Introduction-to-NumPy

May 3, 2020

1 Numpy Refresher

1.0.1 Why do we need a special library for math and DL?

Python provides data types such as lists / tuples out of the box. Then, why are we using special libraries for deep learning tasks, such as Pytorch or TensorFlow, and not using standard types?

The major reason is efficiency - In pure python, there are no primitive types for numbers, as in e.g. C language. All the data types in Python are objects with lots of properties and methods. You can see it using the `dir` function:

```
In [1]: a = 3
        dir(a)[-10:]
```

```
Out[1]: ['__trunc__',
         '__xor__',
         'bit_length',
         'conjugate',
         'denominator',
         'from_bytes',
         'imag',
         'numerator',
         'real',
         'to_bytes']
```

1.0.2 Python Issues

- slow in tasks that require tons of simple math operations on numbers
- huge memory overhead due to storing plain numbers as objects
- runtime overhead during memory dereferencing - cache issues

NumPy is an abbreviation for "numerical python" and as it stands from the naming it provides a rich collection of operations on the numerical data types with a python interface. The core data structure of NumPy is `ndarray` - a multidimensional array. Let's take a look at its interface in comparison with plain python lists.

2 Performance comparison of Numpy array and Python lists

Let's imagine a simple task - we have several 2-dimensional points and we want to represent them as a list of points for further processing. For the sake of simplicity of processing we will not create a `Point` object and will use a list of 2 elements to represent coordinates of each point (`x` and `y`):

```
In [2]: # create points list using explicit specification of coordinates of each point
        points = [[0, 1], [10, 5], [7, 3]]
        points

Out[2]: [[0, 1], [10, 5], [7, 3]]
```

```
In [3]: # create random points
        from random import randint

        num_dims = 2
        num_points = 10
        x_range = (0, 10)
        y_range = (1, 50)
        points = [[randint(*x_range), randint(*y_range)] for _ in range(num_points)]
        points

Out[3]: [[6, 39],
         [4, 5],
         [0, 5],
         [4, 14],
         [8, 8],
         [1, 36],
         [6, 3],
         [9, 45],
         [6, 21],
         [4, 41]]
```

How can we do the same using Numpy? Easy!

```
In [5]: import numpy as np
        points = np.array(points)  # we are able to create numpy arrays from python lists
        points

Out[5]: array([[ 6, 39],
               [ 4,  5],
               [ 0,  5],
               [ 4, 14],
               [ 8,  8],
               [ 1, 36],
               [ 6,  3],
               [ 9, 45],
               [ 6, 21],
               [ 4, 41]])
```

```
In [6]: # create random points using numpy library
        num_dims = 2
        num_points = 10
        x_range = (0, 11)
        y_range = (1, 51)
        points = np.random.randint(
            low=(x_range[0], y_range[0]), high=(x_range[1], y_range[1]), size=(num_points, num_
        )
        points
```

```
Out[6]: array([[ 7,  6],
               [ 6, 34],
               [10, 41],
               [ 2, 30],
               [10,  9],
               [ 5,  2],
               [ 5, 46],
               [ 7, 11],
               [ 7, 20],
               [ 6, 43]])
```

It may look as over-complication to use NumPy for the creation of such a list and we still cannot see the good sides of this approach. But let's take a look at the performance side.

```
In [7]: num_dims = 2
        num_points = 100000
        x_range = (0, 10)
        y_range = (1, 50)
```

2.0.1 Python performance

```
In [8]: %timeit \
        points = [[randint(*x_range), randint(*y_range)] for _ in range(num_points)]
```

```
211 ms ś 8.62 ms per loop (mean ś std. dev. of 7 runs, 1 loop each)
```

2.0.2 NumPy performance

```
In [9]: %timeit \
        points = np.random.randint(low=(x_range[0], y_range[0]), high=(x_range[1], y_range[1])
```

```
4.95 ms ś 9.33 ţs per loop (mean ś std. dev. of 7 runs, 100 loops each)
```

Wow, NumPy is **around 50 times faster** than pure Python on this task! One may say that the size of the array we're generating is relatively large, but it's very reasonable if we take into account the dimensions of inputs (and weights) in neural networks (or math problems such as hydrodynamics).

3 Basics of Numpy

We will go over some of the useful operations of Numpy arrays, which are most commonly used in ML tasks.

3.1 1. Basic Operations

3.1.1 1.1. Python list to numpy array

```
In [10]: py_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

         np_array = np.array(py_list)
         np_array

Out[10]: array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])

In [11]: py_list = [[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]

         np_array= np.array(py_list)
         np_array

Out[11]: array([[ 1,  2,  3],
                [ 4,  5,  6],
                [ 7,  8,  9],
                [10, 11, 12]])
```

3.1.2 1.2. Slicing and Indexing

```
In [12]: print('First row:\t\t\t{}'.format(np_array[0]))
         print('First column:\t\t\t{}'.format(np_array[:, 0]))
         print('3rd row 2nd column element:\t{}'.format(np_array[2][1]))
         print('2nd onwards row and 2nd onwards column:\n{}'.format(np_array[1:, 1:]))
         print('Last 2 rows and last 2 columns:\n{}'.format(np_array[-2:, -2:]))
         print('Array with 3rd, 1st and 4th row:\n{}'.format(np_array[[2, 0, 3]]))

First row:                          [1 2 3]
First column:                           [ 1  4  7 10]
3rd row 2nd column element:          8
2nd onwards row and 2nd onwards column:
[[ 5  6]
 [ 8  9]
 [11 12]]
Last 2 rows and last 2 columns:
[[ 8  9]
 [11 12]]
Array with 3rd, 1st and 4th row:
[[ 7  8  9]
 [ 1  2  3]
 [10 11 12]]
```

3.1.3 1.3. Basic attributes of NumPy array

Get a full list of attributes of an ndarray object here.

```
In [13]: print('Data type:\t{}'.format(np_array.dtype))
         print('Array shape:\t{}'.format(np_array.shape))
```

```
Data type:          int64
Array shape:        (4, 3)
```

Let's create a function (with name `array_info`) to print the NumPy array, its shape, and its data type. We use this function to print arrays further in this section.

```
In [14]: def array_info(array):
             print('Array:\n{}'.format(array))
             print('Data type:\t{}'.format(array.dtype))
             print('Array shape:\t{}\n'.format(array.shape))

         array_info(np_array)
```

```
Array:
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
Data type:          int64
Array shape:        (4, 3)
```

3.1.4 1.4. Creating NumPy array using built-in functions and datatypes

The full list of supported data types can be found here.

Sequence Array

`np.arange([start, ]stop, [step, ]dtype=None)`

Return evenly spaced values in `[start, stop)`.

More delatis of the function can be found here.

```
In [15]: # sequence array
         array = np.arange(10, dtype=np.int64)
         array_info(array)
```

```
Array:
[0 1 2 3 4 5 6 7 8 9]
Data type:          int64
Array shape:        (10,)
```

```
In [16]: # sequence array
         array = np.arange(5, 10, dtype=np.float)
         array_info(array)
```

```
Array:
[5. 6. 7. 8. 9.]
Data type:        float64
Array shape:        (5,)
```

Zeroes Array

```
In [17]: # Zero array/matrix
         zeros = np.zeros((2, 3), dtype=np.float32)
         array_info(zeros)
```

```
Array:
[[0. 0. 0.]
 [0. 0. 0.]]
Data type:        float32
Array shape:        (2, 3)
```

Ones Array

```
In [18]: # ones array/matrix
         ones = np.ones((3, 2), dtype=np.int8)
         array_info(ones)
```

```
Array:
[[1 1]
 [1 1]
 [1 1]]
Data type:        int8
Array shape:        (3, 2)
```

Constant Array

```
In [19]: # constant array/matrix
         array = np.full((3, 3), 3.14)
         array_info(array)
```

```
Array:
[[3.14 3.14 3.14]
 [3.14 3.14 3.14]
```

```
 [3.14 3.14 3.14]]
Data type:        float64
Array shape:        (3, 3)
```

Identity Array

```
In [20]: # identity array/matrix
         identity = np.eye(5, dtype=np.float32)      # identity matrix of shape 5x5
         array_info(identity)
```

```
Array:
[[1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 1.]]
Data type:        float32
Array shape:        (5, 5)
```

Random Integers Array

`np.random.randint(low, high=None, size=None, dtype='l')`

Return random integer from the `discrete uniform` distribution in `[low, high)`. If high is `None`, then return elements are in `[0, low)`

More details can be found here.

```
In [21]: # random integers array/matrix
         rand_int = np.random.randint(5, 10, (2,3)) # random integer array of shape 2x3, value
         array_info(rand_int)
```

```
Array:
[[8 6 6]
 [5 7 9]]
Data type:        int64
Array shape:        (2, 3)
```

Random Array

`np.random.random(size=None)`

Results are from the `continuous uniform` distribution in `[0.0, 1.0)`.

These types of functions are useful is initializing the weight in Deep Learning. More details and similar functions can found here.

```
In [22]: # random array/matrix
         random_array = np.random.random((5, 5))   # random array of shape 5x5
         array_info(random_array)
```

```
Array:
[[0.63645816 0.24775572 0.18963118 0.7128101  0.90532966]
 [0.01083803 0.1958346  0.25679148 0.01552321 0.29276291]
 [0.3558123  0.67062147 0.68810298 0.63005767 0.61921669]
 [0.49418249 0.35856524 0.21676282 0.27337538 0.3862479 ]
 [0.70401416 0.34014999 0.67801031 0.89268371 0.76488405]]
Data type:        float64
Array shape:        (5, 5)
```

Boolean Array

If we compare above `random_array` with some `constant` or `array` of the same shape, we will get a boolean array.

```
In [23]: # Boolean array/matrix
         bool_array = random_array > 0.5
         array_info(bool_array)
```

```
Array:
[[ True False False  True  True]
 [False False False False False]
 [False  True  True  True  True]
 [False False False False False]
 [ True False  True  True  True]]
Data type:        bool
Array shape:        (5, 5)
```

The boolean array can be used to get value from the array. If we use a boolean array of the same shape as indices, we will get those values for which the boolean array is True, and other values will be masked.

Let's use the above `boolen_array` to get values from `random_array`.

```
In [24]: # Use boolean array/matrix to get values from array/matrix
         values = random_array[bool_array]
         array_info(values)
```

```
Array:
[0.63645816 0.7128101  0.90532966 0.67062147 0.68810298 0.63005767
 0.61921669 0.70401416 0.67801031 0.89268371 0.76488405]
Data type:        float64
Array shape:        (11,)
```

Basically, from the above method, we are filtering values that are greater than `0.5`.

Linespace

np.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None, axis=0)

Returns num evenly spaced samples, calculated over the interval [start, stop].

More detais about the function find here

```
In [25]: # Linspace
         linespace = np.linspace(0, 5, 7, dtype=np.float32)   # 7 elements between 0 and 5
         array_info(linespace)
```

```
Array:
[0.        0.8333333 1.6666666 2.5       3.3333333 4.1666665 5.       ]
Data type:        float32
Array shape:        (7,)
```

3.1.5 1.5. Data type conversion

Sometimes it is essential to convert one data type to another data type.

```
In [26]: age_in_years = np.random.randint(0, 100, 10)
         array_info(age_in_years)
```

```
Array:
[42 30 38 32 36 81 69 86 93 94]
Data type:        int64
Array shape:        (10,)
```

Do we really need an int64 data type to store age?

So let's convert it to uint8.

```
In [27]: age_in_years = age_in_years.astype(np.uint8)
         array_info(age_in_years)
```

```
Array:
[42 30 38 32 36 81 69 86 93 94]
Data type:        uint8
Array shape:        (10,)
```

Let's convert it to float128.

```
In [28]: age_in_years = age_in_years.astype(np.float128)
         array_info(age_in_years)
```

```
Array:
[42. 30. 38. 32. 36. 81. 69. 86. 93. 94.]
Data type:         float128
Array shape:         (10,)
```

3.2 2. Mathematical functions

Numpy supports a lot of Mathematical operations with array/matrix. Here we will see a few of them which are useful in Deep Learning. All supported functions can be found here.

3.2.1 2.1. Exponential Function

Exponential functions (also called `exp`) are used in neural networks as activations functions. They are used in softmax functions which is widely used in Classification tasks.
Return element-wise `exponential` of `array`.
More details of `np.exp` can be found **here**

```
In [29]: array = np.array([np.full(3, -1), np.zeros(3), np.ones(3)])
         array_info(array)

         # exponential of a array/matrix
         print('Exponential of an array:')
         exp_array = np.exp(array)
         array_info(exp_array)
```

```
Array:
[[-1. -1. -1.]
 [ 0.  0.  0.]
 [ 1.  1.  1.]]
Data type:         float64
Array shape:         (3, 3)

Exponential of an array:
Array:
[[0.36787944 0.36787944 0.36787944]
 [1.         1.         1.        ]
 [2.71828183 2.71828183 2.71828183]]
Data type:         float64
Array shape:         (3, 3)
```

3.2.2 2.2. Square Root

`np.sqrt` return the element-wise `square-root` (`non-negative`) of an array.
More details of the function can be found here

`Root Mean Square Error` (RMSE) and `Mean Absolute Error` (MAE) commonly used to measure the accuracy of continuous variables.

```
In [30]: array = np.arange(10)
         array_info(array)

         print('Square root:')
         root_array = np.sqrt(array)
         array_info(root_array)
```

```
Array:
[0 1 2 3 4 5 6 7 8 9]
Data type:        int64
Array shape:        (10,)

Square root:
Array:
[0.         1.         1.41421356 1.73205081 2.         2.23606798
 2.44948974 2.64575131 2.82842712 3.        ]
Data type:        float64
Array shape:        (10,)
```

3.2.3 2.3. Logrithm

`np.log` return element-wise natural logrithm of an array.

More details of the function can be found here

`Cross-Entropy` / `log loss` is the most commonly used loss in Machine Learning classification problem.

```
In [31]: array = np.array([0, np.e, np.e**2, 1, 10])
         array_info(array)

         print('Logrithm:')
         log_array = np.log(array)
         array_info(log_array)
```

```
Array:
[ 0.          2.71828183  7.3890561   1.         10.        ]
Data type:        float64
Array shape:        (5,)

Logrithm:
Array:
[      -inf 1.         2.         0.         2.30258509]
Data type:        float64
Array shape:        (5,)
```

```
/usr/lib/python3.7/site-packages/ipykernel_launcher.py:5: RuntimeWarning: divide by zero encou
  """
```

Note: Getting warning because we are trying to calculate `log(0)`.

3.2.4 2.4. Power

`numpy.power(x1, x2)`
Returns first array elements raised to powers from second array, element-wise.
Second array must be broadcastable to first array.
What is **broadcasting**? We will see later.
More detalis about the function can be found here

```
In [32]: array = np.arange(0, 6, dtype=np.int64)
         array_info(array)

         print('Power 3:')
         pow_array = np.power(array, 3)
         array_info(pow_array)
```

```
Array:
[0 1 2 3 4 5]
Data type:        int64
Array shape:        (6,)

Power 3:
Array:
[  0   1   8  27  64 125]
Data type:        int64
Array shape:        (6,)
```

3.2.5 2.5. Clip Values

`np.clip(a, a_min, a_max)`
Return element-wise cliped values between `a_min` and `a_max`.
More details of the finction can be found here
`Rectified Linear Unit` (ReLU) is the most commonly used activation function in Deep Learning.
What ReLU do?
If the value is less than zero, it makes it zero otherwise leave as it is. In NumPy assignment will be implementing this activation function using NumPy.

```
In [33]: array = np.random.random((3, 3))
         array_info(array)

         # clipped between 0.2 and 0.5
```

```
print('Clipped between 0.2 and 0.5')
cliped_array = np.clip(array, 0.2, 0.5)
array_info(cliped_array)

# clipped to 0.2
print('Clipped to 0.2')
cliped_array = np.clip(array, 0.2, np.inf)
array_info(cliped_array)
```

```
Array:
[[0.34681106 0.8493078  0.57783207]
 [0.59383869 0.31548905 0.05104628]
 [0.94695521 0.18246856 0.19085875]]
Data type:         float64
Array shape:         (3, 3)

Clipped between 0.2 and 0.5
Array:
[[0.34681106 0.5        0.5       ]
 [0.5        0.31548905 0.2       ]
 [0.5        0.2        0.2       ]]
Data type:         float64
Array shape:         (3, 3)

Clipped to 0.2
Array:
[[0.34681106 0.8493078  0.57783207]
 [0.59383869 0.31548905 0.2       ]
 [0.94695521 0.2        0.2       ]]
Data type:         float64
Array shape:         (3, 3)
```

3.3 3. Reshape ndarray

Reshaping the array / matrix is very often required in Machine Learning and Computer vision.

3.3.1 3.1. Reshape

`np.reshape` gives an array in new shape, without changing its data.
More details of the function can be found here

```
In [34]: a = np.arange(1, 10, dtype=np.int)
         array_info(a)

         print('Reshape to 3x3:')
         a_3x3 = a.reshape(3, 3)
```

```
array_info(a_3x3)

print('Reshape 3x3 to 3x3x1:')
a_3x3x1 = a_3x3.reshape(3, 3, 1)
array_info(a_3x3x1)
```

```
Array:
[1 2 3 4 5 6 7 8 9]
Data type:        int64
Array shape:        (9,)

Reshape to 3x3:
Array:
[[1 2 3]
 [4 5 6]
 [7 8 9]]
Data type:        int64
Array shape:        (3, 3)

Reshape 3x3 to 3x3x1:
Array:
[[[1]
  [2]
  [3]]

 [[4]
  [5]
  [6]]

 [[7]
  [8]
  [9]]]
Data type:        int64
Array shape:        (3, 3, 1)
```

3.3.2 3.2. Expand Dim

np.expand_dims

In the last reshape, we have added a new axis. We can use `np.expand_dims` or `np.newaxis` to do the same thing.

Mode details for `np.expand_dim` can be found here

```
In [35]: print('Using np.expand_dims:')
         a_expand = np.expand_dims(a_3x3, axis=2)
         array_info(a_expand)
```

```
        print('Using np.newaxis:')
        a_newaxis = a_3x3[..., np.newaxis]
        # or
        # a_newaxis = a_3x3[:, :, np.newaxis]
        array_info(a_newaxis)
```

```
Using np.expand_dims:
Array:
[[[1]
  [2]
  [3]]

 [[4]
  [5]
  [6]]

 [[7]
  [8]
  [9]]]
Data type:        int64
Array shape:        (3, 3, 1)

Using np.newaxis:
Array:
[[[1]
  [2]
  [3]]

 [[4]
  [5]
  [6]]

 [[7]
  [8]
  [9]]]
Data type:        int64
Array shape:        (3, 3, 1)
```

3.3.3 3.3. Squeeze

Sometimes we need to remove the redundant axis (single-dimensional entries). We can use `np.squeeze` to do this.

More details of `np.squeeze` can be found here

Deep Learning very often uses this functionality.

```
In [36]: print('Squeeze along axis=2:')
        a_squeezed = np.squeeze(a_newaxis, axis=2)
```

```
array_info(a_squeezed)

# should get value error
print('Squeeze along axis=1, should get ValueError')
a_squeezed_error = np.squeeze(a_newaxis, axis=1)  # Getting error because of the size
                                                  # axis-1 is not equal to one.
```

```
Squeeze along axis=2:
Array:
[[1 2 3]
 [4 5 6]
 [7 8 9]]
Data type:        int64
Array shape:        (3, 3)

Squeeze along axis=1, should get ValueError
```

```
---------------------------------------------------------------------------

ValueError                                Traceback (most recent call last)

<ipython-input-36-e96f455539e0> in <module>
      5 # should get value error
      6 print('Squeeze along axis=1, should get ValueError')
----> 7 a_squeezed_error = np.squeeze(a_newaxis, axis=1)  # Getting error because of the si
      8                                                   # axis-1 is not equal to one.


<__array_function__ internals> in squeeze(*args, **kwargs)


/usr/lib/python3.7/site-packages/numpy/core/fromnumeric.py in squeeze(a, axis)
   1481         return squeeze()
   1482     else:
-> 1483         return squeeze(axis=axis)
   1484
   1485


ValueError: cannot select an axis to squeeze out which has size not equal to one
```

Note: Getting error because of the size of axis-1 is not equal to one.

3.3.4 3.4. Reshape revisit

We have a 1-d array of length n. We want to reshape in a 2-d array such that the number of columns becomes two, and we do not care about the number of rows.

```
In [37]: a = np.arange(10)
         array_info(a)

         print('Reshape such that number of column is 2:')
         a_col_2 = a.reshape(-1, 2)
         array_info(a_col_2)
```

```
Array:
[0 1 2 3 4 5 6 7 8 9]
Data type:         int64
Array shape:         (10,)

Reshape such that number of column is 2:
Array:
[[0 1]
 [2 3]
 [4 5]
 [6 7]
 [8 9]]
Data type:         int64
Array shape:         (5, 2)
```

3.4 4. Combine Arrays / Matrix

Combining two or more arrays is a frequent operation in machine learning. Let's have a look at a few methods.

3.4.1 4.1. Concatenate

`np.concatenate`, Join a sequence of arrays along an existing axis.
More details of the function find here

```
In [38]: a1 = np.array([[1, 2, 3], [4, 5, 6]])
         a2 = np.array([[7, 8, 9]])

         print('Concatenate along axis zero:')
         array = np.concatenate((a1, a2), axis=0)
         array_info(array)
```

```
Concatenate along axis zero:
Array:
[[1 2 3]
```

```
 [4 5 6]
 [7 8 9]]
Data type:        int64
Array shape:        (3, 3)
```

3.4.2 4.2. hstack

`np.hstack`, stack arrays in sequence horizontally (column-wise).
More details of the function find here

```
In [39]: a1 = np.array((1, 2, 3))
         a2 = np.array((4, 5, 6))
         a_hstacked = np.hstack((a1,a2))

         print('Horizontal stack:')
         array_info(a_hstacked)
```

```
Horizontal stack:
Array:
[1 2 3 4 5 6]
Data type:        int64
Array shape:        (6,)
```

```
In [40]: a1 = np.array([[1],[2],[3]])
         a2 = np.array([[4],[5],[6]])
         a_hstacked = np.hstack((a1,a2))

         print('Horizontal stack:')
         array_info(a_hstacked)
```

```
Horizontal stack:
Array:
[[1 4]
 [2 5]
 [3 6]]
Data type:        int64
Array shape:        (3, 2)
```

3.4.3 4.3. vstack

`np.vstack`, tack arrays in sequence vertically (row-wise).
More details of the function find here

```
In [41]: a1 = np.array([1, 2, 3])
         a2 = np.array([4, 5, 6])
         a_vstacked = np.vstack((a1, a2))

         print('Vertical stack:')
         array_info(a_vstacked)
```

```
Vertical stack:
Array:
[[1 2 3]
 [4 5 6]]
Data type:        int64
Array shape:        (2, 3)
```

```
In [42]: a1 = np.array([[1, 11], [2, 22], [3, 33]])
         a2 = np.array([[4, 44], [5, 55], [6, 66]])
         a_vstacked = np.vstack((a1, a2))

         print('Vertical stack:')
         array_info(a_vstacked)
```

```
Vertical stack:
Array:
[[ 1 11]
 [ 2 22]
 [ 3 33]
 [ 4 44]
 [ 5 55]
 [ 6 66]]
Data type:        int64
Array shape:        (6, 2)
```

3.5 5. Element wise Operations

Let's generate a random number to show element-wise operations.

```
In [43]: a = np.random.random((4,4))
         b = np.random.random((4,4))
         array_info(a)
         array_info(b)
```

```
Array:
[[0.82705322 0.42162109 0.57763648 0.92588903]
 [0.22896344 0.22883764 0.12309067 0.92404846]
 [0.94787189 0.00673047 0.86440805 0.24037018]
```

```
 [0.65578633 0.25077389 0.08279757 0.43422053]]
Data type:        float64
Array shape:        (4, 4)

Array:
[[0.87704119 0.24751413 0.0583565  0.67261   ]
 [0.01344297 0.74007468 0.09248101 0.98681215]
 [0.02960068 0.85738299 0.54050414 0.49886616]
 [0.96294615 0.32490033 0.77393457 0.03797287]]
Data type:        float64
Array shape:        (4, 4)
```

3.5.1 5.1. Element wise Scalar Operation

Scalar Addition

```
In [44]: a + 5 # element wise scalar addition
```

```
Out[44]: array([[5.82705322, 5.42162109, 5.57763648, 5.92588903],
                [5.22896344, 5.22883764, 5.12309067, 5.92404846],
                [5.94787189, 5.00673047, 5.86440805, 5.24037018],
                [5.65578633, 5.25077389, 5.08279757, 5.43422053]])
```

Scalar Subtraction

```
In [45]: a - 5 # element wise scalar subtraction
```

```
Out[45]: array([[-4.17294678, -4.57837891, -4.42236352, -4.07411097],
                [-4.77103656, -4.77116236, -4.87690933, -4.07595154],
                [-4.05212811, -4.99326953, -4.13559195, -4.75962982],
                [-4.34421367, -4.74922611, -4.91720243, -4.56577947]])
```

Scalar Multiplication

```
In [46]: a * 10 # element wise scalar multiplication
```

```
Out[46]: array([[8.27053224, 4.21621092, 5.77636475, 9.25889033],
                [2.28963444, 2.2883764 , 1.2309067 , 9.24048455],
                [9.47871886, 0.06730469, 8.64408046, 2.40370181],
                [6.55786326, 2.50773895, 0.82797571, 4.34220533]])
```

Scalar Division

```
In [47]: a/10 # element wise scalar division
```

```
Out[47]: array([[0.08270532, 0.04216211, 0.05776365, 0.0925889 ],
                [0.02289634, 0.02288376, 0.01230907, 0.09240485],
                [0.09478719, 0.00067305, 0.0864408 , 0.02403702],
                [0.06557863, 0.02507739, 0.00827976, 0.04342205]])
```

3.5.2 5.2. Element wise Array Operations

Arrays Addition

```
In [48]: a + b # element wise array/vector addition
```

```
Out[48]: array([[1.70409442, 0.66913522, 0.63599298, 1.59849903],
               [0.24240641, 0.96891232, 0.21557168, 1.91086061],
               [0.97747256, 0.86411346, 1.40491219, 0.73923634],
               [1.61873248, 0.57567423, 0.85673214, 0.47219341]])
```

Arrays Subtraction

```
In [49]: a - b # element wise array/vector subtraction
```

```
Out[49]: array([[-0.04998797,  0.17410696,  0.51927998,  0.25327903],
               [ 0.21552047, -0.51123704,  0.03060966, -0.06276369],
               [ 0.91827121, -0.85065252,  0.32390391, -0.25849598],
               [-0.30715983, -0.07412644, -0.691137  ,  0.39624766]])
```

Arrays Multiplication

```
In [50]: a * b # element wise array/vector multiplication
```

```
Out[50]: array([[0.72535975, 0.10435718, 0.03370884, 0.62276222],
               [0.00307795, 0.16935694, 0.01138355, 0.91186224],
               [0.02805765, 0.00577059, 0.46721613, 0.11991255],
               [0.63148692, 0.08147652, 0.0640799 , 0.0164886 ]])
```

Arrays Division

```
In [51]: a / b # element wise array/vector division
```

```
Out[51]: array([[9.43003851e-01, 1.70342233e+00, 9.89840846e+00, 1.37656150e+00],
               [1.70322055e+01, 3.09208848e-01, 1.33098323e+00, 9.36397525e-01],
               [3.20219676e+01, 7.85001510e-03, 1.59926258e+00, 4.81833004e-01],
               [6.81020765e-01, 7.71848685e-01, 1.06982649e-01, 1.14350194e+01]])
```

We can notice that the dimension of both arrays is equal in above arrays element-wise operations. **What if dimensions are not equal.** Let's check!!

```
In [52]: print('Array "a":')
         array_info(a)
         print('Array "c":')
         c = np.random.rand(2, 2)
         array_info(c)
         # Should throw ValueError
         a + c
```

```
Array "a":
Array:
[[0.82705322 0.42162109 0.57763648 0.92588903]
 [0.22896344 0.22883764 0.12309067 0.92404846]
 [0.94787189 0.00673047 0.86440805 0.24037018]
 [0.65578633 0.25077389 0.08279757 0.43422053]]
Data type:        float64
Array shape:        (4, 4)

Array "c":
Array:
[[0.0105934  0.2421566 ]
 [0.91502164 0.39175169]]
Data type:        float64
Array shape:        (2, 2)
```

```
    ---------------------------------------------------------------------------

    ValueError                                Traceback (most recent call last)

    <ipython-input-52-0de9e8760d46> in <module>
      5 array_info(c)
      6 # Should throw ValueError
----> 7 a + c


    ValueError: operands could not be broadcast together with shapes (4,4) (2,2)
```

Oh got the ValueError!!
What is this error?
ValueError: operands could not be broadcast together with shapes `(4,4) (2,2)`
Let's see it next.

3.5.3 5.3. Broadcasting

There is a concept of broadcasting in NumPy, which tries to copy rows or columns in the lower-dimensional array to make an equal dimensional array of higher-dimensional array.

Let's try to understand with a simple example.

```
In [53]: a = np.array([[1, 2, 3], [4, 5, 6],[7, 8, 9]])
         b = np.array([0, 1, 0])

         print('Array "a":')
         array_info(a)
         print('Array "b":')
```

```
array_info(b)

print('Array "a+b":')
array_info(a+b)  # b is reshaped such that it can be added to a.


# b = [0,1,0] is broadcasted to     [[0, 1, 0],
#                                    [0, 1, 0],
#                                    [0, 1, 0]]  and added to a.
```

```
Array "a":
Array:
[[1 2 3]
 [4 5 6]
 [7 8 9]]
Data type:        int64
Array shape:        (3, 3)

Array "b":
Array:
[0 1 0]
Data type:        int64
Array shape:        (3,)

Array "a+b":
Array:
[[1 3 3]
 [4 6 6]
 [7 9 9]]
Data type:        int64
Array shape:        (3, 3)
```

3.6 6. Linear Algebra

Here we see commonly use linear algebra operations in Machine Learning.

3.6.1 6.1. Transpose

```
In [54]: a = np.random.random((2,3))
         print('Array "a":')
         array_info(a)

         print('Transose of "a":')
         a_transpose = a.transpose()
         array_info(a_transpose)
```

```
Array "a":
Array:
[[0.28182294 0.58825431 0.16374679]
 [0.20043516 0.16280755 0.78446435]]
Data type:        float64
Array shape:        (2, 3)

Transose of "a":
Array:
[[0.28182294 0.20043516]
 [0.58825431 0.16280755]
 [0.16374679 0.78446435]]
Data type:        float64
Array shape:        (3, 2)
```

3.6.2 6.2. Matrix Multiplication

We will discuss 2 ways of performing Matrix Multiplication.

- `matmul`
- Python `@` operator

Using matmul function in numpy This is the most used approach for multiplying two matrices using Numpy. See docs

In [55]:
```
a = np.random.random((3, 4))
b = np.random.random((4, 2))

print('Array "a":')
array_info(a)
print('Array "b"')
array_info(b)

c = np.matmul(a,b) # matrix multiplication of a and b

print('matrix multiplication of a and b:')
array_info(c)

print('{} x {} --> {}'.format(a.shape, b.shape, c.shape)) # dim1 of a and dim0 of b h
                                                            # same for matrix multiplicat
```

```
Array "a":
Array:
[[0.3072932  0.72392491 0.95212275 0.33296585]
 [0.16950383 0.72563226 0.84075636 0.6571445 ]
 [0.26920146 0.13012234 0.66836313 0.88143425]]
Data type:        float64
```

```
Array shape:        (3, 4)

Array "b"
Array:
[[0.02279685 0.17035125]
 [0.98251453 0.82352564]
 [0.67960531 0.3964682 ]
 [0.72476925 0.43454914]]
Data type:        float64
Array shape:        (4, 2)

matrix multiplication of a and b:
Array:
[[1.60666315 1.17069492]
 [1.76446901 1.2453467 ]
 [1.22704361 0.80102911]]
Data type:        float64
Array shape:        (3, 2)

(3, 4) x (4, 2) --> (3, 2)
```

Using `@` operator This method of multiplication was introduced in Python 3.5. See docs

```
In [56]: a = np.random.random((3, 4))
         b = np.random.random((4, 2))

         print('Array "a":')
         array_info(a)
         print('Array "b"')
         array_info(b)

         c = a@b # matrix multiplication of a and b
         array_info(c)
```

```
Array "a":
Array:
[[0.70821105 0.08868542 0.85941756 0.48417174]
 [0.07735007 0.52167753 0.27974037 0.79268077]
 [0.14792952 0.73619868 0.70837623 0.22259353]]
Data type:        float64
Array shape:        (3, 4)

Array "b"
Array:
[[0.32764555 0.74126452]
 [0.10788242 0.92392647]
 [0.74803993 0.72371475]
```

```
 [0.09866666 0.61692886]]
Data type:         float64
Array shape:         (4, 2)

Array:
[[0.93226006 1.52758321]
 [0.36909137 1.23080841]
 [0.67974761 1.43983505]]
Data type:         float64
Array shape:         (3, 2)
```

3.6.3 6.3. Inverse

```
In [57]: A = np.random.random((3,3))
         print('Array "A":')
         array_info(A)
         A_inverse = np.linalg.inv(A)
         print('Inverse of "A" ("A_inverse"):')
         array_info(A_inverse)

         print('"A x A_inverse = Identity" should be true:')
         A_X_A_inverse = np.matmul(A, A_inverse)  # A x A_inverse = I = Identity matrix
         array_info(A_X_A_inverse)
```

```
Array "A":
Array:
[[0.93065448 0.4431653  0.85616747]
 [0.85776549 0.69236852 0.97069191]
 [0.30205686 0.73201065 0.84577822]]
Data type:         float64
Array shape:         (3, 3)

Inverse of "A" ("A_inverse"):
Array:
[[ -2.46688904   4.97268524  -3.20991381]
 [ -8.53328024  10.43311104  -3.33588579]
 [  8.26646106 -10.80564833   5.21588316]]
Data type:         float64
Array shape:         (3, 3)

"A x A_inverse = Identity" should be true:
Array:
[[ 1.00000000e+00  5.10068159e-16 -2.46483575e-16]
 [ 9.59993606e-16  1.00000000e+00  1.05375834e-16]
 [ 5.56200097e-16 -6.00320283e-16  1.00000000e+00]]
Data type:         float64
```

```
Array shape:         (3, 3)
```

3.6.4 6.4. Dot Product

```
In [58]: a = np.array([1, 2, 3, 4])
         b = np.array([5, 6, 7, 8])

         dot_pro = np.dot(a, b)  # It will be a scalar, so its shape will be empty
         array_info(dot_pro)
```

```
Array:
70
Data type:         int64
Array shape:         ()
```

3.7 7. Array statistics

3.7.1 7.1. Sum

```
In [59]: a = np.array([1, 2, 3, 4, 5])

         print(a.sum())
```

```
15
```

3.7.2 7.2. Sum along Axis

```
In [60]: a = np.array([[1, 2, 3], [4, 5, 6]])
         print(a)
         print('')

         print('sum along 0th axis = ',a.sum(axis = 0)) # sum along 0th axis ie: 1+4, 2+5, 3+6
         print("")
         print('sum along 1st axis = ',a.sum(axis = 1)) # sum along 1st axis ie: 1+2+3, 4+5+6
```

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

sum along 0th axis =  [5 7 9]

sum along 1st axis =  [ 6 15]
```

3.7.3 7.3. Minimum and Maximum

```
In [61]: a = np.array([-1.1, 2, 5, 100])

         print('Minimum = ', a.min())
         print('Maximum = ', a.max())
```

```
Minimum =  -1.1
Maximum =  100.0
```

3.7.4 7.4. Min and Max along Axis

```
In [62]: a = np.array([[-2, 0, 2], [1, 2, 3]])

         print('a =\n',a,'\n')
         print('Minimum = ', a.min())
         print('Maximum = ', a.max())
         print()
         print('Minimum along axis 0 = ', a.min(0))
         print('Maximum along axis 0 = ', a.max(0))
         print()
         print('Minimum along axis 1 = ', a.min(1))
         print('Maximum along axis 1 = ', a.max(1))
```

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

Minimum =  -2
Maximum =  3

Minimum along axis 0 =  [-2  0  2]
Maximum along axis 0 =  [1 2 3]

Minimum along axis 1 =  [-2  1]
Maximum along axis 1 =  [2 3]
```

3.7.5 7.5. Mean and Standard Deviation

```
In [63]: a = np.array([-1, 0, -0.4, 1.2, 1.43, -1.9, 0.66])

         print('mean of the array = ',a.mean())
         print('standard deviation of the array = ',a.std())
```

```
mean of the array =  -0.001428571428571414
standard deviation of the array =  1.1142252730860458
```

3.7.6 7.6. Standardizing the Array

Make distribution of array elements such that`mean=0` and `std=1`.

```
In [64]: a = np.array([-1, 0, -0.4, 1.2, 1.43, -1.9, 0.66])

         print('mean of the array = ',a.mean())
         print('standard deviation of the array = ',a.std())
         print()

         standardized_a = (a - a.mean())/a.std()
         print('Standardized Array = ', standardized_a)
         print()

         print('mean of the standardized array = ',standardized_a.mean()) # close to 0
         print('standard deviation of the standardized  array = ',standardized_a.std()) # equa
```

```
mean of the array =  -0.001428571428571414
standard deviation of the array =  1.1142252730860458

Standardized Array =  [-8.96202458e-01  1.28212083e-03 -3.57711711e-01  1.07826362e+00
  1.28468507e+00 -1.70393858e+00  5.93621943e-01]

mean of the standardized array =  -3.172065784643304e-17
standard deviation of the standardized  array =  1.0
```

4 References

https://numpy.org/devdocs/user/quickstart.html
https://numpy.org/devdocs/user/basics.types.html
https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.astype.html
https://coolsymbol.com/emojis/emoji-for-copy-and-paste.html
https://docs.scipy.org/doc/numpy-1.13.0/reference/routines.math.html
https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.exp.html#numpy.exp
https://docs.scipy.org/doc/numpy/reference/generated/numpy.clip.html
https://docs.scipy.org/doc/numpy/reference/generated/numpy.sqrt.html
https://docs.scipy.org/doc/numpy/reference/generated/numpy.log.html
https://docs.scipy.org/doc/numpy/reference/generated/numpy.power.html
https://docs.scipy.org/doc/numpy/reference/generated/numpy.reshape.html
https://docs.scipy.org/doc/numpy/reference/generated/numpy.expand_dims.html
https://docs.scipy.org/doc/numpy/reference/generated/numpy.squeeze.html
https://docs.scipy.org/doc/numpy/reference/generated/numpy.concatenate.html
https://docs.scipy.org/doc/numpy/reference/generated/numpy.hstack.html#numpy.hstack
https://docs.scipy.org/doc/numpy/reference/generated/numpy.vstack.html#numpy.vstack