with respect to** $x$

Then, using again our example from earlier, we get:

$$\int x^4 + 3x - dx = \frac{1}{5}x^5 - \frac{3}{2}x^2 - 9x + c$$

We can now easily see that the antiderivative of $f(x)$ is a function $F(x)$ whose derivative, $F'(x)$, equals $f(x)$, and indeed, as it is obvious from the name, the antiderivative performs the inverse operation of differentiation. This perspective comes in handy, since it implies that all the derivative formulas we have learned can be used in the opposite direction as integral formulas.

The integral is indefinite because we're performing an integral calculation but have not yet defined the *limits of integration*. Again, if the derivative is the tangent line to a curve, then the indefinite integral is the point underneath that tangent point.
However, when we are interested in for instance the cumulative probability then we must define between which two points of x we want this computation to be done. This is where the *area under the curve* comes into play.

**Area under the curve**

For our purposes, we will limit our use and interpretation of integral calculus in machine learning mainly to probability distributions and their functions. In fact, *Probability Theory* is the source of almost all of the integrals that appear in machine learning.
Recall our famous 'bell' curve, well, this is our *probability density function* (PDF). A PDF will tell us the probability of a *random variable* falling within a specific range of values. This probability is given by the area under the curve of the PDF. It is here where integral calculus works for us; it computes this area under the curve between two points of possible outcomes.

Such an integral describes the computation of the area under the curve of $f(x)$ between say $x = a$ and $x = b$. That is, if we let $A(a, b)$ denote the area under the curve between $a$ and $b$, then:

$$A(a, b) = \int_a^b f(x)dx$$

Where $a$ and $b$ are said to be the *limits of integration*, such that $a$ being the *lower limit*, providing the start of the integration, and $b$ the *upper limit*. As such, we have defined a **definite integral**.

And we can now relate the indefinite with the definite integral and the relationship with derivatives, through the *fundamental theorem of calculus*.

**Theorem 3.3.2.** Let $f(x)$ be a continuous function on $[a, b]$ and also suppose that $F(x)$ is any anti-derivative for $f(x)$, then:

$$\frac{d}{dx}F(x)|_a^b = f(x)$$

Furthermore,

$$\int_a^b f(x)dx = F(x)|_a^b = F(b) - F(a)$$

How to read and understand this theorem? Well, this theorem has two components: the first part of the theorem clearly establishes the relationship between differentiation and integration: it guarantees that any integrable - and *continuous* - function has an antiderivative. The second component argues that if we can find an antiderivative for the integrand (recall, that is $f(x)$), then we can evaluate the **definite** integral by evaluating the antiderivative at the limits of integration and simply subtracting both computations to obtain the area under the curve number.
As such, the fundamental theorem of calculus establishes an equivalence between the set of integral functions and the set of antiderivative functions, and furthermore illustrates that differentiation is indeed the inverse of integration.

**Heuristic 10**

**Integration reverses differentiation.**

Now that we have established that integration is the reverse of differentiation, we can of course also reverse the differentiation formulas that we saw earlier.

**Essential derivative & integration formulas**

| $f(x)$ | $\longrightarrow$ | $f'(x)$ | $\longrightarrow$ | $\int f(x)dx$ |
|---|---|---|---|---|
| $a$ | $\longrightarrow$ | $0$ | $\longrightarrow$ | $ax + c$ |
| $x$ | $\longrightarrow$ | $1$ | $\longrightarrow$ | $\frac{1}{2}x^2 + c$ |
| $x^n$ | $\longrightarrow$ | $nx^{n-1}$ | $\longrightarrow$ | $\frac{1}{n+1}x^{n+1} + c$ |
| $\frac{1}{x} = x^{-1}$ | $\longrightarrow$ | $\frac{-1}{x^2} = -x^{-2}$ | $\longrightarrow$ | $\ln x + c$ |
| $\sqrt{x}$ | $\longrightarrow$ | $\frac{1}{2\sqrt{x}}$ | $\longrightarrow$ | $\frac{2}{3}x^{\frac{3}{2}}$ |
| $e^x$ | $\longrightarrow$ | $e^x$ | $\longrightarrow$ | $e^x + c$ |

**Worked example of definite integral**

Naturally, when computing the definite integral we need to start with the indefinite integral, and can apply the formulas above. However, with definite integrals it becomes (even) easier, as we show here.
Suppose we have the following integral:

$$\int_0^2 x^2 + 1 dx$$

Then we start by computing the indefinite integral, but as we have shown earlier, we must include a constant. Hence, both of the following antiderivatives are valid:

$$F(x) = \frac{1}{3}x^3 + x \qquad \text{and} \qquad F(x) = \frac{1}{3}x^3 + x - \frac{18}{31}$$

Then, applying the Fundamental Theorem of Calculus, like we saw earlier, we can derive:

$$\begin{aligned}\int_0^2 x^2 + 1dx &= \left(\frac{1}{3}x^3 + x\right)\Big|_0^2 \\ &= \frac{1}{3}(2)^3 + 2 - \left(\frac{1}{3}(0)^3 + 0\right) \\ &= \frac{14}{3}\end{aligned}$$

Notice how we simply *evaluate* the indefinite integral at both limits of integration, namely 0 and 2. Now let's have a look at the second antiderivative:

$$\begin{aligned}\int_0^2 x^2 + 1dx &= \left(\frac{1}{3}x^3 + x - \frac{18}{31}\right)\Big|_0^2 \\ &= \frac{1}{3}(2)^3 + 2 - \frac{18}{31} - \left(\frac{1}{3}(0)^3 + 0 - \frac{18}{31}\right) \\ &= \frac{14}{3} - \frac{18}{31} + \frac{18}{31} \\ &= \frac{14}{3}\end{aligned}$$

Well, how convenient, the constant cancels out. So when computing definite integrals we do not need to bother with the constant!