Numpy Notes

import numpy as np

1. Array Size

e.g.

[[1, 2]]: (1,2)

[1, 2]: (2, )

1. Array math
   1. +-\*/

x **=** np.array([[1,2],[3,4]], dtype**=**np.float64)

y **=** np.array([[5,6],[7,8]], dtype**=**np.float64)

*# Elementwise sum; both produce the array*

*# [[ 6.0 8.0]*

*# [10.0 12.0]]*

**print**(x **+** y)

**print**(np.add(x, y))

*# Elementwise difference; both produce the array*

*# [[-4.0 -4.0]*

*# [-4.0 -4.0]]*

**print**(x **-** y)

**print**(np.subtract(x, y))

*# Elementwise product; both produce the array*

*# [[ 5.0 12.0]*

*# [21.0 32.0]]*

**print**(x **\*** y)

**print**(np.multiply(x, y))

*# Elementwise division; both produce the array*

*# [[ 0.2 0.33333333]*

*# [ 0.42857143 0.5 ]]*

**print**(x **/** y)

**print**(np.divide(x, y))

*# Elementwise square root; produces the array*

*# [[ 1. 1.41421356]*

*# [ 1.73205081 2. ]]*

**print**(np.sqrt(x))

Note:

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

v **=** np.array([1, 0, 1])

y **=** x **+** v *# Add v to each row of x using broadcasting*

**print**(y) *# Prints "[[ 2 2 4]*

*# [ 5 5 7]*

*# [ 8 8 10]*

*# [11 11 13]]"*

* 1. np.dot (matrix/vector dot product)

x **=** np.array([[1,2],[3,4]])

y **=** np.array([[5,6],[7,8]])

v **=** np.array([9,10])

w **=** np.array([11, 12])

*# Inner product of vectors; both produce 219*

**print**(v.dot(w))

**print**(np.dot(v, w))

*# Matrix / vector product; both produce the rank 1 array [29 67]*

**print**(x.dot(v))

**print**(np.dot(x, v))

*# Matrix / matrix product; both produce the rank 2 array*

*# [[19 22]*

*# [43 50]]*

**print**(x.dot(y))

**print**(np.dot(x, y))

Note: multiplication of vectors / matrixes:

Note: A\*B is different from np.dot(A, B)!

e.g.

arr0 **=** np.array([[1,2],[3,4]])

arr1 **=** np.array([[1,2],[3,4]])

**print(np.dot(arr0, arr1))**

*# [[ 7 10]*

*[15 22]]*

**print(arr0 \* arr1)**

*# [[ 1 4]*

*[ 9 16]]*

P.S.

How np.dot(A, B) works:

時計 が含まれている画像

自動的に生成された説明

* 1. np.sum(arr, aixs=None)

x **=** np.array([[1,2],[3,4]])

**print**(np.sum(x)) *# Compute sum of all elements; prints "10"*

**print**(np.sum(x, axis**=**0)) *# Compute sum of each column; prints "[4 6]"*

**print**(np.sum(x, axis**=**1)) *# Compute sum of each row; prints "[3 7]"*

* 1. Transpose, note:

x **=** np.array([[1,2], [3,4]])

**print**(x) *# Prints "[[1 2]*

*# [3 4]]"*

**print**(x.T) *# Prints "[[1 3]*

*# [2 4]]*

* np.transpose(arr, axes=None)

Returns an array with axes transposed.

If axes are not specified, reverse the order of the axes at default. Otherwise reshape the array with given axes.

arr = np.arange(24).reshape(2, 3, 4)

print(arr) *# of shape (2,3,4)*

*# [[[ 0 1 2 3]*

*[ 4 5 6 7]*

*[ 8 9 10 11]]*

*[[12 13 14 15]*

*[16 17 18 19]*

*[20 21 22 23]]]*

print(np.transpose(arr)) # of shape (4,3,2)

*# [[[ 0 12]*

*[ 4 16]*

*[ 8 20]]*

*[[ 1 13]*

*[ 5 17]*

*[ 9 21]]*

*[[ 2 14]*

*[ 6 18]*

*[10 22]]*

*[[ 3 15]*

*[ 7 19]*

*[11 23]]]*

print(np.transpose(arr, (0, 2, 1))) *# of shape (2,4,3)*

*# [[[ 0 4 8]*

*[ 1 5 9]*

*[ 2 6 10]*

*[ 3 7 11]]*

*[[12 16 20]*

*[13 17 21]*

*[14 18 22]*

*[15 19 23]]]*

1. Functions
   1. To create an array:
      1. np.zeros((D,H))

Generate an array of size (D, H), with elements 0.

Where D: # rows, H: # columns.

or

np.zeros((H, )): generate an array of size (H, ), with elements 0.

a **=** np.zeros((2,2)) *# Create an array of all zeros*

**print**(a) *# Prints "[[ 0. 0.]*

*# [ 0. 0.]]"*

hidden\_num = 5

b1 **=** np.zeros((hidden\_num, ))

print(b1)

*# [[0. 0. 0. 0. 0.]]*

* + 1. np.ones((D,H))

Generate an array of size [DxH], with elements 1.

b **=** np.ones((1,2)) *# Create an array of all ones*

**print**(b) *# Prints "[[ 1. 1.]]"*

* + 1. np.full((D,H), K)

Generate an array of size [DxH], with elements K.

c **=** np.full((2,2), 7) *# Create a constant array*

**print**(c) *# Prints "[[ 7. 7.]*

*# [ 7. 7.]]"*

* + 1. np.eye(D)

Generate an array of size [DxD], with elements 1 on the diagonal.

d **=** np.eye(2) *# Create a 2x2 identity matrix*

**print**(d) *# Prints "[[ 1. 0.]*

*# [ 0. 1.]]"*

* + 1. np.random.random((D,H))

Generate an array of size [DxH]m with random values of elements at the half-open range of [0, 1).

e **=** np.random.random((2,3)) *# Create an array filled with*

*random values*

print(e)

*# Might print*

*[[0.71501308 0.19170256 0.821027 ]*

*[0.53898364 0.66114444 0.69479722]]*

* + 1. np.random.randn(D,H)

Generate an array of size [DxH]m with random values of mean 0 and standard deviation 1.

e **=** np.random.randn(2,3) *# Create an array filled with*

*random values*

print(e)

*# Might print*

*[[1.37076635 -0.98476718 -0.03825234]*

*[0.75875222 -0.51036498 -2.14860734]]*

* + 1. np.random.normal(mean, std, (D, H))

Generate an array of size (D, H) whose mean is mean and standard deviation is std.

print(np.random.normal(0, 1, (2, 3)))

*# Might print*

*[[ 0.29897801, -1.54360983, 0.99809224]*

*[ 1.14901999, -1.37672078, 0.01036382]]*

* 1. np.flatnonzero()

Return 1D array whose elements are the index of non-zero elements of the original array.

x = np.array([[0, 1, 2],

[3, 0, 0],

[4, 0, 5]])

*# Prints ‘[1 2 3 6 8]’’*

Print(np.flatnonzero(x))

* 1. np.random.choice(a, size=None, replace=True, p=None)

np.random.choice(5, 3, replace=False)

*# array([3,1,0]), note it’s random*

*# Equivalent to np.random.permutation(np.arange(5))[:3].*

* 1. np.reshape(array, (D,H), order = ‘C’)

Reshape the original array to a new array of size [DxH].

v **=** np.array([[0, 1], [2, 3], [4, 5]]) *# v has shape (3, 2)*

**print**(np.reshape(v, (2, 3)))

*# [[0, 1, 2],*

*[3, 4, 5]]*

**print**(np.reshape(v, (2, 3), order = ‘C’))

*# [[0, 1, 2],*

*[3, 4, 5]]*

**print**(np.reshape(v, (2, 3), order = ‘F’))

*# [[0, 4, 3],*

*[2, 1, 5]]*

* 1. np.linspace(start, stop, num)

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

(等差数列生成)

array = np.linspace(2.0, 3.0, num=5)

*# print [2, 2.25, 2.5, 2.75, 3]*

print(array)

* 1. np.c\_(array1, array2)

array = np.c\_[np.array([1,2,3]), np.array([4,5,6])]

*# print [[1, 4],*

*[2, 5],*

*[3, 6]]*

print(array)

* 1. np.linalg.norm(matrix1 - matrix2)

get the L2 distance of two matrices of the same size

a=np.array([1,2])

b=np.array([2,3])

distance=np.linalg.norm(b-a)

*# print 1.414…….*

print(distance)

* 1. np.argsort(arr)

returns an array of indices that index data along the given axis in sorted order… (confusing explanation. Just look at the sample below.)

a = np.array([20, 3, 100])

arg = np.argsort(a)

*# print [1 0 2]*

print(arg)

*# print 3, the smallest element in array*

print(a[arg[0]])

* 1. np.bincount(arr)

return an array of length np.max(arr)+1, with numbers of occurrences of values from 0 to np.max(arr).

(confusing, look at the sample below)

a = np.array([3, 1, 1, 2, 2, 2, 3, 0])

print(np.bincount(a))

*# print [1 2 3 2]*

b = np.array([1, 3, 3, 5, 5, 5])

print(np.bincount(b))

*# print [0 1 0 2 0 3], # frequencies of values in [range(0,5)]*

* 1. np.argmax(arr)

return the index of the greatest element of an array.

a = np.array([3, 1, 1, 2, 2, 2, 3, 0])

b = np.argmax(a)

print(b)

*# 0*

c = np.argmax(np.bincount(a))

print(c)

*# print 2. The greatest value in array [1,2,3,2] indexes 2.*

* 1. np.sum(arr, axis = None)

For array x of size [NxD, like

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

カレンダー が含まれている画像

自動的に生成された説明

1. np.sum(x, axis = None) **(by default)**

adding up all the elements of x, return a certain value

s = np.sum(x, axis = None)

print(s)

*# 78*

1. np.sum(x, axis = 0)

adding up elements by each column of x, return an array of size [1xD]

s = np.sum(x, axis = 0)

print(s)

*# [15, 18, 21, 24]*

アプリケーション が含まれている画像

自動的に生成された説明 グラフィカル ユーザー インターフェイス

自動的に生成された説明

1. np.sum(x, axis = 1)

adding up elements by each row of x, return an array of size [1xN]

s = np.sum(x, axis = 1)

print(s)

*# [10, 26, 42]*

カレンダー

自動的に生成された説明グラフィカル ユーザー インターフェイス

中程度の精度で自動的に生成された説明

* 1. np.array\_split(arr, n\_folds)

splits arr into a list of length n\_folds

x = np.arange(9)

print(np.array\_split(x, 4))

*# [array([0, 1, 2]), array([3, 4]), array([5, 6]), array([7, 8])]s*

* 1. np.concatenate(arrays\_list)

to merge all arrays in given arrays\_list.

Useful for cross-validation, like generating training batch data (splitting original training data into num\_folds of folds, using all but one of the folds as training data and the rest only one as validation data. In the end averaging the accuracies across different folds to evaluate performance) .

x = np.arange(9)

x\_folds = np.array\_split(x, 4)

*# for cross-validation of folds at 5*

num\_folds = 5

for fold in range(num\_folds):

x\_train\_batch = np.concatenate(x\_folds[:fold] + x\_folds[fold+1:])

x\_validate\_batch = x\_folds[fold]

* 1. np.vstack((arr0, arr1,…))

return an array vertically concatenating given arrays of the same # columns

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

b = np.array([[5, 6], [7, 8]])

c = np.array([[9, 10], [11, 12]])

print(np.hstack((a, b)))

*# [[1, 2]*

*[3, 4]*

*[5, 6]*

*[7, 8]]*

print(np.hstack((a, b,c)))

*# [[1, 2]*

*[3, 4]*

*[5, 6]*

*[7, 8]*

*[9, 10]*

*[11, 12]]*

* 1. np.hstack((arr0, arr1,…)) or np.hstack([arr0, arr1,…])

return an array horizontally concatenating given arrays of the same # rows

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

b = np.array([[5, 6], [7, 8]])

c = np.array([[9, 10]])

print(np.hstack((a, b)))

*# [[1, 2, 5, 6]*

*[5, 6, 7, 8]]*

print(np.hstack((a,c)))

*# return error because a and c have different # rows*

* 1. np.mean(arr, axis = None)

1. np.mean(arr) *or* np.mean(arr, axis = None)

return the mean of all elements (by each row and column)

1. np.mean(arr, axis = 0)

return the mean of elements by each column (vertically)

1. np.mean(arr, axis = 1)

return the mean of elements by each row (horizontally)

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

print(np.mean(a))

*# 2.5*

print(np.mean(a, axis=0))

*# [2., 3.]*

print(np.mean(a, axis=1))

*# [1.5, 3.5]*

* 1. np.max(array, …)

arr = np.array([1, -2, 3])

np.max(arr, 0)

*# 3*

* 1. np.maximum(array, …)

arr = np.array([1, -2, 3])

np. maximun(arr, 0)

*# [1, 0, 3]*

Note:

np.max(arr) returns a single value.

np.maximum(arr) returns an array of the same size as arr.

* 1. np.add.at(arr, index\_to\_add, number\_to\_add)

Add number\_to\_add at the position of index\_to\_add of arr.

It changes the original arr.

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

np.add.at(a, [0, 1], 1) *# add 1 to the indices 0 and 1 of array a.*

print(a)

*# array([2, 3, 3, 4])*

* 1. np.prod(arr, axis = None)

return the product of array elements over a given axis.

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

print(np.prod(a))

*# 24 (1\*2\*3\*4)*

print(np.prod(a, axis=0))

*# [3,8] (1\*3, 2\*4)*

print(np.prod(a, axis=1))

*# [2, 12] (1\*2, 3\*4)*

* 1. np.atleast2d(arr)

Convert a one-dimensional array into a two-dimensional array.

It adds an additional dimension to the array.

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

arr\_2d = np.atleast2d(arr)

print(arr\_2d)

*# [[1 2 3 4 5 6]]*

Given two datasets, a training data set X\_train of size [NxD] and a test data set X\_test of size [HxD].

Each data point from either data set is of size [1xD].

The distance matrix between X\_test and X\_train is diff, of size [HxN].

Now, use numpy to calculate the Euclidean distance (L2 distance):

Recall

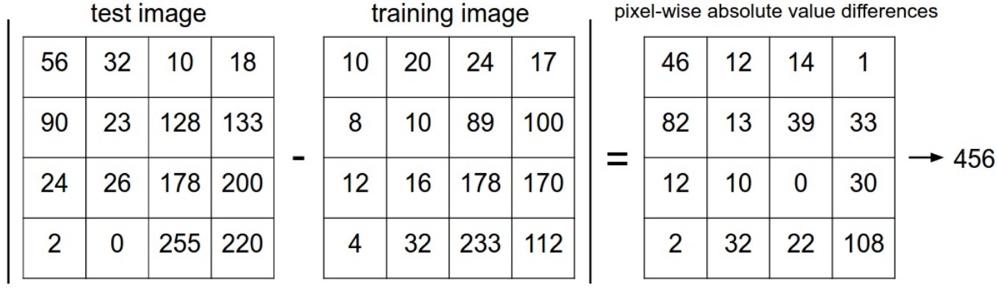
1. between the i-th test data point and the j-th training data point, stored in dists[i][j]:

dists[i][j] = np.sqrt(np.sum(np.square(X\_test[i] - X\_train[j])))

Or

dists[i][j] = np.linalg.norm(X\_test[i] - X\_train[j])

An illustration:



1. between the i-th test data point and all the training data points, stored in dists[i][:]:

dists[i][:] = np.sqrt(np.sum(np.square(X\_test[i] – X\_train), axis = 1))

note: **axis = 1** is essential to add up elements by each row (horizontal).

Or

dists[i][:] = np.linalg.norm(X\_test[i] – X\_train)

1. between all the test data points and all the training data points, stored in dists[:][:]:

test\_sum = np.sum(np.square(X),axis = 1, keepdims = True)

train\_sum = np.sum(np.square(self.X\_train),axis = 1)

cross\_sum = np.dot(X, self.X\_train.T)

dists = np.sqrt(train\_sum + test\_sum - 2\*cross\_sum)

*# note test\_sum also can be:*

test\_sum = np.transpose([np.sum(np.square(X), axis = 1)]))

*# and cross\_num also can be:*

cross\_sum = X @ self.X\_train.T

*# for the deviation of the formular, refer to* [Implementing Euclidean Distance Matrix Calculations From Scratch In Python](https://www.dabblingbadger.com/blog/2020/2/27/implementing-euclidean-distance-matrix-calculations-from-scratch-in-python) or [cs231n assignment1 | RUOCHI.AI](https://zhangruochi.com/cs231n-assignment1/2019/10/01/)

or

dists = np.linalg.norm(X\_test – X\_train)

Usually we generate dists without loops, i.e. just do it at the level of all training data and all test data.