CHAPTER 1

NUMERICAL ARRAYS

1.1 ndarrays

In this course, the data type we shall be focusing on is multi-dimensional numerical arrays. They are called ndarrays (for n-dimensional arrays). Arrays can be used to represent various types of data, e.g. sound, image, video, tabular data, etc.

Multi-dimensional arrays can be easily manipulated with vectorised operations. These are operations that we apply on an entire array without explicitly writing the loop. For example, assume that we have the array `pos`, containing 100 2D coordinates. If we wanted to scale the coordinates by 2 and add an offset of 10 in the y-value, we could simply write `pos = pos * 2 + [0, 10]`. There are no explicit loops, which makes the operations much clearer to read and understand.

In particular, vectorisation is a special type of parallel computing, where the arrays are restricted to a particular size. Modern CPUs have vectorised operations to perform the same task on multiple elements at once (i.e. one clock cycle)- this is called single instruction multiple data (SIMD). GPUs can perform calculations much faster than a CPU.

Shapes of arrays

ndarrays have dimensions (or ranks). A 1D array is a vector. A 2D array is a matrix. An array of higher dimension is called a tensor. The shape of an array refers to the number of elements it can hold in each dimension.

We refer to each dimension within an array by axis. This is indexed by 0, i.e. axis 0 is the rows; axis 1 is the columns; and axis 2 is the depth/frames/etc.

Array operations

There are many types of operations that we can perform on array(s), such as:

- arithmetic, e.g. addition and subtraction;
- indexing and slicing;
- generating arrays, e.g. an array with uniformly distributed numbers between 0 and 1;
- rearranging arrays, e.g. taking the transpose of a matrix or reshaping the array;
- ordering arrays, e.g. sorting the array; and
- aggregation, e.g. finding the sum, start and stop.

Static arrays

There are many differences between a list and an ndarray, as highlighted below:

- ndarrays have a fixed, pre-defined size (or shape). They cannot be extended/resized after definition.
- ndarrays have a fixed, pre-defined type.
- ndarrays can only hold numbers (usually).
- ndarrays are meant to hold multi-dimensional data.
- ndarrays must be rectangular in shape, e.g. equal number of rows in each column.

Nonetheless, ndarrays are still mutable, e.g. we can set a value of the array without creating a new copy.

When initialising ndarrays, we need to specify its type. This is because an ndarray is stored as a block of raw numbers. For this reason, ndarrays have a thin wrapper raw blocks of memory. As a result, they are much more efficiently stored and operations can be performed on them must faster.

1.2 NumPy

We will be using NumPy, which contains a class for ndarrays. An array in NumPy has:

- a shape (e.g. `(5, 6)`) and
- a dtype (e.g. `float64`).

Given a list, we can convert it into an array using `np.array(list)`. We can also create an array in NumPy in the following ways:

- `np.empty(shape)`- it allocates space for an array of the given shape;
- `np.zeros(shape)`- it initialises an array of the given shape with zeroes everywhere;
- `np.ones(shape)`- it initialises an array of the given shape with ones everywhere;
- `np.full(shape, value)`- it initialises an array of the given shape with the given value everywhere.

All the functions that initialise an array first call `np.empty(shape)` to allocate the space. The functions above also have a 'like' version, where instead of providing the shape, we provide an array and take its shape, e.g. `np.ones_like(array)`.

We can also create arrays with random entries given below:

- `np.random.randint(start, stop, shape)`- creates an array with uniform random integers between the given start (inclusive) and the given stop (exclusive);
- `np.random.uniform(start, stop, shape)`- creates an array with uniform random floating point numbers between the given stop and start;
- `np.random.normal(mu, sigma, shape)`- creates an array with normally distributed random floating point numbers with the mean mu, and standard deviation sigma;

We can also create a range of numbers:

- `np.arange(value)`- creates a 1D array of numbers from 0 (inclusive) to value (exlusive);
- `np.arange(start, stop)`- creates a 1D array of numbers from start (inclusive) to stop (exclusive);
- `np.arange(start, stop, step)`- creates a 1D array of numbers from start (inclusive) to end (exclusive), with the given step;
- `np.linspace(start, stop, step)`- creates a 1D array of numbers from start (inclusive) to stop (inclusive), with the given step.

We can open a text file using `np.loadtxt(fname)` and save a file using `np.savetxt(array, fname)`.

Slicing and indexing

Indexing is taking an element from the array, while slicing is taking a subset of the array. We take the multi-dimensional indices when indexing. We have 3 parameters in slicing- `start`, `stop` and `step`, like in vanilla Python. We can reverse an array by using step size -1.

Slicing does not change the rank of an array- it selects a rectangular subset with the same number of dimensions. On the other hand, indexing (usually) reduces the rank- it selects a rectangular subset where one dimension is a singleton.

We can also transpose an array- this exchanges the rows and columns for a 2D array, and flips the order of the axes in general. It is given by `array.T`. This operation takes $O(1)$ time.

Concatenation and Stacking

We can join two arrays (concatenation), or create a new array with the two arrays (stacking). The arrays must be of the right shape for this to happen. If we are concatenating with multiple dimensions, the axis of concatenation must be specified.

Tiling

We can repeat an array multiple times. The function `np.tile(array, lines)` returns another array where the shape of the tiling states how the array should be repeated/joined.

Boolean Arrays

Using boolean arrays, we can check whether elements in the array satisfy a condition. Using the condition `np.where(bool, a, b)`, we create another array, where `a` is what the value of something that satisfies the condition, and `b` is what the value of something that doesn't satisfy the condition. Using `np.nonzero(bool)`, we get an array of indices whether the value is true. We can generate boolean arrays by comparing two arrays, e.g. `x < y`.

1.3 Map and broadcast

Maps

We can apply a function to each element within an array- this is elementwise computation. A map operation can have:

- single argument, e.g. `np.tan(x)` or `-x`;
- two arguments, e.g. `x-y` or `np.maximum(x, y)`; or
- something else, e.g. `np.where(x, 0, 1)`.

In a map operation with multiple arrays, their shapes must be compatible. If the two arrays have the same shape, then the operations is applied elementwise. Otherwise, one of the arrays will be broadcasted so that they have the same shape. Broadcasting involves tiling one of the arrays until the two arrays have the same shape, if possible. This is much more efficient than explicit broadcasting.

Broadcasting

Broadcasting is how arithmetic operations are done on arrays which have different shapes. The rules are:

- If the two arrays have the same number of dimensions, then they must have the same shape. The operation is done elementwise.
- If one array has fewer dimensions than the other, then the last dimensions of the bigger array must match the dimensions of the smaller array.

For example,

- `shape(2, 2) * shape(2, 2)` is allowed;
- `shape(2, 3, 4) * shape(3, 4)` is allowed;
- `shape(2, 3, 4) * shape(2, 3)` is not allowed.

To perform the operation column-wise, we can transpose, perform the operation, then transpose back.

1.4 Reduction and accumulation

Reduction

A reduction/aggregation applies a function to two elements within the array repeatedly to return a scalar. These operations are:

- `np.all(array)` combines the elements in the (boolean) array with `AND`;
- `np.any(array)` combines the elements in the (boolean) array with `OR`;
- `np.min(array)` combines the elements in the array with the minimum operation;
- `np.max(array)` combines the elements in the array with the maximum operation;
- `np.sum(array)` combines the elements in the array with `+`;
- `np.prod(array)` combines the elements in the array with `*`;
- `np.mean(array)` calls `np.sum(array)` and then divides the result by `len(array)`;
- `np.std(array)` calls `np.mean(array)` and then applies the standard deviation formula.

We can specify the axis/axes to only reduce with respect to the given dimension(s). By default, it reduces with respect to all the axes.

Accumulation

Accumulative is the cumulative sum/product of the array. The operations are:

- `np.cumsum(array)` is the accumulation of `+`;
- `np.cumprod(array)` is the accumulation of `*`;
- `np.diff(array)` is the accumulation of `-` and has one less output than input;
- `np.gradient(array)` is like `np.diff(array)`, but uses central differences to get the same length output, and it computes the gradient over every axis and returns them all in a list.

Finding

We can find the indices that match the given criteria:

- `np.argmax(array)` returns the index of the maximum element in the array;
- `np.argmin(array)` returns the the index of the minimum element of the array;

- `np.argsort(array)` returns the indices of tthe elements in the array such that those indices represent the array sorted (we can index the array using the argsorted array to get the sorted array- this is fancy indexing);
- `np.nonzero(array)` returns the indices of all the non-zero elements in the array (or `True` in boolean arrays).

1.5 Introduction to floating point

There are various representations of numbers. The two main classes of numbers are integers and floats. The ranges of different data types is given below.

name	bytes	min	max
int8	1	-128	127
uint8	1	0	255
int16	2	-32,768	32,767
uint16	2	0	65,535
int32	4	-2,147,483,648	2,147,483,647
uint32	4	0	4,294,967,295
int64	8	-9,223,372,036,854,775,808	+9,223,372,036,854,775,807
uint64	8	0	18,446,744,073,709,551,615

Figure 1.1: Ranges for different data types

If we apply an operation that exceeds that maximum value, then we have overflow. Overflows have undefined behaviour. Also, the order of operation matters in code, so some algebraic properties are lost, especially if the numbers are really different. For example, it might be that $ab \neq ba$, $a + b \neq b + a$, and so on.

Floating point numbers can represent a much bigger range of numbers. To acheive this, they store:

- the mantissa- a fractional number with a standardised range (a number from 1.0 to just less than 2.0); and
- the exponent- a scaling or stretching factor (powers of 2).

Using this representation, we can store a lot of digits with very few values. They are, however, less precise than integers for this reason- numbers close to 0 can be represented quite precisely, but big numbers cannot be represented with such precision. A floating point number also stores the sign- whether the number is positive or negative.

A floating point number can therefore be constructed using the three parts. In particular,

$$\texttt{value} = \texttt{sign} * (1.[\texttt{mantissa}]) * (2^{\texttt{exponent}}).$$

Since the value before the mantissa is always 1, we do not need to store it explictly. For example, the `float32` format of a number is:

- 1 bit for the sign;
- 8 bits for the exponent; and
- the remaining 23 bits for the mantissa.

In binary, a `float64` format is:

`1 10000011 0010011101000100100010 1.`

Here,

- the number is negative, since the sign is 1;
- the mantissa represents `1.00100111010001001000101`. In decimal, this is `1.153389573097229`;
- the exponent represents $2^{131-127} = 2^4 = 16$ since $10000011_2 = 131_{10}$. This is because of the implied offset.

Therefore, the number is -18.454233169555664.

The standard for floating point numbers is IEEE754, which specifies the representation of floats, operations on them and some "special" numbers. The standards are given below:

Name	Common name	Base	Digits	Decimal digits	Exponent bits	Decimal E max	Exponent bias	E min	E max	Notes
binary16	Half precision	2	11	3.31	5	4.51	2^4−1 = 15	−14	+15	not basic
binary32	Single precision	2	24	7.22	8	38.23	2^7−1 = 127	−126	+127	
binary64	Double precision	2	53	15.95	11	307.95	2^10−1 = 1023	−1022	+1023	
binary128	Quadruple precision	2	113	34.02	15	4931.77	2^14−1 = 16383	−16382	+16383	
binary256	Octuple precision	2	237	71.34	19	78913.2	2^18−1 = 262143	−262142	+262143	not basic

Figure 1.2: IEEE754 standards

Most of the time, we will be using either single precision `float32` (float) or double precision `float64` (double). GPUs typically are fastest (by a long way) using `float32`, but can do double precision `float64` computations at some significant cost. Most desktop CPUs (e.g. x86) have specialised float64 hardware.

Binary representation of floats

We will now look at some float representations.

- The float representation for 1 is

  `0 01111111111 00..00`

  The sign bit is 0, so the number is positive. The exponent is 0 (after subtracting the bias), and the mantissa is all 0s. So, the value is 2^0.

- The float representation for 4 is

  `0 10000000001 00..00`

  Here, the exponent is 2 and the mantissa is all 0s- the value is 2^2.

- The float representation for 5 is

  `0 10000000001 0100..00`

  Here, the exponent is 2, and the mantissa is 1.25. So, the value is 2^2+1.25.

- The float representation for 0.25 is

  `0 01111111101 00..00`

  Here, the exponent is -2, and the mantissa is all 0s- the value is 2^{-2}.

- The float representation for $1/3$ is

  `0 01111111101 01..01`

  Here, the exponent is -2, and the mantissa is $1.33333\ldots$. So, the value is $2^{-2} * 1.33$. This number isn't represented exactly in binary.

- The float representation for 2000000.01 is

  `0 10000010011 111010000100100....00101001`

  This number also cannot be represented exactly.

- The float representation for 1e-90 is

  `0 01011010100 0000010010111101....11001101111`

  This number can be represented precisely.

- The float representation for 1.5e300 is

  `0 11111100100 0001111010110001....0000110101`

  This number cannot be represented precisely- we represent the value with the closest number to it.

In float64, every integer from -2^{53} to 2^{53} can be represented precisely; integers outside of this range are not represented exactly (this is because the mantissa has 53 bits).

1.6 Floating point weirdness

There are 5 types of errors with floats:

- Invalid operations- this occurs when the result of the operation is not a real number/undefined, e.g. 0/0 or sqrt(-1);
- Division by zero- this occurs when dividing a (nonzero) number by 0;
- Overflow- this occurs if the result of a computation exceeds the limits of the floating point number;
- Underflow- this occurs if the result of a computation is smaller than the smallest representable number, and so is rounded off to zero;
- Inexact- this occurs if a computation will produce an inexact result due to rounding.

An exception might be trapped or untrapped. An untrapped exception will not halt execution, and will instead do some default operation (e.g. untrapped divide by zero will output `np.inf` instead of halting). A trapped exception will cause the process to be signalled in to indicate that the operation is problematic, at which point it can either halt or take another action. Typically, invalid operation is trapped; inexact and underflow aren't trapped; and division by zero and overflow might or might not trapped. It is possible to manually trap them.

There are 2 versions of 0- $+0$ and -0. The `float64` representation of $+0.0$ is

`0 00000000000 00..00`,

and for -0.0 is

`1 00000000000 00..00`.

They only differ in their sign bit, and have both mantissa and exponent 0.

IEEE754 floats encode $+\infty$ and $-\infty$. The `float64` representation of $+\infty$ is

`0 11111111111 00..00`,

and for $-\infty$ is

`1 11111111111 00..00`.

The sign bit represents the sign of the infinity; the mantissa is all 0; and the exponent is all 1.

The infinity value satisfies `np.inf+value = np.inf` and `np.inf*value = np.inf`. NaN (not a number) represents invalid values. The following operations result in NaN:

- `0/0`;
- `np.inf/np.inf` (either positive or negative);
- `np.inf - np.inf` or `np.inf + -np.inf`;
- `np.inf * 0` or `0 * -np.inf`;
- `sqrt(value)`, where value is strictly negative;
- `log(value)`, where value is strictly negative.

The value NaN propagates- any operation on `np.nan` will return `np.nan` (except `1.0**np.nan = 1.0`). Every comparison with NaN results in `False`. In particular, `np.nan = np.nan` returns `False`; we can check whether the value is NaN by using `np.isnan(value)`. However, NaN is equivalent to `False`.

As a floating point number, NaN has exponent all 1 and any non-zero mantissa. So, there are $2^{52} - 1$ versions of NaN in `float64`. An example of such representation is:

`0 11111111111 11..11`.

NaN can be the result of some underflow, which results in division by 0, for example. It can also be used for missing data.

1.7 Roundoff and precision

Due to their representation, floats have roundoff errors. There can be huge gaps between possible floating point representations for big numbers, for instance. Moreover, the value `1.0+1e-8` can be represented quite precisely in `float64`, but the value `1.0+1e-16` gets converted to 1- there is no number closer than 1 for it to be represented by.

We can order the operations to minimise roundoff errors, e.g. `(1.0e30 + 1.0) - 1.0e30` will return 0, but `(1.0e30 - 1.0e30) + 1.0` will return 1. The main issues with roundoff are:

- `x+y` will have a large error if `x` and `y` have different magnitudes (magnitude error), e.g. `1.0 + 1.0` is fine, but `1.0 + 1e300` is bad.
- `x-y` will have a large error if `x` and `y` are approximately equal (cancellation error), e.g. `1000 - 1200` is fine, but `5000.0 - 5000.0000000000005` is not.

Because of roundoff errors, we should not compare two floats for equality. Instead, we should check whether the values are "close" enough, using `np.allclose(x, y)`. This checks whether the difference/ratio of the two numbers is close to 0 or 1 respectively. For a number, we define the relative error of its floating point representation by

$$\epsilon = \frac{|\,\text{float}(x) - x|}{|x|},$$

i.e. the absolute difference between a floating point number and its real part, normalised by the mangitude of the real number. IEEE754 ensures that the value of ϵ is quite small, e.g. for `float64`, it is $\epsilon = 2^{-53}$.

1.8 Arrays in memory

Arrays are tightly packed in memory, which means that they have very small, constant overhead over the storage of elements. There is a short header which describes how the data is present in memory, followed by the data itself.

Arrays are stored as a sequence of numbrs in a long list. This is true for every array- there is a short header followed by a long sequence of numbers. We can see the order of the sequence using `np.ravel(array)`.

Strides and shape

We implement multi-dimensional arrays by striding. It is a set of offset consts (called strides) which specify how to index into the array. There is one value per axis.

For each dimension, there is a stride that tells us how many bytes forward to seek the next dimension for each dimension. For 1D arrays, there is one stride, which is the length of the data type. For 2D arrays, there are two strides- the first may be 8 (one float), and the second may be `8*x.shape[0]`. So, to move to the next column, add 8; to move to the next row, add 8*number of columns. Strides are given in bytes (instead of elements) for more efficiency.

In general, to find the array element at index `[i, j]` in a 2D matrix, the memory offset from the start of the number block will be:

$$\texttt{i} * \texttt{stride[0]} + \texttt{j} * \texttt{stride[1]}.$$

This generalises to higher dimensions (e.g. 3D, 4D tensors). To iterate through an array, the computations can simply increment by the appropriate stride to move to the next element.

This representation of arrays is called a dope vector, which refers to the information in the strides. It is separate to the data, and part of the header. For example, the following is a `float64` 2D matrix of shape `(6, 5)`.

Shape	`(6, 5)`
Strides	`(40, 8)`
Data Type	`float64`
Elements	30
Number of Bytes	240
Bits per element	64
Start Offset	0

To move to the next row, we move by $40 = 8 \times 5$ bytes, while we move to the next column, we move by 8 bytes.

Using strides, transposition becomes $O(1)$. This is because we merely change the strides (the dope vector) and the shape. So, the transposition of the matrix above has the following shape:

Shape	`(5, 6)`
Strides	`(8, 40)`
Data Type	`float64`
Elements	30
Number of Bytes	240
Bits per element	64
Start Offset	0

So, to move to the next row, instead of moving by 40 bytes, we just move by 8 in the transpose. Similarly, to move to the next column, instead of moving by 8 bytes, we move by 40 in the transpose.

There are other operations that change the layout of the matrix, but can performed in $O(1)$, such as:

- `np.flipup(array)`, which flips the matrix up/down. So, if we flip the 6×5 matrix, we get the following shape:

Shape	`(6, 5)`
Strides	`(-40, 8)`
Data Type	`float64`
Elements	30
Number of Bytes	240
Bits per element	64
Start Offset	25

We have reversed the order of the rows. For this reason, the start offset is the $(6-1)\times 5 = 25$-th element, and moving to the next row is moving back 40 bytes.

- `np.fliplr(array)`, which flips the matrix left/right.

Shape	`(6, 5)`
Strides	`(40, -8)`
Data Type	`float64`
Elements	`30`
Number of Bytes	`240`
Bits per element	`64`
Start Offset	`4`

Here, we have reversed the order of the columns here. So, the start is the 4th element (the last element in the first column), and moving to the next column is moving back 8 bytes.

- `np.rot90(array)`, which rotates the matrix by 90 degrees.

Shape	`(5, 6)`
Strides	`(-8, 40)`
Data Type	`float64`
Elements	`30`
Number of Bytes	`240`
Bits per element	`64`
Start Offset	`4`

Rotation by 90 degrees is equivalent to taking the transpose after reversing the order of the columns- the shape and the strides are flipped, but the offset remains the same.

NumPy gives the strides of the original matrix as `(40, 8)` and not `(8, 40)`. NumPy uses C-ordering/row major, called last index changes first. This is different to the Fortran ordering/column major, called first index changes fastest. We can unravel an array in both ways.

1.9 Tensor operations

Reshaping

Along with transposition and flips, we can also reshape an array in constant time. This just changes how the array is indexed, i.e. the strides. We cannot change the number of elements in the array by reshaping. We reshape by `array.reshape(shape)`.

When we reshape, the following rules are obeyed:

- the number of elements in the array remains the same;
- the order of the elements remains the same;

- the last dimension changes fastest, the second last dimension changes the second fastest, and so on.

We can add a (singleton) dimension by indexing with `None`, i.e. `x[:, None]` transforms a 1D vector `x` into a 2D matrix with one column in each row. We can get rid of singleton dimensions using `np.squeeze`. In tensors, we can avoid listing all the indices when slicing, e.g. `x[2, :, :, :, 5]` can be written as `x[2, .., 5]`- this is called ellision.

Swapping and rearranging axes

We can rearrange axes using `np.swapaxs(array, axis1, axis2)` to swap axes `axis1` and `axis2`.

For example, consider the following representation of a $3 \times 4 \times 5$ tensor.

Shape	`(3, 4, 5)`
Strides	`(160, 40, 8)`
Data Type	`float64`
Elements	`60`
Number of Bytes	`480`
Bits per element	`64`
Start Offset	`0`

If we swap axes 0 and 2, we get the following shape:

Shape	`(5, 4, 3)`
Strides	`(8, 40, 160)`
Data Type	`float64`
Elements	`60`
Number of Bytes	`480`
Bits per element	`64`
Start Offset	`0`

This happens in constant time since it merely changes the strides. Reshaping uses the powering rule, i.e. last dimension pours first. However, we might not want to amend the final dimension. In that case, we swap axes, perform the operation and swap back.

We might need to swap multiple axes in one ago. We can do this more efficiently using `np.einsum(expression, array)`, where the expression states how the axes are moved. For example, if we change the $3 \times 4 \times 5$ tensor using the einsum, expression `ijk -> kij`, we change the axes from `(0, 1, 2)` to `(2, 0, 1)`. The representation is:

Shape	`(5, 3, 4)`
Strides	`(8, 160, 40)`
Data Type	`float64`
Elements	`60`
Number of Bytes	`480`
Bits per element	`64`
Start Offset	`0`