zekelabs / AI102-MachineLearning

Branch: master

AI102-MachineLearning / 1. Introduction to Numpy.ipynb

Find file Copy path

crazymuse Editing Awantiks content

e6c7288 21 days ago

1 contributor

1057 lines (1056 sloc) 30.8 KB

1. Introduction to Numpy

Numpy is a Library in python that specializes in dealing with multidimensional Arrays. The cool features of Numpy are

- **Automatic Checking :** Numpy ndArrays automatically check the consistancy of data. For instance, it is not possible to have 1st row with 2 elements and 2nd row with 3 elements
- **Contiguous Storage :** Unlike Python Lists, Numpy stores the data in contiguous Memory Locations, leading to lesser Space
- **Faster Vector Arithmatics :** Because of contiguous storage, the operations are performed faster as compared to default Python Execution for Lists

In [1]:
```
# Importing Numpy
import numpy as np
```

1.1 Initialization of 1D array in Python

A numpy array comes with 2 important state variables. Just like Python, it automatically detects dtype (if not mentioned)

- dtype
- shape

1.1.1 Initialization from python List

numpy.array(list **)**

In [12]:
```
# Initializing from Python List
v = np.array([1,4,9,3,2])
print ('1D array in numpy is %s\n'%v)
print ('dtype of the numpy array is %s\n'%str(v.dtype))
print ('shape of the numpy array is %s\n'%v.shape)
```

```
1D array in numpy is [1 4 9 3 2]

dtype of the numpy array is int32

shape of the numpy array is 5
```

1.1.2 Initialization via arange

numpy.arange(**[**start, **]**stop, **[**step, **]** dtype=None)

In [22]:
```
v1 = np.arange(5)
print ('Creating a numpy array via arange (stop=5) : %s\n'%v1)
v2 = np.arange(2,5)
print ('Creating a numpy array via arange (start=2,stop=5) : %s\n'%v2)
v3 = np.arange(0,-10,-2)
print ('Creating a numpy array via arange (start=0,stop=-10,step=-2) : %s\n'%v3)
```

```
Creating a numpy array via arange (stop=5) : [0 1 2 3 4]

Creating a numpy array via arange (start=2,stop=5) : [2 3 4]

Creating a numpy array via arange (start=0,stop=-10,step=-2) : [ 0 -2 -4 -6 -8]
```

1.2. Initialization of 2D array in Python

When we talk about 2D array, it is important to note that Numpy stores Matrix is ***Row Major Format***. Row major format means that the the complete row will be stored first and then the next row will be stored and so on. You can choose to store a matrix in colum major format by mentioning ***order='F'*** on ndarray creation, which means ***Column Major Format*** or *Fortran Style Format*.

Interpretation of row and column in Numpy

1.2.1 Initialization from List of List

numpy.array **(** ***object***, dtype=None **)**

Here the matrix that we have taken is

$$mat = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

In [31]:
```
pymat = [[1,2,3],[4,5,6]]
npmat = np.array(pymat)
npmat = np.array(pymat)
print ('numpy matrix is \n%s\n'%npmat)
print ('Flattened version of numpy matrix is %s\n'%npmat.flatten())
# By flatten the matrix becomes row major 1D vector (This is the way in which a matrix is stored in memor
y).
```

```
numpy matrix is
[[1 2 3]
 [4 5 6]]

Flattened version of numpy matrix is [1 2 3 4 5 6]
```

1.2.2 Initialization with zero/ones

numpy.**zeros**(*shape*, dtype=float)
numpy.**ones**(*shape*, dtype=float)

In [5]:
```
mat_zeros = np.zeros(shape=(3,5))
print ('Zeros Matris of shape %s is \n%s\n'%(mat_zeros.shape,mat_zeros))
mat_ones = np.ones(shape=(2,3))
print ('Ones Matris of shape %s is \n%s\n'%(mat_ones.shape,mat_ones))
```

```
Zeros Matris of shape (3, 5) is
[[0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]]

Ones Matris of shape (2, 3) is
[[1. 1. 1.]
 [1. 1. 1.]]
```

1.2.3 Initialization with Random Values

1.2.3.1 numpy.**random.random**(*size=None*,)
1.2.3.2 numpy.**random.randint**(low, high, *size=None*, dtype='l') : The value of matrix lies between low and high-1
1.2.3.3 numpy.**random.randn**(d_0, d_1, $\cdots$, d_n)

In [19]:
```
#Using numpy.random.random
```

```
mat1 = np.random.random(size=(3,4))
print ('matrix generated from numpy.random.random is \n%s\n'%mat1)
mat2 = np.random.randint(low=0,high=2,size=(3,4))
print ('matrix generated from numpy.random.random is \n%s\n'%mat2)
mat3 = np.random.randn(3,4)
print ('matrix generated from numpy.random.randn is \n%s\n'%mat3)
```

```
matrix generated from numpy.random.random is
[[0.40809381 0.92255528 0.26384768 0.18020434]
 [0.78683826 0.51616332 0.20181108 0.44676456]
 [0.40513581 0.41943719 0.14935848 0.49202397]]

matrix generated from numpy.random.random is
[[1 0 1 0]
 [0 1 1 1]
 [0 1 1 0]]

matrix generated from numpy.random.randn is
[[ 1.23093258  0.06762907  1.32948035  1.27069921]
 [-1.55068329 -0.48071094  0.85203508 -1.90068409]
 [-1.94335643  0.12680395  0.06573786 -0.66923407]]
```

1.3. Slicing and Indexing in Numpy

Just like python, numpy also has 0 indexing. Let us see some of the commonly used slicing techniques .

- Generic Slicing Operation : [start]:[end]:[jump]
- Only jump **::2**
- Only end **:5**
- Start and jump **2::-1**
- End and Jump **:5:2**
- Start, end and jump **2:7:3**

1.3.1 Remove n elements

vec[:-*n*]

In [29]:
```
vec = np.arange(10)
```

```
vec1 = vec[:-3]
print ('Result of removing last 3 elements from range(10) : \n%s\n'%vec1)
```

```
Result of removing last 3 elements from range(10) :
[0 1 2 3 4 5 6]
```

1.3.2 Access elements at even indices in a 1D array

vec[::2]

```
In [87]: vec1 = np.arange(0,20,3)
         print ('Original array is %s\n'%vec1)
         vec2 = vec1[::2]
         print ('Elements at even indices are %s\n'%vec2)
```

```
Original array is [ 0  3  6  9 12 15 18]

Elements at even indices are [ 0  6 12 18]
```

1.3.3 Access elements at indices in reverse order

vec[::-1]

```
In [88]: vec1 = np.arange(0,20,3)
         print ('Original array is %s\n'%vec1)
         vec2 = vec1[::-1]
         print ('Elements for indices in reverse is %s\n'%vec2)
```

```
Original array is [ 0  3  6  9 12 15 18]

Elements for indices in reverse is [18 15 12  9  6  3  0]
```

1.3.4 Access elements present for a range of indices

vec[a:b]

```
In [92]: vec1 = np.arange(0,20,3)
         print ('Original array is %s\n'%vec1)
         vec2 = vec1[2:5]
         print ('Elements for indices 2:5 is %s\n'%vec2)
```

```
Original array is [ 0  3  6  9 12 15 18]

Elements for indices 2:5 is [ 6  9 12]
```

1.3.5 Access a particular set of index given by a list

vec[idxlist]

```
In [95]: idx=[0,1,5]
         vec1 = np.arange(0,20,3)
         print ('Original array is %s\n'%vec1)
         vec2 = vec1[idx]
         print ('Subarray constructed by indices %s is %s\n'%(idx,vec2))
```

```
Original array is [ 0  3  6  9 12 15 18]

Subarray constructed by indices [0, 1, 5] is [ 0  3 15]
```

1.3.6 Creating a submatrix

```
In [109]: mat = np.random.randint(0,6,(3,5))
          # Create a submatrix with first 2 rows and last 2 columns
          submat1 = mat[0:2,-2:]
          print ('Original Matrix is \n%s\n'%mat)
          print ('Sub Matrix with first 2 rows and last 2 columns is \n%s\n'%submat1)
          submat2 = mat[:,3:0:-1]
          print ('After flipping the columns of the matrix, it looks : \n%s\n'%submat2)
```

```
Original Matrix is
[[1 2 0 0 2]
 [4 1 5 3 2]
 [2 1 3 5 0]]
```

```
Sub Matrix with first 2 rows and last 2 columns is
[[0 2]
 [3 2]]

After flipping the columns of the matrix, it looks :
[[0 0 2]
 [3 5 1]
 [5 3 1]]
```

1.3.7 Horizontal Matrix splitting

numpy.**hsplit**(ary, *indices_or_sections*)

hsplit basically splits a matrix across the horizontal plane based on the indices. Do note that the ***number of rows*** always remains ***constant*** in each section after the horizontal splitting.

```
In [131]: mat = np.random.randint(0,6,(3,7))
          sp1,sp2,sp3 = np.hsplit(mat,[4,6])
          print ('Original matrix of shape %s, is \n%s\n'%(mat.shape,mat))
          print ('First split of shape %s, is \n%s\n'%(sp1.shape,sp1))
          print ('Second split of shape %s, is \n%s\n'%(sp2.shape,sp2))
          print ('Third split of shape %s, is \n%s\n'%(sp3.shape,sp3))
```

```
Original matrix of shape (3, 7), is
[[5 4 1 2 1 5 2]
 [1 5 5 5 0 5 3]
 [2 0 1 5 4 4 4]]

First split of shape (3, 4), is
[[5 4 1 2]
 [1 5 5 5]
 [2 0 1 5]]

Second split of shape (3, 2), is
[[1 5]
 [0 5]
 [4 4]]

Third split of shape (3, 1), is
```

```
[[2]
 [3]
 [4]]
```

1.3.8 Vertical Matrix Splitting

numpy.**vsplit**(ary, *indices_or_sections*)

vsplit is yet another operation which splits the matrix across the vertical plane based on the ***'rowwise split-index array'***. The ***number of columns*** always remain ***constant*** in each split.

In [134]:
```
mat = np.random.randint(0,6,(3,7))
sp1,sp2 = np.vsplit(mat,[1])
print ('Original matrix of shape %s, is \n%s\n'%(mat.shape,mat))
print ('First split of shape %s, is \n%s\n'%(sp1.shape,sp1))
print ('Second split of shape %s, is \n%s\n'%(sp2.shape,sp2))
```

```
Original matrix of shape (3, 7), is
[[5 2 1 1 0 5 5]
 [5 3 0 1 5 2 1]
 [5 5 4 2 2 1 0]]

First split of shape (1, 7), is
[[5 2 1 1 0 5 5]]

Second split of shape (2, 7), is
[[5 3 0 1 5 2 1]
 [5 5 4 2 2 1 0]]
```

Class Assignment for 1.1 to 1.3

1. Create a matrix of size (2,3) with random binary values
2. Find the sum of all 2x2 blocks (overlapping) for a random integer matrix of size (3,5)

1.4. Understanding - Pass By Reference

Just like python lists, Numpy also exhibits default pass-by-reference behavior
Pass by reference Illustration

In [110]:
```
mat1 = np.random.random((2,3))
pt1 = mat1[0].__array_interface__['data'][0]
print ('Memory Location of mat1[0] is : %s\n'%pt1)
pt2 = mat1[1].__array_interface__['data'][0]
print ('Memory Location of mat1[1] is : %s\n'%pt2)
print ('Difference in Memory Location for 3 elements  is : %s bytes\n'%(pt2-pt1))
print ('Memory jump = 8 bytes')
```

```
Memory Location of mat1[0] is : 2041724279440

Memory Location of mat1[1] is : 2041724279464

Difference in Memory Location for 3 elements  is : 24 bytes

Memory jump = 8 bytes
```

1.4.1 Experiment : Change the value of array by Reference

In [117]:
```
def identity(elem):
    return elem
v = np.array([1,2,3])
v1 = v # Copy by reference
v1[0]=10
print ('Value of v, after v1 is modified is %s\n'%v)
# Pass by Function
v = np.array([1,2,3])
v1 = identity(v) # Copy by reference
v1[0]=10
print ('Value of v after func(v) is modified is %s\n'%v)
v = np.array([1,2,3])
v1 = v.copy() # Copy by reference
v1[0]=10
print ('Value of v after v.copy() is modified is %s\n'%v)
```

```
Value of v, after v1 is modified is [10  2  3]
Value of v after func(v) is modified is [10  2  3]

Value of v after v.copy() is modified is [1 2 3]
```

1.4.2 Experiment : Change the value of matrices by Reference

In [136]:
```
mat = np.array([[1,2,3],[4,5,6],[7,8,9]])
print ('Original Matrix is \n%s\n'%mat)

#1 Add 10 to all the even indexes in matrix
mat = np.array([[1,2,3],[4,5,6],[7,8,9]])
mat1 = mat[::2,::2]
mat1+=10
print (' Matrix with 10 added to even indices is \n%s\n'%mat1)
# Changes in mat1 are reflected in mat. This happens because the default assignment in numpy is 'pass by r
eference'
```

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

 Matrix with 10 added to even indices is
[[11 13]
 [17 19]]
```

> **Note :** mat1+=10 is different from mat1=mat1+10, as a new space is allocated for mat1 in the latter case.

1.5 Broadcast Operation in Numpy

Broadcast Operation is like a razor-sharp knife, to be used with great care. It is a way to ***broadcast*** data of lower or same dimension onto another ndarray. It is similar to ***map*** operation in python, scala, hadoop. Broadcast rules are not limited to 2D array. There are extremely specific set of rules for nd-array broadcasting. You can visit ***numpy documentation on broadcasting (https://docs.scipy.org/doc/numpy-1.13.0/user/basics.broadcasting.html)*** to get the complete picture. However, we will be restricting the discussion to only 2D-matrices.

Visualization of broadcast Operation for Matrix

In [147]:
```
mat = np.random.randint(0,6,(3,5))
print ('mat is \n%s\n'%mat)
v = np.array([10,20,30,40,50])
print ('v is \n%s\n'%v)
print ('mat+1 is \n%s\n'%(mat+1)) # Similar behaviour in case of np.array([1])
print ('mat+v is \n%s\n'%(mat+v))
print ('mat*v is \n%s\n'%(mat*v))
print ('mat+mat is \n%s\n'%(mat+mat))
print ('mat*mat is \n%s\n'%(mat*mat))
```

```
mat is
[[1 1 2 1 4]
 [4 2 1 3 5]
 [3 1 1 2 5]]

v is
[10 20 30 40 50]

mat+1 is
[[2 2 3 2 5]
 [5 3 2 4 6]
 [4 2 2 3 6]]

mat+v is
[[11 21 32 41 54]
 [14 22 31 43 55]
 [13 21 31 42 55]]

mat*v is
[[ 10  20  60  40 200]
 [ 40  40  30 120 250]
 [ 30  20  30  80 250]]

mat+mat is
[[ 2  2  4  2  8]
 [ 8  4  2  6 10]
 [ 6  2  2  4 10]]

mat*mat is
```

```
[[ 1  1  4  1 16]
 [16  4  1  9 25]
 [ 9  1  1  4 25]]
```

1.6 Matrix Operations

There are two major matix Operations which are important for us.

1. Inner Product
   - Vector Vector
   - Matrix Vector
   - Matrix Matrix
2. Outer product

1.6.1 Inner Product

In terms of Inner Product or Dot product, <v,w> will be be nothing but sum of element wise product.

$$<v,\ w> = \sum v_i * w_i = \begin{bmatrix} v_0 & v_1 & \cdots & v_n \end{bmatrix}\begin{bmatrix} w_0 w_1 \cdots w_n \end{bmatrix}$$

So, the dot product between vector $v = [1,\ 2,\ 3]$ and $w = [2,\ 4,\ 6]$ will be

$$\begin{aligned} 1*2+2*4+3*6 \quad (1) \\ 2+8+18 \\ 28 \end{aligned}$$

We will be covering matrix matrix multiplication in much depth in next-to-next chapter, but you can take it for granted as of now.

1.6.2 Outer Product

The **outer product** on other end is

$$\begin{bmatrix} v_0 \\ v_1 \\ \cdots \\ v_n \end{bmatrix}\begin{bmatrix} w_0 & w_1 & \cdots & w_n \end{bmatrix} = \begin{bmatrix} v_0 w_0 & \cdots & v_0 w_j & \cdots & v_0 w_n \vdots & \vdots & \vdots v_i w_0 & \cdots & v_i w_j & \cdots & v_i w_n \vdots & \vdots & \vdots v_n w_0 & \cdots & v_n w_j & \cdots & v_n w_n \end{bmatrix}$$

```
In [11]: mat = np.arange(9).reshape(3,3)
         vec1 = np.array([1,2,3])
         vec2 = np.array([2,4,6])
```

```python
vec2 = np.array([2,4,6])
print ('Vector v1 is %s\n'%vec1)
print ('Vector v2 is %s\n'%vec2)
print ('Vector Vector dot product is <v1,v2> is %s\n'%np.dot(vec1,vec2))
print ('---------------\n')

print ('Matrix is\n%s\n'%mat)
print ('Vector v2 is\n%s\n'%vec1)

print ('Matrix Vector product is <mat,vec1> is \n%s\n'%np.dot(mat,vec1))
print ('Matrix Matrix multiplication is <mat,mat> is \n%s\n'%np.dot(mat,mat))
print ('---------------\n')

print ('Vector v1 is %s\n'%vec1)
print ('Vector v2 is %s\n'%vec2)
print ('Vector Vector outer product is <v1,v2> is \n%s\n'%np.outer(vec1,vec2))
```

```
Vector v1 is [1 2 3]

Vector v2 is [2 4 6]

Vector Vector dot product is <v1,v2> is 28

---------------

Matrix is
[[0 1 2]
 [3 4 5]
 [6 7 8]]

Vector v2 is
[1 2 3]

Matrix Vector product is <mat,vec1> is
[ 8 26 44]

Matrix Matrix multiplication is <mat,mat> is
[[ 15  18  21]
 [ 42  54  66]
 [ 69  90 111]]

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

```
Vector v1 is [1 2 3]

Vector v2 is [2 4 6]

Vector Vector outer product is <v1,v2> is
[[ 2  4  6]
 [ 4  8 12]
 [ 6 12 18]]
```

1.7 Statistical Functions

1. **np.mean**(data,axis=0)
2. **np.var**(data,axis=0)
3. **np.sum**(data,axis=0)
4. **np.max**(data,axis=0)
5. **np.min**(data,axis=0)
6. **np.percentile**(data, percentage,axis=0)

In [40]:

```
mat = np.arange(9).reshape((3,3))
print ('Original matrix is \n%s\n'%mat)
print ('Overall mean of matrix is \n%s\n'%np.mean(mat))
print ('Row mean of matrix is \n%s\n'%np.mean(mat, axis=0))
print ('Column mean of matrix is \n%s\n'%np.mean(mat, axis=1))
print ('---------------------------\n')
print ('Overall varience of matrix is %s\n'%np.var(mat))
print ('Overall sum of matrix is %s\n'%np.sum(mat))
print ('Overall min of matrix is %s\n'%np.min(mat))
print ('Overall max of matrix is %s\n'%np.max(mat))

print ('-------------------------------------\n')
marks = np.array([30,31,32,40,90,95,97,98,99,100])
print ('Marks = %s\n'%marks)
print ('Overall 30 percent quantile of marks is %s\n'%(str(np.percentile(marks,30))))
```

```
Original matrix is
[[0 1 2]
 [3 4 5]
 [6 7 8]]
```

```
Overall mean of matrix is
4.0

Row mean of matrix is
[3. 4. 5.]

Column mean of matrix is
[1. 4. 7.]

---------------------------

Overall varience of matrix is 6.666666666666667

Overall sum of matrix is 36

Overall min of matrix is 0

Overall max of matrix is 8

-------------------------------------

Marks = [ 30  31  32  40  90  95  97  98  99 100]

Overall 30 percent quantile of marks is 37.599999999999994
```

1.8 Miscellenous Functions

In this section, we will be discussing some important miscellenous Functions which come in handy for matrix manipulation.

1.8.1 squeeze

numpy.**squeeze**(a,axis=None)
This function tries to reduce the excess dimensions which have only 1 element. If we print the shape of the ndary, these excess dimension will have a 1 in that particular dimension's index. For example, a nd-matrix with shape (1,2,3,1), has 0^{th} and 3^{rd} dimension as excess.

In [5]:
```
ndmat = np.zeros(shape=(1,2,3,1))
print ('Original Shape of ndmat is %s\n'%str(ndmat.shape))
ndmat1 = np.squeeze(ndmat)
```

```
print ('Shape of np.squeeze(ndmat) is %s\n'%str(ndmat1.shape))
ndmat2 = np.squeeze(ndmat,axis=3)
print ('Shape of np.squeeze(ndmat,axis=3) is %s\n'%str(ndmat2.shape))
```

```
Original Shape of ndmat is (1, 2, 3, 1)

Shape of np.squeeze(ndmat) is (2, 3)

Shape of np.squeeze(ndmat,axis=3) is (1, 2, 3)
```

1.8.2 transpose

numpy.**transpose**(a, *axes=None*)
This function reverses the axes for 2D array.
For multidimensional array, it permutes the matrix according to the axis argument

In [21]:
```
mat = np.zeros(shape=(2,3))
print ('Original Shape of 2D matrix is %s\n'%str(mat.shape))
mat_transpose = np.transpose(mat)
print ('Shape of np.transpose(mat) is %s\n'%str(mat_transpose.shape))
mat_transpose[1,0] = 1
print ('np.transpose is by yet again assignment by reference, as changes in transpose reflects in the orig
inal matrix\n')
print ('---------------------------------\n')
# NDimensional matrix
ndmat = np.zeros(shape=(2,3,4))
print ('Original Shape of nDimensional matrix is %s\n'%str(ndmat.shape))
ndmat_transpose = np.transpose(ndmat)
print ('Shape of np.transpose(ndmat) is %s\n'%str(ndmat_transpose.shape))
ndmat_special_transpose = np.transpose(ndmat, axes = [0,2,1])
print ('Shape of np.transpose(ndmat, axes=[0,2,1]) is %s\n'%str(ndmat_special_transpose.shape))
```

```
Original Shape of 2D matrix is (2, 3)

Shape of np.transpose(mat) is (3, 2)

np.transpose is by yet again assignment by reference, as changes in transpose reflects in the original mat
rix

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

```
Original Shape of nDimensional matrix is (2, 3, 4)

Shape of np.transpose(ndmat) is (4, 3, 2)

Shape of np.transpose(ndmat, axes=[0,2,1]) is (2, 4, 3)
```

```
In [ ]:
```

Chapter 1 Assignment

1. Create a null array of size 10 but the fifth value which is 1
2. Reverse a above created array (first element becomes last)
3. Create a 3x3 matrix with values ranging from 0 to 8
4. Find indices of non-zero elements from [1,2,0,0,4,0]
5. Create a 3x3x3 array with random values
6. Create a 10x10 array with random values and find the minimum and maximum values
7. Create a random vector of size 30 and find the mean value