https://colab.research.google.com/github/ageron/handson-ml3/blob/main/math_linear_algebra.ipynb#scrollTo=Q3qoVYXkeuYs

math_linear_algebra.ipynb

Vectors

## Definition

A vector is a quantity defined by a magnitude and a direction. For example, a rocket's velocity is a 3-dimensional vector: its magnitude is the speed of the rocket, and its direction is (hopefully) up. A vector can be represented by an array of numbers called *scalars*. Each scalar corresponds to the magnitude of the vector with regards to each dimension.

For example, say the rocket is going up at a slight angle: it has a vertical speed of 5,000 m/s, and also a slight speed towards the East at 10 m/s, and a slight speed towards the North at 50 m/s. The rocket's velocity may be represented by the following vector:

$$\textbf{velocity} = \begin{pmatrix} 10 \\ 50 \\ 5000 \end{pmatrix}$$

Note: by convention vectors are generally presented in the form of columns. Also, vector names are usually lowercase to distinguish them from matrices (which we will discuss below) and in bold (when possible) to distinguish them from simple scalar values such as $meters\_per\_second = 5026$.

A list of N numbers may also represent the coordinates of a point in an N-dimensional space, so it is quite frequent to represent vectors as simple points instead of arrows. A vector with 1 element may be represented as an arrow or a point on an axis, a vector with 2 elements is an arrow or a point on a plane, a vector with 3 elements is an arrow or a point in space, and a vector with N elements is an arrow or a point in an N-dimensional space… which most people find hard to imagine.

## Norm

The norm of a vector $\mathbf{u}$, noted $\|\mathbf{u}\|$, is a measure of the length (a.k.a. the magnitude) of $\mathbf{u}$. There are multiple possible norms, but the most common one (and the only one we will discuss here) is the Euclidian norm, which is defined as:

$\|\mathbf{u}\| = \sqrt{\sum_i \mathbf{u}_i}$ (square root of the sum of $\mathbf{u}_i$

That's the square root of the sum of all the squares of the components of $\mathbf{u}$. We could implement this easily in pure python, recalling that $\sqrt{x} = x$ *** 1/2*

## Zero, unit and normalized vectors

- A **zero-vector** is a vector full of 0s.
- A **unit vector** is a vector with a norm equal to 1.
- The **normalized vector** of a non-null vector $\mathbf{v}$, noted $\mathbf{v}$^, is the unit vector that points in the same direction as $\mathbf{v}$. It is equal to: $\mathbf{v}$^ $= \mathbf{v} \,/\, \|\mathbf{v}\|$

## Dot product

### Definition

The dot product (also called *scalar product* or *inner product* in the context of the Euclidian space) of two vectors $\mathbf{u}$ and $\mathbf{v}$ is a useful operation that comes up fairly often in linear algebra. It is noted $\mathbf{u}\cdot\mathbf{v}$, or sometimes $\langle\mathbf{u}|\mathbf{v}\rangle$ or $(\mathbf{u}|\mathbf{v})$, and it is defined as:

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

where $\theta$ is the angle between $\mathbf{u}$ and $\mathbf{v}$.

Another way to calculate the dot product is:

$$\mathbf{u}\cdot\mathbf{v}=\sum_i \mathbf{u}_i\times\mathbf{v}_i$$

### Main properties

- The dot product is **commutative**: $\mathbf{u}\cdot\mathbf{v}=\mathbf{v}\cdot\mathbf{u}$.
- The dot product is only defined between two vectors, not between a scalar and a vector. This means that we cannot chain dot products: for example, the expression $\mathbf{u}\cdot\mathbf{v}\cdot\mathbf{w}$ is not defined since $\mathbf{u}\cdot\mathbf{v}$ is a scalar and $\mathbf{w}$ is a vector.
- This also means that the dot product is **NOT associative**: $(\mathbf{u}\cdot\mathbf{v})\cdot\mathbf{w}\neq\mathbf{u}\cdot(\mathbf{v}\cdot\mathbf{w})$ since neither are defined.
- However, the dot product is **associative with regards to scalar multiplication**: $\lambda\times(\mathbf{u}\cdot\mathbf{v})=(\lambda\times\mathbf{u})\cdot\mathbf{v}=\mathbf{u}\cdot(\lambda\times\mathbf{v})$
- Finally, the dot product is **distributive** over addition of vectors: $\mathbf{u}\cdot(\mathbf{v}+\mathbf{w})=\mathbf{u}\cdot\mathbf{v}+\mathbf{u}\cdot\mathbf{w}$.

## Calculating the angle between vectors

One of the many uses of the dot product is to calculate the angle between two non-zero vectors. Looking at the dot product definition, we can deduce the following formula:

$$\theta = \arccos(\mathbf{u} \cdot \mathbf{v} \|\mathbf{u}\| \times \|\mathbf{v}\|)$$

Note that if $\mathbf{u} \cdot \mathbf{v} = 0$, it follows that $\theta = \pi 2$. In other words, if the dot product of two non-null vectors is zero, it means that they are orthogonal.

Let's use this formula to calculate the angle between $\mathbf{u}$ and $\mathbf{v}$ (in radians):

```
def vector_angle(u, v):
    cos_theta = u.dot(v) / LA.norm(u) / LA.norm(v)
    return np.arccos(cos_theta.clip(-1, 1))

theta = vector_angle(u, v)
print("Angle =", theta, "radians")
print("      =", theta * 180 / np.pi, "degrees")
```

```
Angle = 0.8685393952858895 radians
      = 49.76364169072618 degrees
```

Note: due to small floating point errors, `cos_theta` may be very slightly outside the $[-1,1]$ interval, which would make `arccos` fail. This is why we clipped the value within the range, using NumPy's `clip` function.

## Projecting a point onto an axis

The dot product is also very useful to project points onto an axis. The projection of vector $\mathbf{v}$ onto $\mathbf{u}$'s axis is given by this formula:

$$\mathbf{proj_u\, v} = (\mathbf{u} \cdot \mathbf{v} \,/\, \|\mathbf{u}\|^{**}2) \times \mathbf{u}$$

Which is equivalent to:

$$\mathbf{proj_u\, v} = (\mathbf{v} \cdot \mathbf{u}\text{^}) \times \mathbf{u}\text{^}$$

## Matrices

A matrix is a rectangular array of scalars (i.e. any number: integer, real or complex) arranged in rows and columns, for example:

$$\begin{bmatrix} 10 & 20 & 30 \\ 40 & 50 & 60 \end{bmatrix}$$

You can also think of a matrix as a list of vectors: the previous matrix contains either 2 horizontal 3D vectors or 3 vertical 2D vectors.

Matrices are convenient and very efficient to run operations on many vectors at a time. We will also see that they are great at representing and performing linear transformations such rotations, translations and scaling.

## Matrices in python

In python, a matrix can be represented in various ways. The simplest is just a list of python lists:

```
[[10, 20, 30], [40, 50, 60]]
```

A much more efficient way is to use the NumPy library which provides optimized implementations of many matrix operations:

```
A = np.array([
    [10,20,30],
    [40,50,60]
])
A
```

```
array([[10, 20, 30],
       [40, 50, 60]])
```

By convention matrices generally have uppercase names, such as $\boldsymbol{A}$.

In the rest of this tutorial, we will assume that we are using NumPy arrays (type `ndarray`) to represent matrices.

## Size

The size of a matrix is defined by its number of rows and number of columns. It is noted $rows \times columns$. For example, the matrix $A$ above is an example of a $2 \times 3$ matrix: 2 rows, 3 columns. Caution: a $3 \times 2$ matrix would have 3 rows and 2 columns.

## Element indexing

The number located in the $i_{th}$ row, and $j_{th}$ column of a matrix $X$ is sometimes noted $X_{i,j}$ or $X_{ij}$, but there is no standard notation, so people often prefer to explicitly name the elements, like this: "*let* $X = (x_{i,j})_{1 \le i \le m, 1 \le j \le n}$". This means that $X$ is equal to:

$$X = \begin{bmatrix} x_{1,1} & x_{1,2} & x_{1,3} & \cdots & x_{1,n} \\ x_{2,1} & x_{2,2} & x_{2,3} & \cdots & x_{2,n} \\ x_{3,1} & x_{3,2} & x_{3,3} & \cdots & x_{3,n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_{m,1} & x_{m,2} & x_{m,3} & \cdots & x_{m,n} \end{bmatrix}$$

However, in this notebook we will use the $X_{i,j}$ notation, as it matches fairly well NumPy's notation.

## Adding matrices

If two matrices $Q$ and $R$ have the same size $m \times n$, they can be added together. Addition is performed *elementwise*: the result is also an $m \times n$ matrix $S$ where each element is the sum of the elements at the corresponding position: $S_{i,j} = Q_{i,j} + R_{i,j}$

$$S = \begin{bmatrix} Q_{11} + R_{11} & Q_{12} + R_{12} & Q_{13} + R_{13} & \cdots & Q_{1n} + R_{1n} \\ Q_{21} + R_{21} & Q_{22} + R_{22} & Q_{23} + R_{23} & \cdots & Q_{2n} + R_{2n} \\ Q_{31} + R_{31} & Q_{32} + R_{32} & Q_{33} + R_{33} & \cdots & Q_{3n} + R_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ Q_{m1} + R_{m1} & Q_{m2} + R_{m2} & Q_{m3} + R_{m3} & \cdots & Q_{mn} + R_{mn} \end{bmatrix}$$

For example, let's create a $2 \times 3$ matrix $B$ and compute $A + B$:

## Scalar multiplication

A matrix $M$ can be multiplied by a scalar $\lambda$. The result is noted $\lambda M$, and it is a matrix of the same size as $M$ with all elements multiplied by $\lambda$:

$$\lambda M = \begin{bmatrix} \lambda \times M_{11} & \lambda \times M_{12} & \lambda \times M_{13} & \cdots & \lambda \times M_{1n} \\ \lambda \times M_{21} & \lambda \times M_{22} & \lambda \times M_{23} & \cdots & \lambda \times M_{2n} \\ \lambda \times M_{31} & \lambda \times M_{32} & \lambda \times M_{33} & \cdots & \lambda \times M_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \lambda \times M_{m1} & \lambda \times M_{m2} & \lambda \times M_{m3} & \cdots & \lambda \times M_{mn} \end{bmatrix}$$

A more concise way of writing this is:

$(\lambda M)_{i,j} = \lambda (M)_{i,j}$

In NumPy, simply use the `*` operator to multiply a matrix by a scalar. For example:

---

## Matrix multiplication

So far, matrix operations have been rather intuitive. But multiplying matrices is a bit more involved.

A matrix $Q$ of size $m \times n$ can be multiplied by a matrix $R$ of size $n \times q$. It is noted simply $QR$ without multiplication sign or dot. The result $P$ is an $m \times q$ matrix where each element is computed as a sum of products:

$$P_{i,j} = \sum_{k=1}^{n} Q_{i,k} \times R_{k,j}$$

The element at position $i,j$ in the resulting matrix is the sum of the products of elements in row $i$ of matrix $Q$ by the elements in column $j$ of matrix $R$.

$$P = \begin{bmatrix} Q_{11}R_{11} + Q_{12}R_{21} + \cdots + Q_{1n}R_{n1} & Q_{11}R_{12} + Q_{12}R_{22} + \cdots + Q_{1n}R_{n2} & \cdots & Q_{11}R_{1q} + Q_{12}R_{2q} + \cdots + Q_{1n}R_{nq} \\ Q_{21}R_{11} + Q_{22}R_{21} + \cdots + Q_{2n}R_{n1} & Q_{21}R_{12} + Q_{22}R_{22} + \cdots + Q_{2n}R_{n2} & \cdots & Q_{21}R_{1q} + Q_{22}R_{2q} + \cdots + Q_{2n}R_{nq} \\ \vdots & \vdots & \ddots & \vdots \\ Q_{m1}R_{11} + Q_{m2}R_{21} + \cdots + Q_{mn}R_{n1} & Q_{m1}R_{12} + Q_{m2}R_{22} + \cdots + Q_{mn}R_{n2} & \cdots & Q_{m1}R_{1q} + Q_{m2}R_{2q} + \cdots + Q_{mn}R_{nq} \end{bmatrix}$$

You may notice that each element $P_{i,j}$ is the dot product of the row vector $Q_{i,*}$ and the column vector $R_{*,j}$:

$$P_{i,j} = Q_{i,*} \cdot R_{*,j}$$

So we can rewrite $P$ more concisely as:

$$P = \begin{bmatrix} Q_{1,*} \cdot R_{*,1} & Q_{1,*} \cdot R_{*,2} & \cdots & Q_{1,*} \cdot R_{*,q} \\ Q_{2,*} \cdot R_{*,1} & Q_{2,*} \cdot R_{*,2} & \cdots & Q_{2,*} \cdot R_{*,q} \\ \vdots & \vdots & \ddots & \vdots \\ Q_{m,*} \cdot R_{*,1} & Q_{m,*} \cdot R_{*,2} & \cdots & Q_{m,*} \cdot R_{*,q} \end{bmatrix}$$

## Matrix transpose

The transpose of a matrix $M$ is a matrix noted $M^T$ such that the $i^{th}$ row in $M^T$ is equal to the $i^{th}$ column in $M$:

$A^T=[104020503060]^T=\left[\left[102030405060\right]\right]$

In other words, $(A^T)_{i,j} = A_{j,i}$

Obviously, if $M$ is an $m \times n$ matrix, then $M^T$ is an $n \times m$ matrix.

Note: there are a few other notations, such as $M^t$, $M'$, or ${}^tM$.

In NumPy, a matrix's transpose can be obtained simply using the `T` attribute:

Transposition is distributive over addition of matrices, meaning that $(Q+R)^T=Q^T+R^T$.

## Converting 1D arrays to 2D arrays in NumPy

As we mentioned earlier, in NumPy (as opposed to Matlab, for example), 1D really means 1D: there is no such thing as a vertical 1D-array or a horizontal 1D-array. So you should not be surprised to see that transposing a 1D array does not do anything:

## Matrix inverse

Now that we understand that a matrix can represent any linear transformation, a natural question is: can we find a transformation matrix that reverses the effect of a given transformation matrix $F$? The answer is yes… sometimes! When it exists, such a matrix is called the **inverse** of $F$, and it is noted $F^{-1}$.

Only square matrices can be inversed. This makes sense when you think about it: if you have a transformation that reduces the number of dimensions, then some information is lost and there is no way that you can get it back. For example say you use a $2 \times 3$ matrix to project a 3D object onto a plane.

This transformation matrix performs a projection onto the horizontal axis. Our polygon gets entirely flattened out so some information is entirely lost, and it is impossible to go back to the original polygon using a linear transformation. In other words, $F_{project}$ has no inverse. Such a square matrix that cannot be inversed is called a **singular matrix** (aka degenerate matrix)

As you might expect, the dot product of a matrix by its inverse results in the identity matrix:

$$M \cdot M_{-1} = M_{-1} \cdot M = I$$

This makes sense since doing a linear transformation followed by the inverse transformation results in no change at all.

Also, the inverse of scaling by a factor of $\lambda$ is of course scaling by a factor of $1\lambda$:

$$(\lambda \times M)_{-1} = 1\lambda \times M_{-1}$$

Once you understand the geometric interpretation of matrices as linear transformations, most of these properties seem fairly intuitive.

A matrix that is its own inverse is called an **involution**. The simplest examples are reflection matrices, or a rotation by 180°, but there are also more complex involutions, for example imagine a transformation that squeezes horizontally, then reflects over the vertical axis and finally rotates by 90° clockwise. Pick up a napkin and try doing that twice: you will end up in the original position. Here is the corresponding involutory matrix:

## Determinant

The determinant of a square matrix $M$, noted $\det(M)$ or $\det M$ or $|M|$ is a value that can be calculated from its elements $(M_{i,j})$ using various equivalent methods. One of the simplest methods is this recursive approach:

$$|M| = M_{1,1} \times |M_{(1,1)}| - M_{1,2} \times |M_{(1,2)}| + M_{1,3} \times |M_{(1,3)}| - M_{1,4} \times |M_{(1,4)}| + \cdots \pm M_{1,n} \times |M_{(1,n)}|$$

- Where $M_{(i,j)}$ is the matrix $M$ without row $i$ and column $j$.

For example, let's calculate the determinant of the following $3 \times 3$ matrix:

$$M = \left[\left|\, 147258360 \,\right]\right|$$

Using the method above, we get:

$$|M| = 1 \times |[56\ 80]| - 2 \times |[46\ 70]| + 3 \times |[45\ 78]|$$

Now we need to compute the determinant of each of these $2 \times 2$ matrices (these determinants are called **minors**):

$$|||[5860]|||=5\times0-6\times8=-48$$

$$|||[4760]|||=4\times0-6\times7=-42$$

$$|||[4758]|||=4\times8-5\times7=-3$$

Now we can calculate the final result:

$$|M|=1\times(-48)-2\times(-42)+3\times(-3)=27$$

One of the main uses of the determinant is to *determine* whether a square matrix can be inversed or not: if the determinant is equal to 0, then the matrix *cannot* be inversed (it is a singular matrix), and if the determinant is not 0, then it *can* be inversed.

The determinant can also be used to measure how much a linear transformation affects surface areas: for example, the projection matrices $F_{project}$ and $F_{project\_30}$ completely flatten the polygon $P$, until its area is zero. This is why the determinant of these matrices is 0. The shear mapping modified the shape of the polygon, but it did not affect its surface area, which is why the determinant is 1.

## Eigenvectors and eigenvalues

An **eigenvector** of a square matrix $M$ (also called a **characteristic vector**) is a non-zero vector that remains on the same line after transformation by the linear transformation associated with $M$. A more formal definition is any vector $v$ such that:

$$M\cdot v=\lambda\times v$$

Where $\lambda$ is a scalar value called the **eigenvalue** associated to the vector $v$.

For example, any horizontal vector remains horizontal after applying the shear mapping (as you can see on the image above), so it is an eigenvector of $M$. A vertical vector ends up tilted to the right, so vertical vectors are *NOT* eigenvectors of $M$.

If we look at the squeeze mapping, we find that any horizontal or vertical vector keeps its direction (although its length changes), so all horizontal and vertical vectors are eigenvectors of $F_{squeeze}$.

However, rotation matrices have no eigenvectors at all (except if the rotation angle is 0° or 180°, in which case all non-zero vectors are eigenvectors).

NumPy's `eig` function returns the list of unit eigenvectors and their corresponding eigenvalues for any square matrix.

## Trace

The trace of a square matrix $M$, noted $tr(M)$ is the sum of the values on its main diagonal.

The trace does not have a simple geometric interpretation (in general), but it has a number of properties that make it useful in many areas:

- $tr(A+B)=tr(A)+tr(B)$
- $tr(A \cdot B)=tr(B \cdot A)$
- $tr(A \cdot B \cdot \dots \cdot Y \cdot Z)=tr(Z \cdot A \cdot B \cdot \dots \cdot Y)$
- $tr(A^T \cdot B)=tr(A \cdot B^T)=tr(B^T \cdot A)=tr(B \cdot A^T)=\sum_{i,j} X_{i,j} \times Y_{i,j}$
- …

It does, however, have a useful geometric interpretation in the case of projection matrices (such as $F_{project}$ that we discussed earlier): it corresponds to the number of dimensions after projection