

Artificial Neural Networks

Course-2

Gavrilit Dragos

rev 1.0

π

AGENDA FOR TODAY

› NumPy

- What is NumPy
- DataTypes
- Arrays (constructors)
- Shapes
- Slicing and indexes
- Operations with arrays
- Serialization/Deserialization

What is numpy

What is numpy

- › One of the most important Python's mathematical modules, delivering high-speed numerical computing with a robust C/C++ backend for unparalleled efficiency.
- › **Current version:** 1.26.0 (16.Sep.2023)
- › **Site:** <https://numpy.org/>
- › **Documentation:** <https://numpy.org/doc/stable/>

What is numpy

- › To install numpy:
 - Link: <https://numpy.org/>
 - PIP: `pip install numpy`
 - CONDA: `conda install numpy`

- › Compatibility (for latest version): Python 3.12

What is numpy

Support for:

- Arrays (ndarray)
- Mathematical functions (including basic arithmetic operations, statistical functions, algebraic operations, etc)
- Broadcasting (A powerful feature that allows NumPy to work with arrays of different shapes when performing arithmetic operations)
- Linear Algebra
- Indexing and Slicing
- FFT and Polynomials (Built-in support for Fast Fourier Transforms and polynomial operations).
- Support to save/load data into different formats

Data Types

Data types

NumPy supports a variety of numeric data types, which are integral for handling diverse computational requirements.

› *Integers:*

- *int8* : 8-bit signed integer (-128 to 127)
- *int16* : 16-bit signed integer (-32768 to 32767)
- *int32* : 32-bit signed integer (-2^{31} to $2^{31} - 1$)
- *int64* : 64-bit signed integer (-2^{63} to $2^{63} - 1$)

› *Unsigned Integers:*

- *uint8* : 8-bit unsigned integer (0 to 255)
- *uint16* : 16-bit unsigned integer (0 to 65535)
- *uint32* : 32-bit unsigned integer (0 to $2^{32} - 1$)
- *uint64* : 64-bit unsigned integer (0 to $2^{64} - 1$)

Data types

NumPy supports a variety of numeric data types, which are integral for handling diverse computational requirements.

› ***Floating Point:***

- **float16**: Half precision float
- **float32**: Single precision float
- **float64**: Double precision float
- **float128**: Extended precision float (**not available on all platforms**)

› ***Complex Numbers:***

- **complex64**: Complex number with two 32-bit floats (real and imaginary components)
- **complex128**: Complex number with two 64-bit floats
- **complex256**: Complex number with two 128-bit floats (**not available on all platforms**)

Data types

NumPy supports a variety of numeric data types, which are integral for handling diverse computational requirements.

- › ***Boolean***:

- `bool_`: Boolean type storing True and False values

- › ***Strings***:

- `string_`: Fixed-size string type
- `unicode_`: Fixed-size Unicode type

- › ***Datetime***:

- `datetime64`: Date and time representation
- `timedelta64`: Represents the difference between two dates or times

Data types

Numerical data types are directly mapped to corresponding C (and occasionally C++) data types, ensuring both speed and compact memory usage.

The primary reason NumPy is so efficient compared to native Python lists or arrays is that ***it leverages these low-level languages (C/C++)*** for the storage and operations, avoiding the overhead of Python's dynamic typing and object-oriented features.

Data types

C/C++ type backend type mapping:

Numpy type	C/C++ type
int8	Int8_t
int16	int16_t
int32	int32_t
int64	int64_t
uint8	uint8_t
uint16	uint16_t
uint32	uint32_t
uint64	uint64_t
bool_	bool

Numpy type	C/C++ type
float16	Software representation
float32	float
float64	double
float128	long double (depending on architecture)
complex64	mapped to C structs containing two float
complex128	mapped to C structs containing two double
Strings	Array of chars

Arrays (constructors)

Arrays

- › An array can be created using the method `.array(...)` defined as follows:
 - `.array(<data>, [type])`, where
 - › `<data>` is the raw data (one usual solution is to provide a list of values)
 - › `[type]` is an optional parameter that describes a possible scalar type for the data. If not provided, it will be inferred from the data type.
- › Example:

```
import numpy as np

a = np.array([1,2,3],dtype=np.int32)
print(a)
print(a.dtype)
```

Output

```
[1 2 3]
int32
```

Arrays

- › You can also build an array with multiple dimensions (by providing lists within lists as the first parameter.

- › Example:

```
import numpy as np

a = np.array([[1,2,3],[4,5,6]],dtype=np.int32)
print(a)
print(a.dtype)
```

Output

```
[[1 2 3]
 [4 5 6]]
int32
```

Arrays

- › Or a more complex matrix (in terms of shape and definition):
- › Example:

```
import numpy as np

a = np.array(
    [
        [
            [1,2,3],[4,5,6],[7,8,9]
        ],
        [
            [10,20,30],[40,50,60],[70,80,90]
        ],
    ])
print(a)
print(a.dtype)
```

Output

```
[[[ 1  2  3]
  [ 4  5  6]
  [ 7  8  9]]

 [[10 20 30]
  [40 50 60]
  [70 80 90]]]
int32
```


Arrays

- › However, keep in mind that when building n-dimensional arrays you have to respect the shape (sizes) for each list within the declaration.
- › Example:

```
import numpy as np

a = np.array([[1,2,3],[4,5]],dtype=np.int32)
print(a)
print(a.dtype)
```

Error

Traceback (most recent call last):

File "E:\Lucru\RN\teste\a.py", line 3, in <module>

a = np.array([[1,2,3],[4,5]],dtype=np.int32)

ValueError: setting an array element with a sequence. The requested array has an inhomogeneous shape after 1 dimensions. The detected shape was (2,) + inhomogeneous part.

Arrays

- › However, keep in mind that when building n-dimensional arrays you have to respect the shape (sizes) for each list within the declaration.
- › Example:

```
import numpy as np  
  
a = np.array([[1,2,3],[4,5]],dtype=np.int32)  
print(a)  
print(a.dtype)
```

In our case, [1,2,3] has the size of 3 and [4,5] has the size 2 (and its impossible to construct a matrix this way)

Arrays

- › When converting the value from an array into internal data, a cast is being performed (that might lose precision).
- › Examples:

```
import numpy as np

a = np.array([1,"2",4],dtype=np.int32)
print(a)
print(a.dtype)
```

Output

```
[1 2 4]
int32
```

```
import numpy as np

a = np.array([1,2.7,4],dtype=np.int32)
print(a)
print(a.dtype)
```

Output

```
[1 2 4]
int32
```

Arrays

- › However, keep in mind that the case has to be possible (this usually translate into having an operator such as `int()` defined for that specific type).
- › Example:

```
import numpy as np

a = np.array([1,"some text",4],dtype=np.int32)
print(a)
print(a.dtype)
```

Error

Traceback (most recent call last):

File "E:\Lucru\RN\teste\a.py", line 3, in <module>

a = np.array([1,"some text",4],dtype=np.int32)

ValueError: invalid literal for int() with base 10: 'some text'

Arrays

- › If the second parameter is missing, the type is inferred. If the input data contains values of different types, the larger one will be selected.

- › Example:

```
import numpy as np  
  
a = np.array([1,2.5,4])  
print(a)  
print(a.dtype)
```

Output

```
[1.  2.5 4. ]  
float64
```

Since 2.5 (a float64) is the biggest value, the entire array will be considered of this type.

Arrays

- › We should also mention that using strings (or large strings) might have a different behavior.

- › Example:

```
import numpy as np
```

```
a = np.array(["1",2.5,4,"hello world from python and numpy"])  
print(a)  
print(a.dtype)
```

Output

```
['1' '2.5' '4' 'hello world from python and numpy']  
<U33
```

This has the following translation:

- “<” → little endian
- “U” → Unicode string
- “33” → 33 characters

Arrays

- › Another way of create an array is with `.zeros(<shape>,[type])` method.
- › Examples:

```
import numpy as np

a = np.zeros((3,4))
print(a)
print(a.dtype)
```

Output

```
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
float64
```

```
import numpy as np

a = np.zeros((2,4),np.uint16)
print(a)
print(a.dtype)
```

Output

```
[[0 0 0 0]
 [0 0 0 0]]
uint16
```

Arrays

- › Similarly, `.ones(<shape>,[type])` method performs the same task but uses 1 instead of 0 to fill a matrix:
- › Examples:

```
import numpy as np

a = np.ones((2,3))
print(a)
print(a.dtype)
```

Output

```
[[1.  1.  1.]
 [1.  1.  1.]]
float64
```

```
import numpy as np

a = np.ones((2,4),np.uint16)
print(a)
print(a.dtype)
```

Output

```
[[1  1  1  1]
 [1  1  1  1]]
uint16
```


Arrays

- › Another method is `.arange(...)` that creates a vector with consecutive values. Since every array/vector can be reshape into another array, this method can be used to create any kind of array
- › Format:
 - `.arange(stop,[type])`
 - `.arange(start, stop, [type])`
 - `.arange(start, stop, step, [type])`

Arrays

- › Example (with **stop** parameter):

```
import numpy as np  
  
a = np.arange(10)  
print(a)  
print(a.dtype)
```

Output

```
[0 1 2 3 4 5 6 7 8 9]  
int32
```

- › Example (with **start** and **stop** parameter):

```
import numpy as np  
  
a = np.arange(3,10)  
print(a)  
print(a.dtype)
```

Output

```
[3 4 5 6 7 8 9]  
int32
```

Arrays

- › Example (with **start**, **stop** and **step** parameter):

```
import numpy as np

a = np.arange(3,10,2)
print(a)
print(a.dtype)
```

Output

```
[3 5 7 9]
int32
```

- › Example (with **start**, **stop** , **step** and **type** parameters):

```
import numpy as np

a = np.arange(3,10,2,dtype=np.float16)
print(a)
print(a.dtype)
```

Output

```
[3. 5. 7. 9.]
float16
```

Arrays

- › For large arrays, you can use `.empty(shape,[type])` to create an empty array and then fill it with values. If the type is not provided, ***float*** is implied.

- › Example:

```
import numpy as np

a = np.empty((3,2),dtype=np.int32)
print(a)
print(a.dtype)
```

Output (probable)

```
[[1800082696 459]
 [          32  0]
 [1800059672 459]]
int32
```

- › Notice that the values are random (what lies in memory at the moment of allocation).

Arrays

- › Another method is `.full(shape, fill_value, [type])` that can be used to create a matrix filled with a specific value

- › Example:

```
import numpy as np

a = np.full((3,2),3)
print(a)
print(a.dtype)
```

Output

```
[[3 3]
 [3 3]
 [3 3]]
int32
```

Arrays

› Overview

Method	Purpose
<code>.array(<data>, [type])</code>	Create an array based on python lists / tuples
<code>.zeros(<shape>,[type])</code>	Create an array filled with zeros based on a shape and a type
<code>.ones(<shape>,[type])</code>	Create an array filled with ones based on a shape and a type
<code>.full(<shape>,value, [type])</code>	Create an array filled with <i>value</i> based on a shape and a type
<code>.arange(stop,[type])</code>	Creates a vector with values from 0 to stop
<code>.arange(start, stop, [type])</code>	Creates a vector with values from <i>start</i> to <i>stop</i>
<code>.arange(start, stop, step, [type])</code>	Creates a vector with values from <i>start</i> to <i>stop</i> (with step <i>step</i>)
<code>.empty(<shape>,[type])</code>	Create an uninitialized array based on a shape and type

Shapes

Shapes

There is a difference between how we (people) understand a vector / and array / etc and how a computer stores the data.



A	B	C	D	E	F	G	H
---	---	---	---	---	---	---	---

A	B	C	D
E	F	G	H

A	B
C	D
E	F
G	H

	E	F
A	B	
C	D	H

Vector
Shape: (8)

Bi-dimentional
matrix
Shape: (2 , 4)

Bi-dimentional
matrix
Shape: (4 , 2)

tri-dimentional
matrix
Shape: (2 , 2, 2)

Shapes

- › For any ***numpy*** array you can use `.shape` and `.size` to get the shape of the array and its size (total number of elements in the array)
- › Example:

```
import numpy as np

a = np.array([1,2,3])
print(a)
print(a.shape)
print(a.size)
```

Output

```
[1 2 3]
(3,)
3
```

```
import numpy as np

a = np.array([[1,2,3],[4,5,6]])
print(a)
print(a.shape)
print(a.size)
```

Output

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

Shapes

- › We can also reshape any array (as long as the number of elements of the resulting shape is the same as the one from the current array). For this we can use the method **reshape(...)**.
- › Example:

```
import numpy as np

a = np.array([1,2,3,4,5,6,7,8])
print(a)
print(a.shape)
print(a.size)
a = a.reshape((2,4))
print(a)
print(a.shape)
```

Output

```
[1 2 3 4 5 6 7
 8]
(8,)
8
[[1 2 3 4]
 [5 6 7 8]]
(2, 4)
```

Shapes

- › Additionally, we can also use:
 - `.size` → to get the total number of elements in the array
 - `.ndim` → to get the number of dimensions
 - `.itemsize` → to get the size of an item/element (in bytes)
- › Example:

```
import numpy as np

a = np.array([[1,2,3],[4,5,6]])
print("Array      = ",a)
print("Shape      = ",a.shape)
print("Dim        = ",a.ndim)
print("Size       = ",a.size)
print("Type       = ",a.dtype)
print("ItemSize  = ",a.itemsize)
```

Output

```
Array      =  [[1 2 3]
               [4 5 6]]
Shape      =  (2, 3)
Dim        =  2
Size       =  6
Type       =  int32
ItemSize   =  4
```

Indexes & Slicing

Indexes & Slicing

- › Indexes and slices work in a similar manner as with Python list (in particular if we are referring to vectors).

- › Example:

```
import numpy as np

a = np.array([1,2,3,4,5,6])
print("a[0]    = ", a[0])
print("a[-1]   = ", a[-1])
print("a[1:3]  = ", a[1:3])
print("a[:2]   = ", a[:2])
```

Output

```
a[0]    = 1
a[-1]   = 6
a[1:3]  = [2 3]
a[:2]   = [1 2]
```

Indexes & Slicing

- › In case of matrixes, the logic is similar (but the results are somehow different as they will not be a scalar but rather another matrix or vectors).
- › Example:

```
import numpy as np

a = np.array([[1,2,3],[4,5,6]])
print("a[0]    = ",a[0])
print("a[-1]   = ",a[-1])
print("a[1:4]  = ",a[1:4])
print("a[:2]   = ",a[:2])
```

Output

```
a[0]    = [1 2 3]
a[-1]   = [4 5 6]
a[1:4]  = [[4 5 6]]
a[:2]   = [[1 2 3]
           [4 5 6]]
```

Indexes & Slicing

- › In case of matrixes, the logic is similar (but the results are somehow different as they will not be a scalar but rather another matrix or vectors).
- › Example:

```
import numpy as np
```

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

```
print("a[0] = ",a[0])
```

```
print("a[-1] = ",a[-1])
```

```
print("a[1:4] = ",a[1:4])
```

```
print("a[:2] = ",a[:2])
```

Output

```
a[0] = [1 2 3]
```

```
a[-1] = [4 5 6]
```

```
a[1:4] = [[4 5 6]]
```

```
a[:2] = [[1 2 3]
```

```
[4 5 6]]
```

Notice that even if the upper index (3) is not an index that exists in the matrix (the shape of the matrix is (2x3) – so only 2 rows (not 3), the slicing operation is limited to the maximum number of rows

Indexes & Slicing

- › In case of matrixes, you can access an element or a smaller matrix by using the “,” character and specify multiple dimensions and operations such as slicing to be performed for each dimension.
- › Example:

```
import numpy as np

a = np.array([[1,2,3],[4,5,6]])
print("a[0,0] = ",a[0,0])
print("a[-1,-1] = ",a[-1,-1])
print("a[1:2,2] = ",a[1:2,2])
print("a[:2,:2] = ",a[:2,:2])
```

Output

```
a[0,0] = 1
a[-1,-1] = 6
a[1:2,2] = [6]
a[:2,:2] = [[1 2]
            [4 5]]
```


Indexes & Slicing

- › In case of matrixes, you can access an element or a smaller matrix by using the “,” character and specify multiple dimensions and operations such as slicing to be performed for each dimension.
- › Example:

```
import numpy as np
```

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

```
print("a[0,0] = ",a[0,0])
```

```
print("a[-1,-1] = ",a[-1,-1])
```

```
print("a[1:2,2] = ",a[1:2,2])
```

```
print("a[:2,:2] = ",a[:2,:2])
```

Output

```
a[0,0] = 1
```

```
a[-1,-1] = 6
```

```
a[1:2,2] = [6]
```

```
a[:2,:2] = [[1 2]  
            [4 5]]
```

In this case we request a slicing from the original matrix (of shape 2x3) to another matrix of shape 2x2

Indexes & Slicing

- › You can also use another special character (...) that special to fill all of the rest of the dimensions (similar to using ':' in Python list for each dimension).
- › Example:

```
import numpy as np

a = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
print("a[0,0]      = ", a[0, 0])
print("a[-1,-1,-1] = ", a[-1, -1, -1])
print("a[1,...]     = ", a[1,...])
print("a[1,:,:]     = ", a[1,:,:])
```

Notice that a[1,...] and a[1,:,:] are equivalent ! 

Output

```
a[0,0]      = [1 2 3]
a[-1,-1,-1] = 12
a[1,...]     = [[ 7  8  9]
                [10 11 12]]
a[1,:,:]     = [[ 7  8  9]
                [10 11 12]]
```

Indexes & Slicing

- › Additionally, the index operation can be used with a function parameter (something that evaluates to true or false). That function works as a filtering method, keeping only the elements that yield true when that function is called upon them. When using this method of filtering use the name of the array as a reference to each element:
- › Example:

```
import numpy as np

a = np.array([1,2,3,4,5,6])
print("a[a>5]      = ", a[a>5])
print("a[a%2==0] = ", a[a%2==0])
```

Output

```
a[a>5]      = [6]
a[a%2==0] = [2 4 6]
```

Indexes & Slicing

- › Additionally, the index operation can be used with a function parameter (something that evaluates to true or false). That function works as a filtering method, keeping only the elements that yield true when that function is called upon them. When using this method of filtering use the name of the array as a reference to each element:

- › Example

In this case, the **"a"** from **"a>5"** expression represents each element from **"a"** matrix.

```
import numpy as np
a = np.array([1,2,3,4,5,6])
print("a[a>5] = ", a[a>5])
print("a[a%2==0] = ", a[a%2==0])
```

Output

```
a[a>5] = [6]
a[a%2==0] = [2 4 6]
```

Indexes & Slicing

- › When this method is being used with matrixes, it works in a similar manner, but the result is not a matrix , but a vector.
- › Example:

```
import numpy as np

a = np.array([[1,2,3],
              [4,5,6]])
print("a[a>5]      = ", a[a>5])
print("a[a%2==0] = ", a[a%2==0])
```

Output

```
a[a>5]      = [6]
a[a%2==0] = [2 4 6]
```

- › You can use this method to convert a matrix into a vector:

```
import numpy as np

a = np.array([[1,2,3],[4,5,6]])
print("a[a>0] = ",a[a>0])
```

Output

```
a[a>0] = [1 2 3 4 5 6]
```

Indexes & Slicing

› You can also use for statement to iterate through a matrix or vector.

› Example:

```
import numpy as np

a = np.array([1,2,3,4,5,6])
for i in a:
    print(i,)
```

Output

1
2
3
4
5
6

Indexes & Slicing

- › In case of matrixes, the output of one iteration will be a lower dimension matrix or a vector for a bidimensional matrix.

- › Example:

```
import numpy as np

a = np.array([[1,2,3],[4,5,6]])
for i in a:
    print(i,)
```

Output
[1 2 3]
[4 5 6]

Indexes & Slicing

› To iterate through all elements in a matrix use multiple imbricated for statements:

› Example:

```
import numpy as np

a = np.array([[1,2,3],
              [4,5,6]])

for row in a:
    for element in row:
        print(row,":",element)
```

Output

```
[1 2 3] : 1
[1 2 3] : 2
[1 2 3] : 3
[4 5 6] : 4
[4 5 6] : 5
[4 5 6] : 6
```


Operations

Indexes & Slicing

- › The following operations are supported by numpy:
 - **Arithmetic operations** (addition, subtraction, multiplication, division) between arrays of the same shape. The operation are performed element wise and the operation results in a new array of the same shape.
 - **Scalar operations** (also called broadcasting) – addition, subtraction, multiplication and division between an array and a scalar.
 - **DOT product**
 - **Matrix multiplication**
 - **Transpose matrix**
 - **Statistical operations** (sum, min, max, mean, ...)

Array operations

› Example (vector addition):

```
import numpy as np

v = np.array([1,2,3])
u = np.array([10,20,30])
print("sum = ",v+u)
print("dif = ",u-v)
print("mul = ",v*u)
print("div = ",u/v)
```

Output

```
sum =  [11 22 33]
dif =  [ 9 18 27]
mul =  [10 40 90]
div =  [10. 10. 10.]
```

› Example (vector addition – different shapes):

```
import numpy as np

v = np.array([1,2,3])
u = np.array([10,20,30,40])
print("sum = ",v+u)
```

Error

Traceback (most recent call last):

File "E:\Lucru\RN\teste\a.py", line 5, in <module>

print("sum = ",v+u)

ValueError: operands could not be broadcast together with shapes (3,) (4,)

Array operations

› Example (matrix addition):

```
import numpy as np

v = np.array([[1,2,3],[4,5,6]])
u = np.array([[10,20,30],[40,50,60]])
print("sum = ",v+u)
print("dif = ",u-v)
print("mul = ",v*u)
print("div = ",u/v)
```

Output

```
sum =  [[11 22 33]
        [44 55 66]]
dif =  [[ 9 18 27]
        [36 45 54]]
mul =  [[ 10  40  90]
        [160 250 360]]
div =  [[10. 10. 10.]
        [10. 10. 10.]
```

› Example (matrix addition – different shapes):

```
import numpy as np

v = np.array([[1,2,3],[4,5,6]])
u = np.array([[10,20],[40,50]])
print("sum = ",v+u)
```

Error

```
Traceback (most recent call last):
  File "E:\Lucru\RN\teste\a.py", line 5, in <module>
    print("sum = ",v+u)
  ValueError: operands could not be broadcast
  together with shapes (2,3) (2,2)
```

Array operations

- › Arithmetic operations create a new array. This can impact the overall performance and memory. If a new array is not needed, but instead it is possible to perform the operation directly over an existing array, you can use operators such as `'+='`, `'-='`, `'*='`, `'/='`
- › Example (matrix addition – different shapes):

```
import numpy as np

v = np.array([[1,2,3],[4,5,6]])
u = np.array([[10,20,30],[40,50,60]])
v += u
print("sum = ",v)
```

Output	
sum =	$\begin{bmatrix} 11 & 22 & 33 \\ 44 & 55 & 66 \end{bmatrix}$

Array operations

- › Operations between a matrix and a scalar are performed elementwise (for each element in the matrix, the scalar operation is being performed). The result is a new matrix with the same shape as the original one. It is also possible to compute this changes in place via `+=`, `-=`, `*=`, `/=` operators.

- › Example:

```
import numpy as np

m = np.array([[1,2,3],[4,5,6]])
print("add = ",m+2)
print("sub = ",m-1)
print("mul = ",m*2)
print("div = ",m/2)
```

Output

```
add =  [[3 4 5]
        [6 7 8]]
sub =  [[0 1 2]
        [3 4 5]]
mul =  [[ 2  4  6]
        [ 8 10 12]]
div =  [[0.5 1.  1.5]
        [2.  2.5 3.  ]]
```

Array operations

› Dot product can be computed using `.dot()` method

› Example:

```
import numpy as np  
  
v = np.array([1,2,3])  
u = np.array([4,5,6])  
print("dot = ",v.dot(u))
```

Output

Dot = 32

$$v = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}, u = \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix}, \text{ then } v \cdot u = (1 \times 4) + (2 \times 5) + (3 \times 6) = 4 + 10 + 18 = 32$$

Array operations

- › You can also use `.dot()` method to compute matrix multiplication. The same result is obtained via '@' operator.
- › Example:

```
import numpy as np

v = np.array([[1,2,3],
              [4,5,6]])
u = np.array([[10,20,30,40],
              [40,50,60,70],
              [80,90,100,110]])
print("result      = ",v.dot(u))
print("matrix_mul = ",v @ u)
```

Output

```
result      = [[ 330  390  450  510]
               [ 720  870 1020 1170]]
matrix_mul = [[ 330  390  450  510]
               [ 720  870 1020 1170]]
```


Array operations

- › You can also use `.dot()` method to compute matrix multiplication. The same result is obtained via '@' operator.

- › Example:

```
import numpy as np

v = np.array([1, 2, 3],
              [4, 5, 6])
u = np.array([[10, 20, 30, 40],
              [40, 50, 60, 70],
              [80, 90, 100, 110]])
print("result = ", v.dot(u))
print("matrix_mul = ", v @ u)
```

390 = dot product (row 1 from matrix "v" and
column 2 from matrix "u") =

$$1 \times 20 + 2 \times 50 + 3 \times 90 = 20 + 100 + 270 = 390$$

Output

```
result = [[ 330  390  450  510]
          [ 720  870 1020 1170]]
matrix_mul = [[ 330  390  450  510]
              [ 720  870 1020 1170]]
```

Array operations

- › To transpose a matrix, use the `.transpose()` method
- › Example:

```
import numpy as np  
  
m = np.array([[1,2,3],  
              [4,5,6]])  
print("matrix = ",m)  
print("transpose = ",m.transpose())
```

Output

```
matrix =  [[1 2 3]  
           [4 5 6]]  
transpose = [[1 4]  
             [2 5]  
             [3 6]]
```

Array operations

- › Several statistical operations are available. When used against a matrix, without any parameters, all elements from the matrix are being used for analysis.

- › Example:

```
import numpy as np

m = np.array([[1,2,3],[4,5,6]])
print("sum      = ",m.sum())
print("min      = ",m.min())
print("max      = ",m.max())
print("mean     = ",m.mean())
```

Output

sum	=	21
min	=	1
max	=	6
mean	=	3.5

Array operations

- › However, all of the statistical methods have a parameter called ***axes*** that can allow specifying an axes/shape that can be used to limit the computations for a specific dimension
- › Example:

```
import numpy as np

m = np.array([[1,2,3],
              [4,5,6]])

print("sum      = ", m.sum(axis=0))
print("min      = ", m.min(axis=0))
print("max      = ", m.max(axis=0))
print("mean     = ", m.mean(axis=0))
```

Output

sum	=	[5 7 9]
min	=	[1 2 3]
max	=	[4 5 6]
mean	=	[2.5 3.5 4.5]

Array operations

- › However, all of the statistical methods have a parameter called ***axis*** that can allow specifying an axes/shape that can be used to limit the computations for a specific dimension

- › Example:

axis = 0 (means "X" axis). This translates that all operations will be performed over "X" axis. For sum, the values [5 7 9] are obtained in the following way: $5=1+4$, $7=2+5$, $9=3+6$

```
import numpy as np

m = np.array([1, 2, 3],
              [4, 5, 6])

print("sum      = ", m.sum(axis=0))
print("min      = ", m.min(axis=0))
print("max      = ", m.max(axis=0))
print("mean     = ", m.mean(axis=0))
```

Output

sum	=	[5 7 9]
min	=	[1 2 3]
max	=	[4 5 6]
mean	=	[2.5 3.5 4.5]

Array operations

- › Similarly, using `axis=1` refers to the “Y” axis (the second dimension) and all computation will be made around that axis.

- › Example:

```
import numpy as np

m = np.array([[1,2,3],
              [4,5,6]])

print("sum      = ", m.sum(axis=1))
print("min      = ", m.min(axis=1))
print("max      = ", m.max(axis=1))
print("mean     = ", m.mean(axis=1))
```

Output

sum	=	[6 15]
min	=	[1 4]
max	=	[3 6]
mean	=	[2. 5.]

Array operations

- › Similarly, using `axis=1` refers to the “Y” axis (the second dimension) and all computation will be made around that axis.

- › Example:

In this case, the values for sum method are computed in the following way:

$$6=1+2+3, 15=4+5+6$$

```
import numpy as np

m = np.array([[1,2,3],
              [4,5,6]])

print("sum      = ",m.sum(axis=1))
print("min      = ",m.min(axis=1))
print("max      = ",m.max(axis=1))
print("mean     = ",m.mean(axis=1))
```

Output

```
sum      = [ 6 15]
min      = [1 4]
max      = [3 6]
mean     = [2. 5.]
```

Array operations

- › Other statistical operations (such as percentile, mean, median) are used as functions exported directly from the numpy module.
- › Example:

```
import numpy as np

v = np.array([1,1,2,2,3,4,5,6])
print("mean          = ", np.mean(v))
print("median        = ", np.median(v))
print("average        = ", np.average(v))
print("percentile 5   = ", np.percentile(v,5))
print("percentile 50  = ", np.percentile(v,50))
print("percentile 95  = ", np.percentile(v,95))
```

Output

mean	=	3.0
median	=	2.5
average	=	3.0
percentile 5	=	1.0
percentile 50	=	2.5
percentile 95	=	5.64

Array operations

- › You can also use method such as `.square()`, `.sqrt()` or `.cbert()` [*Cubic root*] to performed more advanced mathematical computation (for both arrays and scalars)
- › Example:

```
import numpy as np

m = np.array([[1,2,3],[4,5,6]])
print("Square          = ",np.square(m))
print("Sqrt            = ",np.sqrt(m))
print("Cubic root      = ",np.cbrt(m))
print("Scalar square(2) = ",np.square(2))
print("Scalar sqrt(100) = ",np.sqrt(100))
print("Scalar cbrt(27)  = ",np.cbrt(27))
```

Output

```
Square          = [[ 1  4  9]
                   [16 25 36]]
Sqrt            = [[1.  1.41 1.73]
                   [2.  2.23 2.44]]
Cubic root      = [[1.  1.25 1.44]
                   [1.58 1.70 1.81]]
Scalar square(2) = 4
Scalar sqrt(100) = 10.0
Scalar cbrt(27)  = 3.0
```

Array operations

- › Besides these functions, there a lot of other methods available in the numpy library (including methods for sorting elements, resize-ing arrays, etc).
- › Those methods are improved for optimal performance.

Array operations

- › All of these functions can be used together to compute more complex functions.
- › Example (Root Mean Square Error – RMSE):

$v = [...]$, and $u = [...]$, $len(v) = len(u) = n$,

$$RMSE(v, u) = \sqrt{\frac{1}{n} \sum_{i=1}^n (v_i - u_i)^2}$$

Array operations

- › Let's see how we can write RMSE in numpy
- › Example :

```
import numpy as np

v = np.array([1,2,3,4])
u = np.array([5,6,7,8])

# it is assumed the v and u have the same size
rmse = np.sqrt(np.sum(np.square(v-u))/v.size)

print("RMSE = ",rmse)
```

Output

RMSE = 4

Array operations

- › Let's see how we can write RMSE in numpy
- › Example :

```
import numpy as np

v = np.array([1,2,3,4])
u = np.array([5,6,7,8])

# it is assumed the v and u have the same size
rmse = np.sqrt(np.sum(np.square(v-u)/v.size))

print("RMSE = ",rmse)
```

Debug

```
v-u = [1,2,3,4] - [5,6,7,8] =
      = [1-5, 2-6, 3-7, 4-8] =
      = [-4,-4,-4,-4]
```

Array operations

- › Let's see how we can write RMSE in numpy
- › Example :

```
import numpy as np

v = np.array([1,2,3,4])
u = np.array([5,6,7,8])

# it is assumed the v and u have the same size
rmse = np.sqrt(np.sum(np.square(v-u)/v.size))

print("RMSE = ",rmse)
```

Debug

```
v-u = [1,2,3,4] - [5,6,7,8] =
      = [1-5, 2-6, 3-7, 4-8] =
      = [-4,-4,-4,-4]
```

```
square(v-u) = [-42, -42, -42, -42]
              = [16, 16, 16, 16]
```

Array operations

- › Let's see how we can write RMSE in numpy
- › Example :

```
import numpy as np

v = np.array([1,2,3,4])
u = np.array([5,6,7,8])

# it is assumed the v and u have the same size
rmse = np.sqrt(np.sum(np.square(v-u))/v.size)

print("RMSE = ",rmse)
```

Debug

```
v-u = [1,2,3,4] - [5,6,7,8] =
      = [1-5, 2-6, 3-7, 4-8] =
      = [-4,-4,-4,-4]
```

```
square(v-u) = [-42, -42, -42, -42]
              = [16, 16, 16, 16]
```

```
sum(square(..)) = 16+16+16+16
                 = 64
```

Array operations

- › Let's see how we can write RMSE in numpy
- › Example :

```
import numpy as np

v = np.array([1,2,3,4])
u = np.array([5,6,7,8])

# it is assumed the v and u have the same size
rmse = np.sqrt(np.sum(np.square(v-u))/v.size)

print("RMSE = ",rmse)
```

Debug

```
v-u = [1,2,3,4] - [5,6,7,8] =  
      = [1-5, 2-6, 3-7, 4-8] =  
      = [-4,-4,-4,-4]
```

```
square(v-u) = [-42, -42, -42, -42]  
              = [16, 16, 16, 16]
```

```
sum(square(..)) = 16+16+16+16  
                 = 64
```

```
sum(...)/size = 64 / 4 = 16
```


Array operations

- › Let's see how we can write RMSE in numpy
- › Example :

```
import numpy as np

v = np.array([1,2,3,4])
u = np.array([5,6,7,8])

# it is assumed the v and u have the same size
rmse = np.sqrt(np.sum(np.square(v-u))/v.size)

print("RMSE = ",rmse)
```

Debug

```
v-u = [1,2,3,4] - [5,6,7,8] =
      = [1-5, 2-6, 3-7, 4-8] =
      = [-4,-4,-4,-4]

square(v-u) = [-42, -42, -42, -42]
              = [16, 16, 16, 16]

sum(square(..)) = 16+16+16+16
                 = 64

sum(...)/size = 64 / 4 = 16

sqrt(sum(...)/size) = sqrt(16)
                    = 4
```

Serialization/Deserialization

Serialization/Deserialization

- › While the `.save()` method saves a file in a binary format, there is another method (`.savetxt(...)`) that can be used to save a file as a text (more suitable for debug purposes).

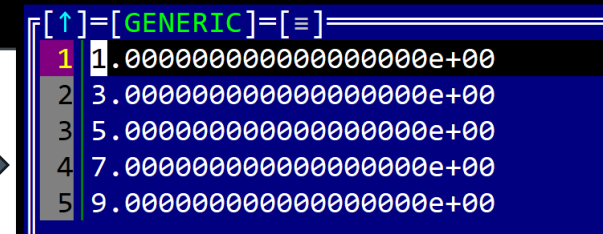
- › Example:

```
import numpy as np  
  
v = np.array([1,3,5,7,9])  
np.savetxt("my_array.txt", v)  
u = np.loadtxt("my_array.txt")  
print(u)
```

Output

[1 3 5 7 9]

my_array.txt file content



```
[↑]=[GENERIC]=[≡]  
1 1.0000000000000000e+00  
2 3.0000000000000000e+00  
3 5.0000000000000000e+00  
4 7.0000000000000000e+00  
5 9.0000000000000000e+00
```

Serialization/Deserialization

- › In practice, complex models are formed out of multiple arrays (and it is easier to save all of them into an archive than in individual files). For this purpose, there are two methods: `.savez(...)` and `.savez_compressed(...)` (“z” stands for zip). The default extension will be ***npz***

- › Example:

```
import numpy as np

v = np.array([1,3,5,7,9])
u = np.array([[1,3],[5,7]])
t = np.array([1,2,3])

np.savez("my_arrays", vector1=v, matrix1=u, vector2=t)
```

Content of my_arrays.npz

vector1.npy

vector2.npy

matrix1.npy

Serialization/Deserialization

- › In practice, complex models are form out of multiple arrays (and it is easier to save all of them into an archive than in individual files). For this purpose, there are two methods: `.savez(...)` and `.savez_compressed(...)` (“z” stands for zip). The default extension will be ***npz***

- › Example:

```
import numpy as np

v = np.array([1,3,5,7,9])
u = np.array([[5, 2], [5, 7]])
t = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
```

Array “u” will be store in this file

```
np.savez("my_arrays", vector1=v, matrix1=u, vector2=t)
```

Content of my_arrays.npz

vector1.npy

vector2.npy

matrix1.npy

Serialization/Deserialization

- › To load a zip archive, created by the previous code, we can use the `np.load(...)` method and the property `files` (to access all arrays).

- › Example:

```
import numpy as np

data = np.load("my_arrays.npz")
print(data.files)
for fname in data.files:
    print(fname + " = ", data[fname])
```

Output

```
['vector1', 'matrix1', 'vector2']
vector1 = [1 3 5 7 9]
matrix1 = [[1 3]
           [5 7]]
vector2 = [1 2 3]
```

Serialization/Deserialization

› Another interesting method is `.genfromtxt(...)` with the following definition:

```
› numpy.genfromtxt( fname,  
                    dtype=<class 'float'>,  
                    comments='#',  
                    delimiter=None,  
                    skip_header=0,  
                    skip_footer=0,  
                    missing_values=None,  
                    filling_values=None,  
                    usecols=None,  
                    replace_space='_',  
                    case_sensitive=True,  
                    ...  
                    )
```


Serialization/Deserialization

- › This method (`.genfromtxt(...)`) can be used to load data in a list format files (such as a .csv or .tsv file) into an array or matrix.
- › Some examples:

```
import numpy as np

# Load data from a regular cvs file with no headers
array = np.genfromtxt('csv_file.csv', delimiter=',')

# Load data from a cvs file with headers
array = np.genfromtxt('csv_file.csv', delimiter=',', skip_header=1)
```

π

Q & A

