NumPy入门教程

404 student

零、写在前面

NumPy数组库是python中的一个库，能够帮助创建一个支持多种操作的数组。作为一名ai学生，这个库必然是要牢牢掌握的。据我所知，学习经济金融这一块的朋友们在做数据分析时，也需要用到NumPy，因此也可以参考本教程。

虽然我在程序设计实习这门课已经学过了，但是我上课几乎一个字没听（~~我也不知道我在干什么~~），再加上在ai引论这门课曾经多次使用这个库，每次写lab的时候我都对numpy的各种函数感到迷茫。所以我决定重新从头学习一遍。

本教程参考B站UP主“爆肝杰哥”的视频《Python深度学习：NumPy数组库》，谢谢杰哥（×）。虽然UP主说不准抄袭讲义，但是他应该找不到我（）

一、数组基础

第一个知识点：导入numpy时要先写：`import numpy as np`。（真有人不知道吗）

1. 数据类型

为了节省内存，一个numpy数组只容纳一种数据类型，所以可以把numpy数组简单分为**整数型数组**和**浮点型数组**。但凡数组中有一个元素是浮点数，那么这个数组就是浮点型数组。比如，[1,2,3] 是整数型数组，但 [1.0,2,3] 是浮点型数组，输出为[1. 2. 3.]。注意，输出numpy数组时，元素之间**没有逗号**。

往整数型数组中插入浮点数，浮点数会**变成整数**（向下取整）。同理，整数插入到浮点型数组时，也会**变成浮点数**。

要转换一个numpy数组的类型，需要使用 `.astype()` 方法。例如，要把整数型数组arr1转化成浮点型数组arr2，代码如下：

```
arr2 = arr1.astype(float)
```

当然也不只有这种方法。如果**整数型数组与浮点数做运算、整数型数组遇到除法、整数型数组与浮点型数组做运算**，都会变成浮点数。这一点和整数是一样的。浮点型数组一般不会自动变成整数型数组。

2. 数组维度

在深度学习中，基本用到的是**一维和二维**数组。不同维度的数组，从外形上的本质区别就是有**几层中括号**。

有的函数需要传入数组的**形状参数**。所谓形状参数，就是标明**维数和各维长度**的元组。元组中有几个数字表示有几维，每个数字的大小表明长度。比如，一维数组的形状参数为(x,)，二维(x,y)，三维(x,y,z)，以此类推。对于三维数组，x表示**最大的中括号里有多少个二级中括号**；y表示二级中括号里有多少个三级中括号；z表示最小的中括号里有几个数字。二维的同理。

想知道一个数组的形状，使用 `.shape` 属性。返回的就是形状参数。注意，**属性**是不需要参数的，后面不接括号；**方法**本质上是一个函数，需要接括号，有时括号里还有参数。

转换数组的形状，需要使用 `.reshape()` 方法。这个方法可以实现一维数组与二维数组的互相转换，也可以实现二维数组的长度变化。例如，对数组 `arr1 = [1 2 3]` 使用 `arr2 = arr1.reshape((3,1))` ，就会变成 `[[1][2][3]]` 的三行一列数组。

值得一提的是，可以留一个维数设置成-1，让系统自己算。比如上文可以改成 `reshape((1,-1))` ，仍然可以正常运行。

下文中，一维数组称为**向量**，二维数组称为**矩阵**。

二、数组的创建

1. 创建指定数组

使用 `np.array()` 函数，将一个已知的列表转化为numpy数组。

```
import numpy as np
arr1 = np.array([1,2,3])  # arr1 = [1 2 3]
arr2 = np.array([[1,2,3]])  # arr2 = [[1 2 3]]
arr3 = np.array([[1],[2],[3]])  # arr3 = [[1] [2] [3]]
arr4 = np.array([[1,2],[3,4]])  # arr4 = [[1 2][3 4]]
```

对于同样数据的数组，向量所需要的中括号是最少的，消耗的内存也是最少的。同理，列矩阵所消耗的内存是最大的。

2. 创建递增数组

使用 `np.arange()` 函数创建递增数组（arange全称为array-range）。括号里的参数和 `range()` 一样，分别是（首项，尾项，步长）。比如：

```
arr1 = np.array(10,20,2)  # arr1 = [10 12 14 16 18]。注意是左闭右开区间
```

3. 创建同值数组

使用 `np.zeros()` 和 `np.ones()` 函数，可以创建出全部为0和全部为1的数组，括号内的参数是**形状参数**。如果要变成全部为x的数组，只需给全为1的数组乘以x即可。比如：

```
arr1 = np.zeros(3)  # arr1 = [0. 0. 0.]
arr2 = np.ones((3,1))  # arr2 = [[1.] [1.] [1.]]
arr3 = 3.14 * np.ones(3)  # arr3 = [3.14 3.14 3.14]
```

注意， `np.zeros()` 和 `np.ones()` 生成的都是浮点型数组。这个设定可能是为了防止插进去的浮点数被截断。

4. 创建随机数组

使用 `np.random` 系列函数，可以创建出各式各样的随机数组。比如：创建**0到1之间均匀分布的浮点型随机数组**，可以使用 `np.random.random()` 函数，参数是形状参数。注意，**均匀分布**是指每个数字的产生是等概率的。

```
arr1 = np.random.random(5)
```

如果想要**改变范围**，只需要给这个函数配凑常数即可。例如，要创建a到b之间的随机数组，只需把基底设为a，并给每个数放大为(b-a)倍即可。用代码实现如下：

```
arr2 = a + (b - a) * np.random.random(5)
```

创建**整数型随机数组**，可以使用 `np.random.randint()` 函数，参数有三个，分别是（最小值，最大值，形状参数），范围仍然是**左闭右开区间**。

```
arr3 = np.random.randint(10, 100, (1,15))
```

其实创建整数型随机数组也可以借助浮点型随机数组，搭配 `.astype()` 实现。比如，上文的arr3可以这样实现：

```
arr4 = 10 + ((100 - 10) * np.random.random((1,15))).astype(int)
```

创建**服从正态分布的随机数组**，可以使用 `np.random.normal()` 函数，参数有三个，分别是（平均值，标准差，形状参数）。如果大家忘记了正态分布是什么，只需要记得随机数是平均值的概率最大，离平均值越远概率越小，在平均值两侧对称的数字概率相等；标准差越小，远离平均值时的概率降低速度就越快。实在无法理解的可以自行搜索。

```
arr5 = np.random.normal(0, 1, (2,3))
```

一种特殊的正态分布是**标准正态分布**，其平均值为0，方差为1。这种正态分布有专属的函数 `np.random.randn()` ，参数只有形状参数。

三、数组的索引

1. 访问与修改数组元素

与列表一样，numpy数组可以实现随机访问，只需要用**下标**索引即可。同时，也可以作为左值进行修改。

```
arr1 = np.arange(1,10)
print(arr1[3])  # 正序访问，输出为4
print(arr1[-2])  # 倒序访问，输出为8
arr1[3] = 666  # 修改数组元素
```

矩阵也同样可以访问与修改，下标可以使用传统的 `[x][y]` 索引，也可以使用 `[x,y]` 索引，其中x是行数，y是列数，都是从0开始的。我个人还是倾向于用前者，但是注意，传统的这种索引实际上是进行了两次索引（先找到第x行，再找到第y个元素），所以效率会降低。

```
arr2 = np.array([[1,2,3],[4,5,6]])
print(arr2[1,2])
print(arr2[1][2])
```

2. 花式索引

花式索引（Fancy indexing）可以同时返回数组中的多个元素。对于向量，只需要把原先的**下标**改成**下标列表**，返回的就是一个局部向量，比如：

```
arr1 = np.arange(0,90,10)
print(arr1[ [0,2,5] ])  # [0 20 50]
```

对于矩阵，只需要把原来的下标对改成**下标列表对**，返回的是一个向量。比如：

```
arr2 = np.arange(1,17).reshape(4,4)
print(arr2[ [0,1,2],[2,1,0] ])  # [3 6 9]，是arr2中[0,2],[1,1],[2,0]组成的向量
```

和普通索引一样，可以修改元素。如果想把多个元素同时赋值成同一个值，只用 `= 100`；如果赋值成不同值，也可以用列表赋值，比如 `= [100,200,300]`。如果赋值成不同值，需要注意列表的元素数量要和被赋值的元素数量一样，否则会报错。

普通索引使用一层中括号，花式索引使用**两层中括号**。

3. 数组切片

i. 向量的切片

向量的切片与列表切片操作完全一致，都是用 `[x:y:z]` 表示切出向量的 `[x,y)` 部分，每z个元素采样一次。比如：

```
arr1 = np.arange(10)
print(arr1[1:4])  # [1 2 3]
print(arr1[1:-7]) # [1 2 3]，倒数第7个数也是第4个数
print(arr1[1:])  # 切到结尾
print(arr1[:4])  # 从头开始切
print(arr1[1:4:2])  # [1 3]，每两个元素取一个
```

ii. 矩阵的切片

类似于索引，矩阵的切片其实就是对两个下标分别切片。但是注意，不能用两个中括号来索引了。比如：

```
arr2 = np.arange(1,21).reshape(4,5)
print(arr2[1:3,1:4])  # [[7 8 9][12 13 14]]
```

特别地，可以切出矩阵的一整行和一整列。值得一提的是，当切出矩阵的一整列时，输出的会是**一个向量**（系统为了**节省空间**的操作）。值得二提的是，当切出矩阵的一整行时，可以简写，**不需要写第二个下标**。

iii. 切片的本质是引用

这个观点十分重要，因此单独拿一段出来强调。当修改切片时，原来的数组也会**随之改变**，即**浅拷贝**。比如：

```
arr = np.arange(10)  # [0 1 ··· 9]
cut = arr[0:3]  # [0 1 2]
cut[0] = 100  # 修改cut，arr也会随之修改
print(arr)  # [100 1 2 ··· 9]
```

这样有个好处，就是能够节省内存，切片实际上与原数组占用的是同一片内存区域。因此，我们一般会使用切片，比如 `arr[:] = <表达式>` ，来替代 `arr = <表达式>` 。（其实我没看懂为啥能节省内存，有知道的朋友请在评论区为我解答）

同理，直接把一个数组赋值给另一个数组也是**浅拷贝**，比如 `cut = arr` ，如果修改cut，arr也会随之变化。

如果真的要创建新变量，即**深拷贝**，使用 `.copy()` 方法。比如：

```
arr = np.arange(10)  # [0 1 ··· 9]
copy = arr[0:3].copy()  # [0 1 2]
copy[0] = 100  # 修改copy，arr不会随之修改
print(arr)  # [0 1 2 ··· 9]
copy2 = arr.copy()  # 修改copy2，arr不会随之修改
```

四、数组的变形

1. 数组的转置

使用 `.T` 来**转置矩阵**。注意，只能转置矩阵，不能转置向量。因此，对于向量，要先变成矩阵。比如：

```
arr1 = np.arange(1,4)  # [1 2 3]
arr2 = arr1.reshape((1,-1))  # [[1 2 3]]
arr3 = arr2.T  # [[1] [2] [3]]
```

2. 数组的翻转

数组有两种翻转方式，一种是左右翻转，也就是把最后一列挪到第一列、倒数第二列挪到第二列、…、以此类推。另一种是上下翻转，就是把行倒着排。对应了两个函数，一个是**左右翻转**的 `np.fliplr()` ，即flip-left-right，另一个是**上下翻转**的 `np.flipud()` ，即flip-up-down。参数是被翻转的数组。

注意：**向量只可以上下翻转**。是不是很反常识？其实在NumPy的世界观里，向量都是列向量。

```
arr1 = np.arange(10)
arr_ud = np.flipud(arr1)  # [9 8 7 ··· 0]
```

仍然值得一提的是，经过我的尝试，如果修改了arr_ud，那么原数组arr1**也会被修改**，其实和切片是一样的。你也可以理解成翻转是一种特殊的切片。

3. 矩阵的重塑

用的是我们之前已经提及的 `.reshape()` 方法，参数是形状参数。既然讲过了，并且之前用过很多次了，我就不举例了，但是还是强调一下：可以留一个维度写-1，让系统自己算。这个真的很实用。

4. 数组的拼接

两个向量拼接，得到的是**加长版向量**。使用 `np.concatenate()` 函数实现，参数是一个列表，包含了需要拼接的向量。比如：

```
arr1 = np.array([1,2,3])
arr2 = np.array([4,5,6])
arr3 = np.concatenate([arr1,arr2])
print(arr3)  # [1 2 3 4 5 6]
```

两个矩阵仍然是用上述函数拼接，但是多了一个参数axis，表示沿着哪个方向拼接。默认值是0，即沿着行的方向拼接；也可以加上 `np.concatenate([], axis=1)` ，改成按列拼接。注意，沿着行拼接时，**行数要一样**；沿着列拼接时，**列数要一样**，否则会报错。

<拓> axis = -1表示按照最后一个维度进行拼接。

向量和矩阵**不能直接拼接**！需要先把向量转换成行矩阵。

5. 数组的分裂

一个向量分裂，使用 `np.split()` 函数实现，参数为（向量名，[断点列表]）。其中，断点列表是由整数构成的列表，表示在**这个位置之前断开**。返回的值也是一个列表，包括了被断开的向量。比如：

```
arr = np.arange(1,10)
arr1,arr2,arr3 = np.split(arr, [2,8])
print(arr1)  # [1 2]
print(arr2)  # [3 4 5 6 7 8]
print(arr3)  # [9]
```

矩阵的分裂也是用上述函数，但是还是需要注明分裂的方向，即参数为（向量名，[断点列表]，axis=0）。默认值为0，即沿着行切；设为1，则沿着列切。

五、数组的运算

数组可以与常数、与其他数组做运算。

1. 数组与常数做运算

数组与常数计算，就是数组中的每个数字都与常数做相应的运算。因此，加、减、乘、除、幂、括号、整除、求余等运算符都与Python完全一样。以防大家忘记Python有哪些运算符，这里还是再列举一下，举例就懒得写了。

运算符	含义
+、-、*、/	加、减、乘、除
**	幂
()	修正运算次序
//	整除
%	求余

2. 数组与数组之间的运算

如果用上文的各种运算符作用在数组之间的运算，那么会进行**逐元素计算**（element-wise），这要求两个数组的形状完全相同。比如：

```
arr1 = [[1 2 3][4 5 6]]  # 大家创建np数组的时候一定不要这么写啊
arr2 = [[7 8 9][10 11 12]]  # 我在偷懒
print(arr1 * arr2)  # [[7 16 27][40 55 72]]
```

在这里我发现一个比较好玩的事情：如果赋值 `arr2 = -arr1` ，此时修改arr2是不会影响arr1的。也就是说，创建 `arr2 = -(-arr1)` 的话，可以直接实现**深拷贝**！真是太有意思了（）

3. 广播

广播是为了处理**不同形状的数组之间的运算**。不同形状的数组有如下运算规则：

- 向量与矩阵做运算，向量自动被升级为**行矩阵**；
- 如果某矩阵是行矩阵或列矩阵，则其被**广播**，以适配另一个矩阵的形状。

需要提前说的是，一般只推荐对向量进行广播，后两种基本上不会使用，因为实在是太反直觉了。

i. 向量被广播

当一个形状为(x,y)的矩阵与一个向量做运算时，要求该向量的**形状必须为y**。运算时，这个向量会自动升级成(1,y)的行矩阵，再被自动广播为(x,y)的矩阵，每一个元素都会向下复制x次。比如：

```
arr1 = [-100 0 100]
arr2 = [[1 2 3][4 5 6]]
print(arr1 * arr2)   # [[-100 0 300] [-400 0 600]]
```

ii. 列矩阵被广播

行矩阵被广播的情况就和向量被广播的情况是一样的，此处不再赘述。这里讲解列矩阵如何被广播。

当一个形状为(x,y)的矩阵与一个列矩阵做运算时，要求该列矩阵的形状必须为(x,1)，它会被自动广播为(x,y)的矩阵，每一个元素都会向右复制y次。比如：

```
arr1 = [[0] [1] [2]]
arr2 = [[1 2] [3 4] [5 6]]
print(arr1 * arr2)  # [[0 0] [3 4] [10 12]]
```

iii. 行矩阵与列矩阵同时被广播

一个(1,y)的行矩阵与一个(x,1)的列矩阵相乘，两个矩阵会同时广播成(x,y)的矩阵，然后再相乘。通过简单的计算，可以知道，这种乘法得到的矩阵，**第i行第j列的值**是由行矩阵的第i个元素和列矩阵的第j个元素相乘得到的。懒得举例了，大家可以自行尝试。

六、数组的函数

本节讲一些常用的特殊函数。

1. 矩阵乘积

终于回归了线性代数中最熟悉的矩阵乘法，通过 `np.dot()` 函数实现。值得一提的是，当乘积中有**向量**时，它会根据需要，随意变成行矩阵或列矩阵，但是输出的结果**必须是向量**！这一点非常重要，接着看下去你就明白了。

i. 向量与向量的乘积

两个向量相乘，一定是**前者为行向量，后者为列向量**，输出为一个**数字**。向量长度**必须相同**。比如：

```
arr1 = np.arange(5)
arr2 = np.arange(5)
print(np.dot(arr1,arr2))  # 30
```

提问：为什么不是前者变成列向量，后者变成行向量？如果这样，输出的将是一个 5×5 的矩阵，不符合规定。

ii. 向量与矩阵的乘积

向量**在前面就会变成行向量，在后面就会变成列向量**。这是为了保证输出结果为向量。比如：

```
arr1 = [1 2]
arr2 = [[300 300] [600 600]]
print(np.dot(arr1,arr2))  # [1500 1500]
print(np.dot(arr2,arr1))  # [900 1800]
```

iii. 矩阵与矩阵的乘积

这个就不必多说了，和线性代数里的规则一样的。

2. 数学函数

这里介绍一部分最重要的数学函数。我只会列举函数、参数与它的效果，不会举例说明，大家可以自行尝试，如果发现了什么彩蛋可以在评论区告诉我。

函数名	参数	作用
np.abs()	(数组)	对每个元素取绝对值
np.sin()	(数组)	对每个元素取正弦，输出为科学计数法
np.cos()	(数组)	对每个元素取余弦，输出为科学计数法
np.tan()	(数组)	对每个元素取正切，输出为科学计数法
np.pi	无	π
np.exp()	(数组)	每个元素x变成e^x
np.log()	(数组)	每个元素x变成lnx

这里提两个我发现的小彩蛋：第一、 `tan(pi/2)` 不会直接输出无穷大，而是一个非常非常大的数字；第二、如果要把每个数字x变成log_a(x)，只需用**换底公式**计算即可。由于我的网站暂时疑似不支持latex公式，我只好让你们自己去搜换底公式了。

3. 聚合函数

我不知道为啥叫这个名，反正就是介绍几个常见的函数。

i. 最大值、最小值

使用 `np.max()` 和 `np.min()` 函数。参数是（数组，axis）。如果数组是向量，则不需要axis值；是矩阵的话，如果不写，返回的是整个矩阵中的最大（或最小）元素；如果axis = 0，则把矩阵**压缩成一个向量，每一个元素是原本每列中最大（或最小）的数字**；axis = 1时则相反。比如：

```
arr = [[1 2 3] [4 5 6]]
print(np.max(arr))  # 6
print(np.max(arr, axis = 0))  # [4 5 6]
print(np.max(arr, axis = 1))  # [3 6]
```

ii. 求和、求积

使用 `np.sum()` 和 `np.prod()` 函数。参数和上述完全一样。如果axis = 0，则把矩阵**压缩成一个向量，每一个元素是原本每列的和或积**。比如：

```
arr = [[1 2 3] [4 5 6]]
print(np.sum(arr))  # 21
print(np.sum(arr, axis = 0))  # [5 7 9]
print(np.sum(arr, axis = 1))  # [6 15]
```

iii. 均值、标准差

使用 `np.mean()` 和 `np.std()` 函数。参数和上述完全一样。比如：

```
arr = [[1 2 3] [4 5 6]]
print(np.mean(arr))  # 3.5
print(np.mean(arr, axis = 0))  # [2.5 3.5 4.5]
print(np.mean(arr, axis = 1))  # [2 5]
```

iv. 聚合函数的注意事项

- 关于axis的理解：axis = 0 时，向量的长度保持**行长度**；axis = 1 时，向量的长度保持**列长度**
- 由于大型数组可能出现**数据缺失**情况，所有聚合函数有安全版本：在函数前面加一个**nan**，比如 `np.nanmin()` 等等。如果数组里有数字缺失，会直接忽略这个数字。

七、布尔型数组

除了整数型数组和浮点型数组，还有一种常用的数组叫**布尔型数组**。

1. 创建布尔型数组

布尔型数组的创建需要借助各种**比较符号**：`>`，`<`，`==`，`!=`，≥，≤。

我们可以把数组与单个数字比较，也可以将两个相同形状的数组比较。比如：

```
arr1 = [[1 2 3] [4 5 6]]
print(arr1 > 4)  # [[False False False] [False True True]]
arr2 = [[6 5 4] [3 2 1]]
print(arr1 > arr2)  # 懒得写了
```

连接不同的布尔型数组，依靠与 & 、或 | 、非 ~ 。注意，这里和Python可不一样了！

2. True的数量

统计布尔型数组中True的数量，有三种函数：

- `np.sum()` 函数，统计True的个数。
- `np.any()` 函数，只要有一个True，就返回True。
- `np.all()` 函数，全为True才返回True。

比如：

```
arr1 = [1 2 3 4 5]
arr2 = [5 4 3 2 1]
arr_bool = (arr1 == arr2)
print(np.sum(arr_bool))  # 1
print(np.any(arr_bool))  # True
print(np.all(arr_bool))  # False
```

3. 布尔型数组作为掩码

布尔型数组的第一个作用，就是以布尔型数组为索引，找出数组中所有满足条件的元素。比如：

```
arr1 = [[1 2 3] [4 5 6]]
arr2 = arr1[arr1 > 4]
print(arr2)  # [5 6]
```

注意，在掩码之后，数组会退化成向量。

4. 布尔型数组寻找满足条件的元素所在位置

布尔型数组的第二个作用，是寻找满足条件的元素所在的位置。这需要使用 `np.where()` 函数，参数是一个布尔型数组，返回的值是一个**元组**，元组只有一个元素，是所有满足条件的**元素下标组成的列表**。如果加上一个下标[0]，就返回一个数组。比如：

```
arr = [1 2 3 4 5]
print(np.where(arr > 2))  # (array([2,3,4]),)
print(np.where(arr > 2)[0])  # [2 3 4]
```

那就有一个小连招，能返回数组的最大值下标： `np.where(arr == np.max(arr))[0]` 。

八、从数组到张量

PyTorch作为当前首屈一指的深度学习库，吸收了NumPy的语法，又将使用CPU的数组进步到使用GPU的张量，运算速度也提高了。要调用这个库，需要安装**torch库**。注意PyCharm中，pytorch库并非官方库，不要下载错误。

1. 数组与张量

NumPy与PyTorch的基础语法基本一致，区别在于：

- np换成torch；
- array换成张量tensor；
- n维数组换成n维张量。

数组与张量之间也能互相转化：把数组arr换成张量ts，只需 `ts = torch.tensor(arr)`；反过来，就是 `arr = np.array(ts)`。

2. 语法不同点

PyTorch只是少量修改了上文中NumPy的部分函数或方法。在原视频UP主的努力下，发现了这些改变，感谢UP主。下面是两者的不同点比较：

函数作用	NumPy的函数	PyTorch的函数	PyTorch中的特殊点
数组类型	`.astype()`	`.type()`	无
01随机数组	`np.random.dandom()`	`torch.rand()`	无
随机整数数组	`np.random.randint()`	`torch.randint()`	不接纳向量
随机正态数组	`np.random.normal()`	`torch.normal()`	不接纳向量
随机标准正态数组	`np.random.randn()`	`torch.randn()`	无
复制数组	`.copy()`	`.clone()`	无
数组拼接	`np.concatenate()`	`torch.cat()`	无
数组分裂	`np.split()`	`torch.split()`	第二个参数改为分裂后长度的列表
矩阵乘积	`np.dot()`	`torch.matmul()`	无
矩阵乘积	`np.dot(v,v)`	`torch.dot()`	向量乘积
矩阵乘积	`np.dot(v,m)`	`torch.mv()`	向量乘以矩阵
矩阵乘积	`np.dot(m,m)`	`torch.mm()`	矩阵乘积
指数	`np.exp()`	`torch.exp()`	不接纳矩阵
对数	`np.log()`	`torch.log()`	不接纳矩阵

函数作用	NumPy的函数	PyTorch的函数	PyTorch中的特殊点
平均数	`np.mean()`	`torch.mean()`	只接纳浮点型数组
标准差	`np.std()`	`torch.std()`	只接纳浮点型数组

3. 预告片

PyTorch作为深度学习的库，肯定是适配了很多深度学习的函数，不过这并非本节主题，我就不介绍了，大家可以期待我更新PyTorch教程！

九、总结

内容实在太多了，大家不一定要记住，只要对有哪些函数有大概的印象，再来这个教程里寻找相应函数即可。希望对大家有帮助！