ELEG 5491: Introduction to Deep Learning

NumPy for Python

Prof. LI Hongsheng

Office: SHB 428
e-mail: hsli@ee.cuhk.edu.hk
web: https://blackboard.cuhk.edu.hk

Department of Electronic Engineering
The Chinese University of Hong Kong

Feb. 2023

NumPy for Python

- Short for “Numerical Python”
- Open source add-on modules to Python
- It provides common mathematical and numerical routines in pre-compiled, fast functions.
- Similar functions to those in MATLAB
- Many mathematical Python software and packages are built based on NumPy

- Documentation: `https://docs.scipy.org/doc/`

- **IMPORTANT:** Learn to use Google. At most times, it will bring you to some answers on Stack Overflow

Importing

- Before using NumPy, one should import NumPy into the program
- It is common to import under the brief name `np`

```
>>> import numpy as np
```

- One should access NumPy objects using `np.X`

Array creation & accessing

- The central feature of NumPy is the array object class
- Arrays are similar to lists in Python, except that every element of an array must be of the same type, typically a numeric type like `float` or `int`
- Arrays make operations with large amounts of numeric data very fast and are generally much more efficient than lists
- An array can be created from a list

```
>>> a = np.array([1, 4, 5, 8], float)
>>> a
array([ 1.,  4.,  5.,  8.])
>>> type(a)
<type 'numpy.ndarray'>
```

- Array elements are accessed, sliced, and manipulated just like lists

```
>>> a[:2]
array([ 1.,  4.])
>>> a[3]
8.0
>>> a[0] = 5.
>>> a
array([ 5.,  4.,  5.,  8.])
```

- NumPy array is **mutable**

Multidimensinoal array

- Unlike lists, different axes of a multidimensional array are accessed using commas inside bracket notation

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]], float)
>>> a
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]])
>>> a[0,0]
1.0
>>> a[0,1]
2.0
```

- Array slicing works with multiple dimensions in the same way as usual

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]], float)
>>> a[1,:]
array([ 4.,  5.,  6.])
>>> a[:,2]
array([ 3.,  6.])
>>> a[-1:,-2:]
array([[ 5.,  6.]])
```

- The shape property of an array returns a tuple with the size of each array dimension

```
>>> a.shape
(2, 3)
```

- The dtype property tells you what type of values are stored by the array

```
>>> a.dtype
dtype('float64')
```

Multidimensinoal array

- When used with an array, the `len` function returns the length of the first axis

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]], float)
>>> len(a)
2
```

- The `in` statement can be used to test if values are present in an array

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]], float)
>>> 2 in a
True
>>> 0 in a
False
```

- Arrays can be reshaped to create **a new array**

```
>>> a = np.array(range(10), float)
>>> a
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.])
>>> a = a.reshape((5, 2))
>>> a
array([[ 0.,  1.],
       [ 2.,  3.],
       [ 4.,  5.],
       [ 6.,  7.],
       [ 8.,  9.]])
>>> a.shape
(5, 2)
```

Multidimensinoal array

- The copy function can be used to create a new, separate copy of an array in memory if needed

```
>>> a = np.array([1, 2, 3], float)
>>> b = a
>>> c = a.copy()
>>> a[0] = 0
>>> a
array([0., 2., 3.])
>>> b
array([0., 2., 3.])
>>> c
array([1., 2., 3.])
```

- Lists can also be created from arrays:

```
>>> a = np.array([1, 2, 3], float)
>>> a.tolist()
[1.0, 2.0, 3.0]
>>> list(a)
[1.0, 2.0, 3.0]
```

- Transposed versions of arrays (new arrays) can also be generated

```
>>> a = np.array(range(6), float).reshape((2, 3))
>>> a
array([[ 0.,  1.,  2.],
       [ 3.,  4.,  5.]])
>>> a.transpose()
array([[ 0.,  3.],
       [ 1.,  4.],
       [ 2.,  5.]])
```

Multidimensinoal array

- One-dimensional versions of multi-dimensional arrays can be generated with `flatten`

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]], float)
>>> a
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]])
>>> a.flatten()
array([ 1.,  2.,  3.,  4.,  5.,  6.])
```

- Two or more arrays can be concatenated together using `concatenate`

```
>>> a = np.array([1,2], float)
>>> b = np.array([3,4,5,6], float)
>>> c = np.array([7,8,9], float)
>>> np.concatenate((a, b, c))
array([1., 2., 3., 4., 5., 6., 7., 8., 9.])
```

Multidimensinoal array

- If an array has more than one dimension, it is possible to specify the axis along which multiple arrays are concatenated. By default (without specifying the axis), NumPy concatenates along the first dimension

```
>>> a = np.array([[1, 2], [3, 4]], float)
>>> b = np.array([[5, 6], [7,8]], float)
>>> np.concatenate((a,b))
array([[ 1.,  2.],
       [ 3.,  4.],
       [ 5.,  6.],
       [ 7.,  8.]])
>>> np.concatenate((a,b), axis=0)
array([[ 1.,  2.],
       [ 3.,  4.],
       [ 5.,  6.],
       [ 7.,  8.]])
>>> np.concatenate((a,b), axis=1)
array([[ 1.,  2.,  5.,  6.],
       [ 3.,  4.,  7.,  8.]])
```

Multidimensinoal array

- The dimensionality of an array can be increased using the `newaxis` constant in bracket notation:

```
>>> a = np.array([1, 2, 3], float)
>>> a
array([1., 2., 3.])
>>> a[:,np.newaxis]
array([[ 1.],
       [ 2.],
       [ 3.]])
>>> a[:,np.newaxis].shape
(3,1)
>>> b[np.newaxis,:]
array([[ 1.,  2.,  3.]])
>>> b[np.newaxis,:].shape
(1,3)
```

Other ways to create arrays

- The arange function is similar to the range function but returns an array

```
>>> np.arange(5, dtype=float)
array([ 0.,   1.,   2.,   3.,   4.])
>>> np.arange(1, 6, 2, dtype=int)
array([1, 3, 5])
```

- The functions zeros and ones create new arrays of specified dimensions filled with these values

```
>>> np.ones((2,3), dtype=float)
array([[ 1.,   1.,   1.],
       [ 1.,   1.,   1.]])
>>> np.zeros(7, dtype=int)
array([0, 0, 0, 0, 0, 0, 0])
```

- The zeros_like and ones_like functions create a new array with the same dimensions and type of an existing one

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]], float)
>>> np.zeros_like(a)
array([[ 0.,   0.,   0.],
       [ 0.,   0.,   0.]])
>>> np.ones_like(a)
array([[ 1.,   1.,   1.],
       [ 1.,   1.,   1.]])
```

- To create an identity matrix of a given size

```
>>> np.identity(4, dtype=float)
```

Array mathematics

- When standard mathematical operations are used with arrays, they are applied on an element-by-element basis

```
>>> a = np.array([1,2,3], float)
>>> b = np.array([5,2,6], float)
>>> a + b
array([6., 4., 9.])
>>> a - b
array([-4., 0., -3.])
>>> a * b
array([5., 4., 18.])
>>> b / a
array([5., 1., 2.])
>>> a % b
array([1., 0., 3.])
>>> b**a
array([5., 4., 216.])
```

- For two-dimensional arrays, multiplication remains elementwise and does not correspond to matrix multiplication

```
>>> a = np.array([[1,2], [3,4]], float)
>>> b = np.array([[2,0], [1,3]], float)
>>> a * b
array([[2., 0.], [3., 12.]])
```

- Errors are thrown if arrays do not match in size

Broadcasting

- However, arrays that do not match in the number of dimensions will be *broadcasted* by Python to perform mathematical operations
- This often means that the smaller array will be repeated as necessary to perform the operation indicated

```
>>> a = np.array([[1, 2], [3, 4], [5, 6]], float)
>>> b = np.array([-1, 3], float)
>>> a
array([[ 1.,  2.],
       [ 3.,  4.],
       [ 5.,  6.]])
>>> b
array([-1.,  3.])
>>> a + b
array([[ 0.,  5.],
       [ 2.,  7.],
       [ 4.,  9.]])
```

- the one-dimensional array b was broadcasted to a two-dimensional array that matched the size of a

```
array([[-1.,  3.],
       [-1.,  3.],
       [-1.,  3.]])
```

Broadcasting

- Python automatically broadcasts arrays in this manner. Sometimes, however, how we should broadcast is ambiguous. In these cases, we can use the `newaxis` constant to specify how we want to broadcast

```
>>> a = np.zeros((2,2), float)
>>> b = np.array([-1., 3.], float)
>>> a
array([[ 0.,  0.],
       [ 0.,  0.]])
>>> b
array([-1., 3.])
>>> a + b
array([[-1.,  3.],
       [-1.,  3.]])
>>> a + b[np.newaxis,:]
array([[-1.,  3.],
       [-1.,  3.]])
>>> a + b[:,np.newaxis]
array([[-1., -1.],
       [ 3.,  3.]])
```

- In addition to the standard operators, NumPy offers a large library of common mathematical functions that can be applied elementwise to arrays. Among these are the functions: `floor`, `ceil`, `rint`, `abs`, `sign`, `sqrt`, `log`, `log10`, `exp`, `sin`, `cos`, `tan`, `arcsin`, `arccos`, `arctan`, `sinh`, `cosh`, `tanh`, `arcsinh`, `arccosh`, and `arctanh`

Basic array operations

- The items in an array can be summed or multiplied

```
>>> a = np.array([2, 4, 3], float)
>>> a.sum()
9.0
>>> a.prod()
24.0
```

- Alternatively, standalone functions in the NumPy module can be accessed

```
>>> np.sum(a)
9.0
>>> np.prod(a)
24.0
```

- A number of routines enable computation of statistical quantities in array datasets, such as the mean (average), variance, and standard deviation

```
>>> a = np.array([2, 1, 9], float)
>>> a.mean()
4.0
>>> a.var()
12.666666666666666
>>> a.std()
3.5590260840104371
```

- One could find the minimum and maximum element values

```
>>> a = np.array([2, 1, 9], float)
>>> a.min()
1.0
>>> a.max()
9.0
```

Basic array operations

- The argmin and argmax functions return the array indices of the minimum and maximum values

```
>>> a = np.array([2, 1, 9], float)
>>> a.argmin()
1
>>> a.argmax()
2
```

- For multidimensional arrays, each of the functions thus far described can take an optional argument axis that will perform an operation along only the specified axis

```
>>> a = np.array([[0, 2], [3, -1], [3, 5]], float)
>>> a.mean(axis=0)
array([ 2.,  2.])
>>> a.mean(axis=1)
array([ 1.,  1.,  4.])
>>> a.min(axis=1)
array([ 0., -1.,  3.])
>>> a.max(axis=0)
array([ 3.,  5.])
```

- Arrays could also be sorted

```
>>> a = np.array([6, 2, 5, -1, 0], float)
>>> sorted(a)
[-1.0, 0.0, 2.0, 5.0, 6.0]
>>> a.sort()
>>> a
array([-1.,  0.,  2.,  5.,  6.])
```

Basic array operations

- Unique elements can be extracted from an array:

```
>>> a = np.array([1, 1, 4, 5, 5, 5, 7], float)
>>> np.unique(a)
array([ 1.,  4.,  5.,  7.])
```

- For two dimensional arrays, the diagonal can be extracted

```
>>> a = np.array([[1, 2], [3, 4]], float)
>>> a.diagonal()
array([ 1.,  4.])
```

Comparison operators and value testing

- Boolean comparisons can be used to compare members elementwise on arrays of equal size
- The return value is an array of Boolean `True / False` values

```
>>> a = np.array([1, 3, 0], float)
>>> b = np.array([0, 3, 2], float)
>>> a > b
array([ True, False, False], dtype=bool)
```

- Arrays can be compared to single values using broadcasting

```
>>> a = np.array([1, 3, 0], float)
>>> a > 2
array([False,  True, False], dtype=bool)
```

- The `any` and `all` operators can be used to determine whether or not any or all elements of a Boolean array are true

```
>>> c = np.array([ True, False, False], bool)
>>> any(c)
True
>>> all(c)
False
```

Comparison operators and value testing

- Compound Boolean expressions can be applied to arrays on an element-by-element basis

```
>>> a = np.array([1, 3, 0], float)
>>> np.logical_and(a > 0, a < 3)
array([ True, False, False], dtype=bool)
>>> b = np.array([True, False, True], bool)
>>> np.logical_not(b)
array([False,  True, False], dtype=bool)
>>> c = np.array([False, True, False], bool)
>>> np.logical_or(b, c)
array([ True,  True,  False], dtype=bool)
```

Array item selection and manipulation

- Boolean arrays can be used as array selectors
- The return value is an array of Boolean `True / False` values

```
>>> a = np.array([[6, 4], [5, 9]], float)
>>> a >= 6
array([[ True, False],
       [False,  True]], dtype=bool)
>>> a[a >= 6]
array([ 6.,  9.])
```

- More complicated selections can be achieved using Boolean expressions

```
>>> a[np.logical_and(a > 5, a < 9)]
>>> array([ 6.])
```

- it is possible to select using integer arrays. The integer arrays contain the indices of the elements to be taken from an array

```
>>> a = np.array([2, 4, 6, 8], float)
>>> b = np.array([0, 0, 1, 3, 2, 1], int)
>>> a[b]
array([ 2.,  2.,  4.,  8.,  6.,  4.])
```

- For multidimensional arrays, multiple one-dimensional integer arrays are sent to the selection bracket, one for each axis

```
>>> a = np.array([[1, 4], [9, 16]], float)
>>> b = np.array([0, 0, 1, 1, 0], int)
>>> c = np.array([0, 1, 1, 1, 1], int)
>>> a[b,c]
array([  1.,   4.,  16.,  16.,   4.])
```

Vector and matrix mathematics

- To perform a dot product

```
>>> a = np.array([1, 2, 3], float)
>>> b = np.array([0, 1, 1], float)
>>> np.dot(a, b)
5.0
```

- The dot function also generalizes to matrix multiplication

```
>>> a = np.array([[0, 1], [2, 3]], float)
>>> b = np.array([2, 3], float)
>>> c = np.array([[1, 1], [4, 0]], float)
>>> a
array([[ 0.,  1.],
       [ 2.,  3.]])
>>> np.dot(b, a)
array([  6.,  11.])
>>> np.dot(a, b)
array([  3.,  13.])
>>> np.dot(a, c)
array([[  4.,   0.],
       [ 14.,   2.]])
>>> np.dot(c, a)
array([[ 2.,  4.],
       [ 0.,  4.]])
```

Vector and matrix mathematics

- NumPy also comes with a number of built-in routines for linear algebra calculations. These can be found in the sub-module `linalg`
- The determinant

```
>>> a = np.array([[4, 2, 0], [9, 3, 7], [1, 2, 1]], float)
>>> a
array([[ 4.,  2.,  0.],
       [ 9.,  3.,  7.],
       [ 1.,  2.,  1.]])
>>> np.linalg.det(a)
-53.999999999999993
```

- The eigenvalues and eigenvectors of a matrix

```
>>> vals, vecs = np.linalg.eig(a)
>>> vals
array([ 9.        ,  2.44948974, -2.44948974])
>>> vecs
array([[-0.3538921 , -0.56786837,  0.27843404],
       [-0.88473024,  0.44024287, -0.89787873],
       [-0.30333608,  0.69549388,  0.34101066]])
```

- The inverse of a matrix

```
>>> b = np.linalg.inv(a)
>>> b
array([[ 0.14814815,  0.07407407, -0.25925926],
       [ 0.2037037 , -0.14814815,  0.51851852],
       [-0.27777778,  0.11111111,  0.11111111]])
>>> np.dot(a, b)
array([[  1.00000000e+00,   5.55111512e-17,   2.22044605e-16],
       [  0.00000000e+00,   1.00000000e+00,   5.55111512e-16],
       [  1.11022302e-16,   0.00000000e+00,   1.00000000e+00]])
```

Vector and matrix mathematics

- Singular value decomposition

```
>>> a = np.array([[1, 3, 4], [5, 2, 3]], float)
>>> U, s, Vh = np.linalg.svd(a)
>>> U
array([[-0.6113829 , -0.79133492],
       [-0.79133492,  0.6113829 ]])
>>> s
array([ 7.46791327,  2.86884495])
```

```
>>> Vh
array([[-0.61169129, -0.45753324, -0.64536587],
       [ 0.78971838, -0.40129005, -0.46401635],
       [-0.046676  , -0.79349205,  0.60678804]])
```

Random numbers

- NumPy's built-in pseudorandom number generator routines in the sub-module `random`
- The random number seed can be set

```
>>> np.random.seed(293423)
```

- The `rand` function can be used to generate two-dimensional random arrays in $[0.0, 1.0)$, or the `resize` function could be employed

```
>>> np.random.rand(2,3)
array([[ 0.50431753,  0.48272463,  0.45811345],
       [ 0.18209476,  0.48631022,  0.49590404]])
>>> np.random.rand(6).reshape((2,3))
array([[ 0.72915152,  0.59423848,  0.25644881],
       [ 0.75965311,  0.52151819,  0.60084796]])
```

- To generate random integers in the range [min, max)

```
>>> np.random.randint(5, 10)
9
```

- To draw from a continuous normal (Gaussian) distribution with mean=1.5 and standard deviation=4.0

```
>>> np.random.normal(1.5, 4.0)
0.83636555041094318
```

Random numbers

- The random module can also be used to randomly shuffle the order of items in a list

```
>>> l = range(10)
>>> l
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> np.random.shuffle(l)
>>> l
[4, 9, 5, 0, 2, 7, 6, 8, 1, 3]
```

- Notice that the shuffle function modifies the list in place, meaning it does not return a new list but rather modifies the original list itself

Learn scikit-learn and PyTorch

- `scikit-learn` and PyTorch are popular Python libraries for conventional machine learning and deep learning techniques

- PyTorch documentation: `https://pytorch.org/docs/stable/index.html`
- `scikit-learn` documentation: `https://scikit-learn.org/stable/`

- There are also nice tutorials and examples from their offical websites:
  - `scikit-learn`: `https://scikit-learn.org/stable/tutorial/index.html`
  - PyTorch: `https://pytorch.org/tutorials/`
  - PyTorch examples: `https://github.com/pytorch/examples`