Intro to NumPy for Data Computations

What's in this notes:

You can run all the codes in this document on Google Colab through this link. You can also find other 30+ machine learning notebooks that cover data visualization and analysis, classical machine learning, computer vision and natural language processing.

These NumPy notes were compiled by Jean de Dieu Nyandwi. You can find him on Twitter and LinkedIn.

Overview

The prime ingredient for a Machine Learning project is data and the most work is often spent in working with data. In this document, we will cover the basics of NumPy, a powerful tool for performing computations on data. To understand how powerful NumPy is, most Python Machine Learning frameworks are built on top of NumPy. Examples are Seaborn, Pandas, Scikit-Image and OpenCV.

The rest of this document is going to be about the useful skills you will need in day to day machine learning work, typically working with arrays.

NumPy Vs Data

In Machine Learning, there are three main types of data that we deal with: images, texts, and structured data.

Image: Example of three types of data: Images, texts, and structured data.

All of these types have one thing in common. They are made of an *array* of numbers. The image of cat shown is made of pixel numbers, each pixel value ranging from 0 to 255, a tweet containing a text can be represented in an array of numbers, and the features in a tabled data can be represented as an array of numbers. Pretty much all types of data can be represented as an array.

Given how almost all types of data (even video, which is a sequence of images) can be converted to an array of numbers, there is a need to know how to work with an array. In this chapter, we will learn how to create an array in NumPy and other basic operations that are helpful in data computation and processing. Although we will cover it in the latter chapters, data processing is an important part in any Machine Learning project.

NumPy Array

An array in NumPy can either be a vector or a matrix. A vector is a one dimensional array, whereas a matrix has two or more dimensions.

If you are using Google Colab, you do not need to install NumPy. We just need to import it as follows.

```
Import numpy as np
```

This is how we create a 1 dimensional (1D) array or a vector in NumPy.

```
array_1d = np.array([1,2,3,4,5])
print(array_1d)

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

We can also create a 2D array or a matrix by adding more vectors inside the array definition.

```
array_2d = np.array([ [1,2,3,4,5], [6,7,8,9,10] ])

print(array_2d)

[[ 1,  2,  3,  4,  5],
 [ 6,  7,  8,  9, 10]]
```

You can also create an array from a Python list. A list in Python is a data structure which can hold different data of different types inside **[.....]**. This is how you can create an array from a Python list.

```
num_list = [1,2,3,4,5]
arr = np.array(num_list)
print(arr)

[1 2 3 4 5]
```

All of the above were ways to create an array from scratch. Often you do not need to go from scratch. In the next section, you are going to see various ways to create specific arrays.

Different Techniques to Generate an Array

NumPy provides various options for creating specific arrays such as identity array, zero or one zero array, a range of values in a given interval, and a random array. They are all easy to create.

Let's start with generating an identity array. Identity array is a 2D array having 1s in the diagonal. The integer value you provide will be the size of that particular array.

```
identity_array = np.identity(4)
print(identity_array)

[[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]]
```

You can also generate the same array with the function *np.eye(4)*. If you want to have different numbers other than 1s, you can multiply a constant to the array, as follows.

```
identity_array = np.identity(4) * 7
print(identity_array)

[[7. 0. 0. 0.]
 [0. 7. 0. 0.]
 [0. 0. 7. 0.]
 [0. 0. 0. 7.]]
```

To generate an array with zeros or ones with a chosen size, we can use the function *np.ones()* or *np.zeros()*. We will only have to provide a *tuple ()* of the size of the array, first digit being the number of rows, and the last being the columns. Take an example, for creating the zeros array of 5 by 6.

```
zero_array = np.zeros((5,6))
print(zero_array)

[[0., 0., 0., 0., 0., 0.],
 [0., 0., 0., 0., 0., 0.],
 [0., 0., 0., 0., 0., 0.],
 [0., 0., 0., 0., 0., 0.],
 [0., 0., 0., 0., 0., 0.]]
```

Now that we have looked into creating an identity, zeros and ones array, let's see how we can create an array of numbers in an interval. We will achieve this using *np.arange()* function.

```
arr = np.arange(0, 5)
print(arr)
[0, 1, 2, 3, 4]

## Note how 5 is not part of the output array
## The default step size is 1. We can control it
arr = np.arange(0, 20,2)
print(arr)

[ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18]
```

In case we want to generate an array in a given interval, but with even spaced numbers, we can use *np.linspace()*. This is how we would go about it. We are creating a 1D array between 0 and 100 but evenly divided by 5.

```
arr = np.linspace(0, 100, 5)
print(arr)

[  0.,  25.,  50.,  75., 100.]
```

So far, we have created an array from various options. Oftentimes, we want to create an array with random numbers. NumPy offers different ways to help us with that.

```
# 1D random array
random_arr = np.random.rand(4)
print(random_arr)

[0.45321517, 0.25281992, 0.69909098, 0.90542031]

# 2D random array
random_arr = np.random.rand(4, 5)
print(random_arr)

[[0.60010644, 0.78050133, 0.17062182, 0.06617089, 0.37704404],
 [0.55344506, 0.34656887, 0.10235701, 0.74557143, 0.42350861],
 [0.41219881, 0.24338808, 0.1523242 , 0.30722854, 0.73695501],
 [0.07837244, 0.20150229, 0.32618776, 0.15177508, 0.03581211]]
```

Same as we had the option to create an array in a given interval using *np.arange(),* we can also create a random array in a given interval. Let us generate a random array having 10 random numbers in an interval between 5 and 50.

```
arr = np.random.randint(5, 50, 10)
print(arr)

[37, 31, 29, 48, 45, 28, 40, 20, 49, 47]

## To generate one integer in an interval (5,50)

arr = np.random.randint(5, 50)
print(arr)
8
```

That's all for creating a random array. One thing to note is that by running these codes, you may not get the same values. They are random at each run time. In case you want to keep the value every time you run the code, you could insert *a random seed* line in a cell. Everytime you can run the following code, you will always get the same number, as long as there is a seed, and you do not change the seed value.

```
Import random
random.seed(10)

random.randint(5, 50)

41
```

That's all for creating an array. How would we select a set or a slice of values in an array? That's what the next topic covers.

Data Selection: Indexing and Slicing An Array

1D and 2D Array Selection

When we have data in an array, we want to be able to grab a given range or values we want. While indexing is selecting an individual value in an array, slicing is selecting a group of values in rows or columns or both.

Selecting values in the 1D array is quite straight. This is how you would select a single element in an array. You only have to provide an index of the value you want to return. This is how you would go about it.

```
Array_1d = np.array([1,2,3,4,5])
## Indexing, returning a single value at index 2
## Index starts at 0

array_1d[2]
3

## Indexing the last value of the array
array_1d[-1]
5

## Slicing, returning group of values
array_1d[2:4]

array([3, 4])
```

That was for the 1D array. For 2D it might be a little complicated since you are selecting data between multiple rows and columns, but it is not that complex when you look into it.

```
Import random
random.seed(10)

random.randint(5, 50)

41
```

```
array_2d = np.array([[1,2,3], [4,5,6], [7,8,9]])

## Selecting an individual element in 2D array
## Returning a value at row 1 and column 2

array_2d[1][2]
6
## Selecting the whole row
array_2d[2]
array([4, 5, 6])

## Selecting the whole column
array_2d[:, 2]
array([3, 6, 9])
```

You can wish to select specific rows and columns at the same time. If you want that, you can use a *comma (,)* to separate the rows and columns of interests.To group either columns or rows you can use *a colon (:).*

```
## Selecting all first two rows and first two columns

array_2d[0:2, 0:2]
array([[1, 2],
       [4, 5]])
```

There are more examples on how to select data in a 2D array in the notebook associated with this chapter. If that was not enough, take some time to practice it and come back to do some conditional selection using comparison operators.

Conditional data selection

If you want to select the data in an array based on the condition, it is possible. Below are the examples in which we return the values in an array using a comparison operator.

```
## In 2D array we defined early, return all values less than 6

array_2d[ array_2d < 6 ]
array([1, 2, 3, 4, 5])

## Return all even numbers (numbers divisible by 2)

array_2d[ array_2d % 2 == 0 ]

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

That was about indexing and slicing data in an array. On our goal to study the basics of NumPy, we are going to learn about the useful mathematical and other basis operations.

For now, you may be tempted to jump into Machine Learning, but you will often need to understand the underlying tools like NumPy.

Mathematical and other Basic Operations

Arithmetic Operation

Earlier in this chapter, we saw how to create an array and select values in it. We may also like to perform basic arithmetic operations such as addition, multiplication, subtraction, and division.

The good thing is that it is not as different from the first time you learned how to add two numbers. Below is how you can add or subtract two arrays.

```
arr1 = np.arange(0, 5)    # [0,1,2,3,4]
arr2 = np.arange(6,11)   #[6,7,8,9,10]

## Addition

arr1 + arr2
array ([ 6, 8, 10, 12, 14 ])

## Subtraction

arr1 - arr2
array ([ -6, -6, -6, -6, -6 ])
```

When performing any arithmetic operation, because the computation is element wise (corresponding elements are computed together), it is a precaution to make sure that the both arrays under operation have the same size or shape.

To multiply or divide the array, the process remains the same.

```
## Multiplication

arr1 *  arr2
array ( [ 0, 7, 16, 27, 40])

## Division

arr1 / arr2
array ( [ 0., 0.142, 0.25, 0.333, 0.4 ])
```

Universal Functions

NumPy universal functions *(ufunc*) allows us to compute various mathematical, trigonometric, logical and comparison operations. There are plenty of functions in these operations domains and using them is quite simple.

Ufunc provides functions to implement the arithmetic operations. Using these functions will provide the same results as we calculated before: *np.add(arr1, arr2), np.subtract(arr1, arr2), np.multiply(arr1, arr2), and np.divide(arr1,arr2).*

Comparison functions return *true* or *false*. Let's say you want to compare values in two arrays we defined earlier to return false or true if wise elements are greater or less than one other.

```
## Comparison functions

## Each value in arr1 is less than arr2

np.greater(arr1, arr2)
array ( [ False, False, False, False, False])

np.less(arr1, arr2)
array ( [ True, True, True, True, True])
```

There are many functions that we are not covering here such as logarithmic, trigonometric, etc...For a full list of these functions, check the NumPy ufuncs in official documentation.

Basic Statistics

NumPy provides functions and methods suitable for statistical computations. Some of the basic stats functions you might frequently need when working with arrays are mean, median, variance, and standard deviation.

Mean is usually calculated by summing all values over the number of values. It can also be referred to as the average value of the array. Median on the other hand, is the middle value of the array.

```
arr  = np.arange(0, 5)    # [0,1,2,3,4]

## Mean

np.mean(arr)
2.0

## Median

np.median(arr)
2.0
```

Standard deviation of the array is how much each array element deviates from the mean value of the array. On the other hand, variance is the square of the standard deviation.

```
arr  = np.arange(0, 5)    # [0,1,2,3,4]

## Standard Deviation

np.std(arr)
1.414

## Variance
## 1.414 * 1.414 ~=2.0
np.var(arr)
2.0
```

Note that this is not a deep dive into statistics. If you would learn to learn more about stats, you would love the great book *Think Stats*, which is available here for free. In the next and final section of NumPy basics, we are going to see how to manipulate value in arrays.

Array Manipulation

Data manipulation is an important step in machine learning project workflow. NumPy provides various functionalities for manipulating arrays. In this last section of intro to NumPy, we are going to look into all these functionalities.

Shaping the Array

If you have created an array and you want to *reshape* it, it is so simple with NumPy. Using np.reshape(array_name, newshape=(rows, columns) or array_name.reshape(rows, columns), you can change the shape of the array. The rows and columns of the new shape have to conform with the existing data of the array. Otherwise, it won't work. Take an example, you can convert (3,3) array into (1,9) but you can't convert it into (5,5).

Let's first see how to find the shape of an array and we will follow with reshaping these particular arrays.

```
arr1 = np.arange(0, 10)
arr2 = np.array ([ [1,2,3],  [4,5,6],  [7,8,9] ])

arr1
array ([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

arr2
array ([ [1, 2, 3],
         [4, 5, 6],
         [7, 8, 9] ])

np.shape(arr1)
(10, )

np.shape(arr2)
(3, 3)

# arr2.shape will give the same results
```

```
# arr1 is (10,): 10 rows, 1 column. Let's reshape it into (5,2)
# arr1.reshape(5,2) would also work
np.reshape(arr1, newshape = (5, 2))
array([ [0, 1],
        [2, 3],
        [4, 5],
        [6, 7],
        [8, 9] ])

arr2_reshaped = arr2.reshape(9,1)
arr2_reshaped.T  # The output array will be transposed(.T), columns are changed into rows, vise-versa
array ([ [1, 2, 3, 4, 5, 6, 7, 8, 9] ])
```

We can also use np.resize() to change the shape of the array.

```
np.resize(arr2, (1,9) )
array ( [ [1, 2, 3, 4, 5, 6, 7, 8, 9] ] )
```

Joining Arrays

At times, we may want to join or concatenate arrays that we have created. We can achieve that with np.concatenate().

```
arr1 = np.array ([ [1,2,3],  [4,5,6],  [7,8,9] ])
arr2 = np.array( [[10,11,12]] )

# Joining these two arrays

np.concatenate((arr1,  arr2))

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

When joining two arrays, setting the *axis* parameter to None will flatten the array. The output array will be a single vector. As you will see later in Deep Learning, there is often a flattening layer whose purpose is to convert the high dimensional output of prior layers to one single column vector in order to meet the format expected by the next layer, in this case being densely connected layers.

```
# Setting the axis to None flatten the array
np.concatenate((arr1, arr2), axis=None)
array ( [ [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 ] ] )
```

One last thing about joining arrays: We can join two 1D arrays to make a 2D array. This is called *stacking,* and it can be on either column or row, hence *Column Stacking* and *Row Stacking.*

```
arr1 = np.arange (0, 6)
arr2 = np.arange (6, 12)
# Column stacking
np.column_stack((arr1,  arr2))
array ([ [ 0,  6],
         [ 1,  7],
        [ 2,  8],
       [ 3,  9],
      [ 4, 10],
     [ 5, 11] ])

# Row stacking
np.row_stack((arr1,  arr2))

array( [ [ 0,  1,  2,  3,  4,  5],
      [ 6,  7,  8,  9, 10, 11] ] )
```

Splitting Arrays

With NumPy's flexibility, splitting an array is very straight. Using np.split (array_name, number_of_splits), you can split the specified array into a number of sub arrays.

```
# Splitting an array
np.split(arr1, 3) # arr1...array ([0,1,2,3,4,5])
[array ([0, 1]), array ([2, 3]), array ([4, 5])]
```

Adding and Repeating Array's Values

Like Python list, we can also append elements to an array after we have created it. This is important because you will not need to recreate the array everytime you want to add new elements.

```
# Adding new element to an array
np.append(arr1, 6) # arr1...array ([0,1,2,3,4,5])
array([0, 1, 2, 3, 4, 5, 6])
```

You can also add or repeat the array to itself a number of times. To repeat each single value in an array (value after another), you can use np.repeat().

```
# Self repeating array
np.tile(arr1, 2) # arr1...array ([0,1,2,3,4,5])
array ( [0, 1, 2, 3, 4, 5, 0, 1, 2, 3, 4, 5] )

# Repeating each single value after itself
np.repeat(arr1, 3)
array ( [0, 0, 0, 1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4, 5, 5, 5] )
```

Sorting Array

Whenever the array is not arranged in an order, you can sort it with np.sort(). To sort a 1D array, you do not need to set the axis. For sorting a 2D array, you would have to specific the axis parameter. Always setting the axis to *None* will flatten the array. To learn more on sorting the 2D arrays, check out numpy.sort.

```
# Sorting 1D array
arr = np.array ( [ [1,2,3,4,5,3,2,1,3,5,6,7,7,5,9,5] ] )

np.sort (arr)

array ( [[1, 1, 2, 2, 3, 3, 3, 4, 5, 5, 5, 5, 6, 7, 7, 9]] )
```

In the above array we used for sorting, we can return the unique values. As simple as all the functions we tried, this is no difference.

```
# Finding the unique values in messy array
np.unique(arr)
array( [1, 2, 3, 4, 5, 6, 7, 9] )
```

Reversing an Array

Before we close this chapter, let's also see how to reverse an array. Using flip functions provided by NumPy, we can either flip the array up/down or left/right.

```
>>> arr = np.array ([ [1,2,3],  [4,5,6],  [7,8,9] ])
>>> arr
array(  [[1, 2, 3],
         [4, 5, 6],
         [7, 8, 9]])

# Up/down flipping
>>> np.flipud(arr)

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

# Left/right  flipping

>>> np.fliplr(arr)

array([[3, 2, 1],
       [6, 5, 4],
       [9, 8, 7]])
```

That's for array manipulation! Understanding all these array possibilities will help you in your career when it comes to data manipulation. Remember, all kinds of real-world data can be converted into arrays, and so it's good to know how to move around arrays.

Final Notes

This document has provided you with all the basics skills of NumPy. You have learned how to create an array from scratch, generate an array with existing functions, perform mathematical operations and statistics as well. You also learned how to manipulate an array, which is one of the important skills to have as someone who works with data.