Theory: Submodule linalg

14 minutes 0 / 5 problems solved

In previous topics, we have discussed a great number of operations with NumPy arrays and matrices, and there are many to come. For instance, sometimes you need to find a matrix rank or a matrix determinant, to decompose a matrix, etc. In these situations, you can use the **linalg** (*linear algebra*) submodule; it offers various methods to apply linear algebra on NumPy arrays. In this topic, we are going to cover the important functions of this module. The *linalg* submodule is a part of NumPy, so just import the library to start using the submodule.

§1. Matrix Determinant

Sometimes you would need to compute the matrix determinant. In this case, you can use the `np.linalg.det()` function and just pass your NumPy array to it. In the example below, the function accepts a two-dimensional array.

```
arr = np.array([[1, 3], [2, 4]])
print(np.linalg.det(arr))  # -2.0
```

Note that the function accepts the square matrix. If the matrix is not square, it will produce an error.

This function can also compute determinants for several matrices of the same shape. Mind the next example.

```
arr = np.array([[[1, 2], [3, 2]],
                [[4, 7], [1, 8]]])
print(np.linalg.det(arr))  # [-4. 25.]
```

In this example, the array contains several subarrays of the same shape: `[[1, 2], [3, 2]]` and `[[4, 7], [1, 8]]`. If the shapes of the subarrays were different, there would be an error. The determinants are computed separately for each array. The result is the NumPy array of the obtained determinants.

§2. Invertible Matrix

In this section, we are going to discuss how to deal with **an inverse of a matrix**. In linalg, the `np.linalg.inv()` function allows us to obtain the inverse of the matrix. In the example below, we create a square matrix using `np.array()` and then pass it to `np.linalg.inv()` for computing the inverse of the matrix.

```
arr1 = np.array([[2, 1, 3], [1, 2, 4], [2, 4, 6]])
arr2 = np.linalg.inv(arr1)
print(arr2)
# [[ 0.66666667 -1.          0.33333333]
#  [-0.33333333 -1.          0.83333333]
#  [ 0.          1.         -0.5       ]]
```

To check if the obtained matrix is really inverse, we can use the `np.dot()` function: if the result is an identity matrix, it means that `arr2` is *an inverse of the matrix*.

```
print(np.dot(arr1, arr2))
# [[1. 0. 0.]
#  [0. 1. 0.]
#  [0. 0. 1.]]
```

There is one more useful thing to mention. Have a look at the following snippet:

Current topic:

Submodule linalg

Topic depends on:

- ✕ Linear dependence and independence
- ✕ Operations with several arrays
- ✕ Eigenvalues and eigenvectors
- ✕ An inverse operator

Table of contents:

```
arr = np.array([[-1, 1.5], [2/3, -1]])
print(np.linalg.inv(arr))
# Traceback (most recent call last):
# ...
# numpy.linalg.LinAlgError: Singular matrix
```

You can see the error. What happened? The matrix that we have passed to the function, has a determinant of 0. In this case, our matrix is not invertible. It is also known as **a singular matrix**. Remember: if you try to pass a ***singular matrix*** to the function in your program, the `LinAlgError` will be raised.

§3. Array Eigenvalues

In linear algebra, you may come across **eigenvalues** and **eigenvectors**. NumPy also allows computing ***eigenvalues*** and ***eigenvectors*** of a square matrix with a help of the `np.linalg.eig()` function.

```
arr = np.array([[1, 2], [3, 4]])
w, v = np.linalg.eig(arr)

print(np.linalg.eig(arr))
# (array([-0.37228132,  5.37228132]), array([[-0.82456484, -0.41597356],
#        [ 0.56576746, -0.90937671]]))

print(w)
# [-0.37228132  5.37228132]



print(v)

# [[-0.82456484 -0.41597356]

#  [ 0.56576746 -0.90937671]]
```

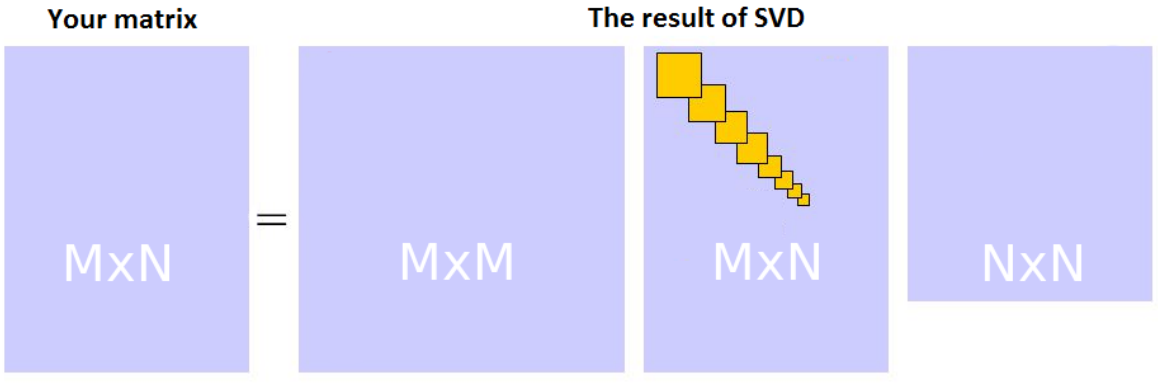
There are a few things to mention. The `eig()` function returns a tuple that consists of two arrays. The first `w` array contains the eigenvalues. The second `v` array contains a matrix, columns of which form eigenvectors so that the column `v[:,i]` is the eigenvector corresponding to the `w[i]` eigenvalue. To ensure that the values are correct, let's check the equality of dot products: for instance, the dot product of the original array and the first eigenvector must be equal to the dot product of the first eigenvalue and the first eigenvector.

```
print(np.dot(arr, v[:, 0]))  # [ 0.30697009 -0.21062466]
print(np.dot(w[0], v[:, 0]))  # [ 0.30697009 -0.21062466]
```

As you can see, the eigenvalues and eigenvectors are correct.

§4. Singular Value Decomposition

In NumPy's linalg, there is a possibility to implement **singular value decomposition** (also known as SVD). We'll explain it briefly below. Let's have a look at the picture:



In linear algebra, the SVD is a procedure of representing your matrix as a dot product of three matrices. The first matrix consists of columns that are known as left singular vectors. The second matrix is a diagonal one. Each element on its diagonal is a singular value of the original matrix. The last

matrix is a matrix with right-singular vectors. The singular values and the vectors are obtained with the help of the eigenvalues and the eigenvectors. The SVD is a very useful method, especially for machine learning: if you have a huge array of data, it allows you to compress it more efficiently.

Right now, we are interested in practical programming rather than in theoretical approaches. The `np.linalg.svd()` function in NumPy provides the means for the SVD:

```
arr = np.array([[1, 2, 3], [4, 5, 6], [0, 3, 2], [7, 2, 5]])
svd_matr = np.linalg.svd(arr)
print(svd_matr)
# (array([[-0.2733616 , -0.27938277,  0.86441813, -0.31622777],
#        [-0.67416854, -0.36789866, -0.10073405,  0.63245553],
#        [-0.21105836, -0.56855365, -0.48187252, -0.63245553],
#        [-0.65285878,  0.68069273, -0.1021412 , -0.31622777]]),
#  array([12.81712466,  4.14874014,  0.71363215]),
#  array([[-0.58827915, -0.45692452, -0.66719384],

#        [ 0.72645468, -0.66105233, -0.18781218],

#        [-0.35523405, -0.59517208,  0.72081826]]))
```

The result of SVD is a tuple. The first array in the tuple is the array with the left singular vectors. The second array contains three singular values. The third array has the right singular vectors.

If it is impossible to decompose the given array, the `LinAlgError` will be raised.

§5. Matrix Rank

The **matrix rank** is the maximum number of **linearly independent** rows or columns. The rows and columns are *linearly independent* if no column or row of a matrix can be defined as a linear combination of other columns or rows. Imagine, we have the following matrix.

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{pmatrix}$$

This matrix has *2 rows* and *4 columns*. In linear algebra, the rank of a matrix doesn't exceed the minimum number of rows/columns. In this case, it will not exceed *2*. There are a lot of ways for calculating ranks in linear algebra, but we will not focus on them right now.

Let's look at how we could compute the matrix rank in NumPy. The `np.linalg.matrix_rank()` function can help with it:

```
arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])
rank = np.linalg.matrix_rank(arr)
print(rank)  # 2
```

The function can also compute ranks for several matrices of the same shape:

```
arr = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
rank = np.linalg.matrix_rank(arr)
print(rank)  # [2 2]
```

The result is a NumPy array with ranks for each submatrix: `[[1, 2], [3, 4]]` and `[[5, 6], [7, 8]]`.

§6. Conclusion

So far, we have had a quick look at the *linalg* submodule in NumPy. Now you know:

- how to compute a square array determinant with the help of `np.linalg.det()`;
- how to find an inverse of a matrix using `np.linalg.inv()`;
- how to use `np.linalg.eig()` for computing eigenvectors and eigenvalues;
- how to get the SVD of a matrix with the help of `np.linalg.svd()`;
- how to find the matrix rank using `np.linalg.matrix_rank()`.

It is impossible to cover all functions that exist in this submodule. If you are eager to find out more information, read the Linear Algebra section of the NumPy documentation.

Report a typo

10 users liked this theory. **0** didn't like it. **What about you?**

Start practicing

Comments (0) Hints (0) Useful links (0) Show discussion