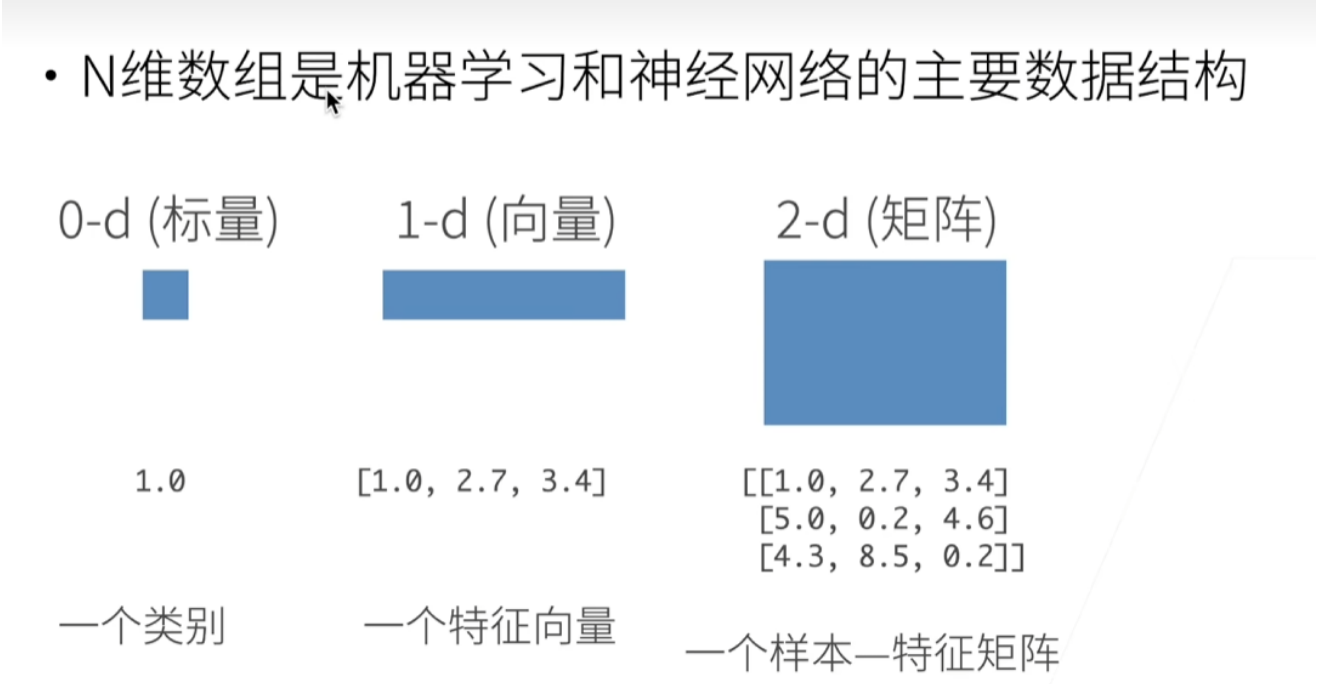
03.数据操作+数据预处理

数据操作

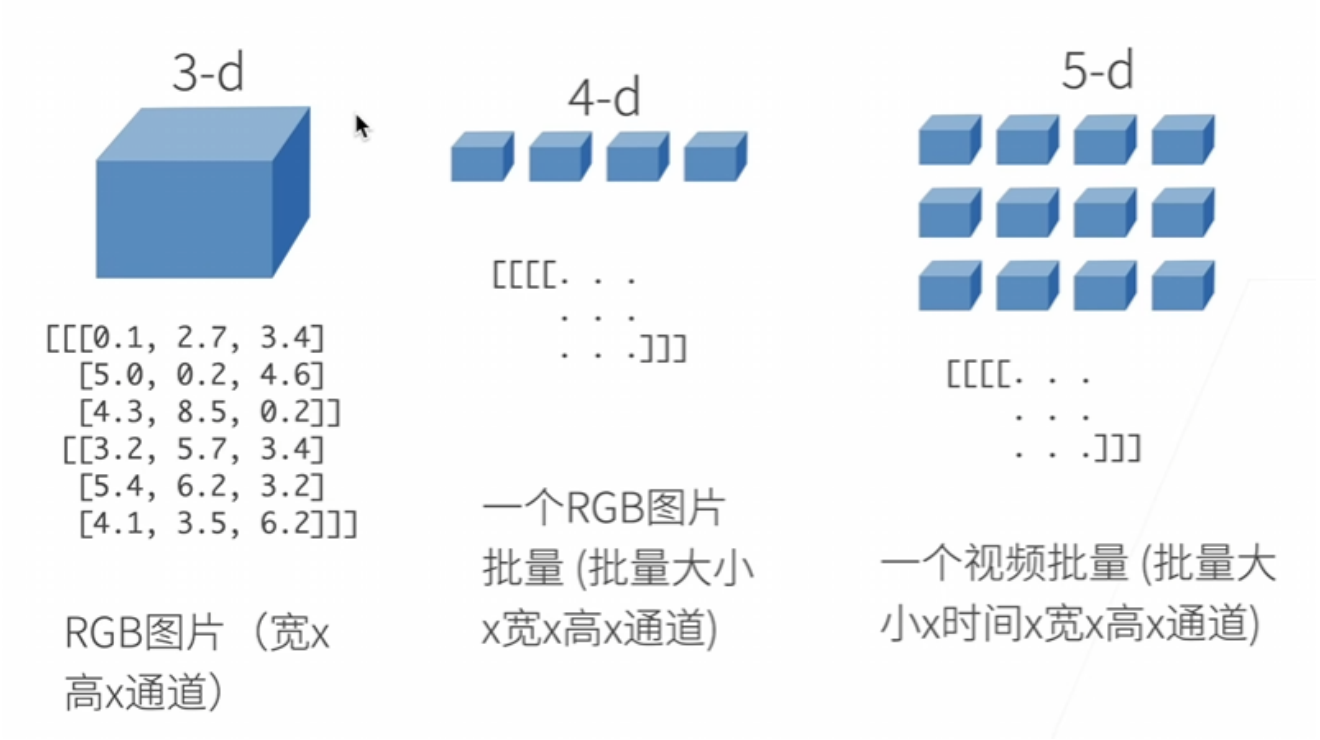
N维数组

N维数组是机器学习和神经网络中主要的数据结构。

如下图所示：



其中我们把**一个单值称为标量。记作0-d**
一个**一维数组通常作为特征向量，记作1-d**
二维数组即线性代数中常见的矩阵，我们称为特征矩阵，记作2-d



如上图所示，对于更高维的数组，我们有比较常用的用法。

一个3-d数组，也就是三维数组，通常用于储存RGB图片（宽x高x通道）
其中列数就是图像的宽度，行数即为图像的高度，每个基本元素由一个一维数组表示其RGB通道的值。

一个4-d数组，通常用于表示一个RGB图片批量，在深度学习中，我们称为一个Batch。
实际上，这个4-d数组就是由多个3-d数组组成的，例如在读取数据集的时候一次性读取128章节图像，那么这个Batch的大小就是128。

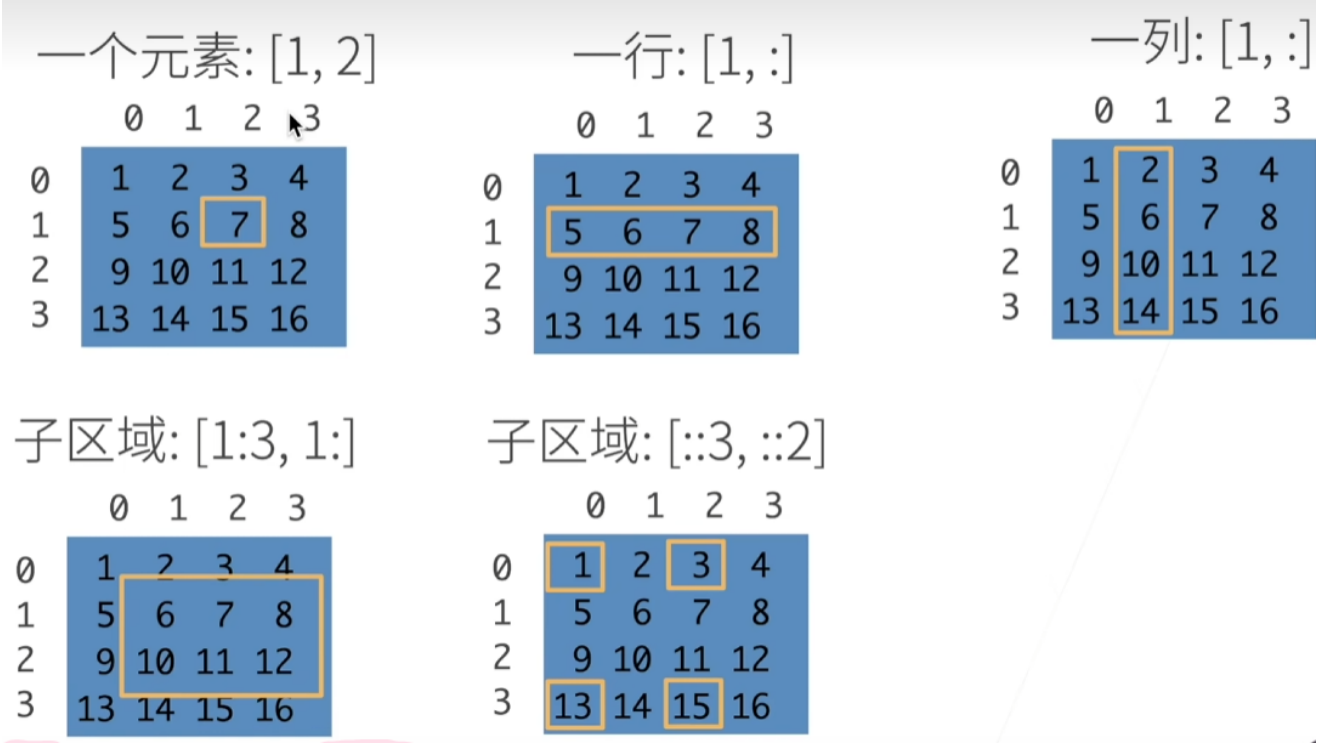
5-d数组典型是一个视频批量，由批量大小、时间、宽、高、通道组成。

创建数组

创建数组需要给定三个参数：

1. 形状：例如3x4的矩阵
2. 数据类型：指定每个数组元素的数据类型
3. 元素值：指定每个数组元素的值，例如全为0或者随机数。

访问数组



如上图所示，访问数组元素的基本方法是：

```
var[start_row:end_row:step_row,start_col:end_col:step_col]
```

例如，访问数组中N行M列的元素：

```
var[n,m]
```

访问第一行的元素：

```
var[1,:]
```

访问第一列的元素：

```
var[:,1]
```

访问某个区域的元素，这里以第二行到第三行，第二列到结尾为例：

```
var[1:3,1:]
```

访问某个区域中以指定步长间隔的元素，这里以所有行列的元素，在行中步长为3，列中步长为2访问：

```
var[::3,::2]
```

其中，步长为3指下一个访问的元素是当前索引+3，例如第一个访问的元素是0，则下一个是0+3=3，而不是中间间隔三个元素。

访问最后一个元素：

```
var[-1]
```

数据操作实现

以上的关于数据操作的描述是基于Pytorch的张量，在数据操作的实现上，也采用Pytorch进行。
首先导入Pytorch：

```
import torch
```

关于张量

在PyTorch中，张量（Tensor）是最基本的操作对象。它是一个多维数组，类似于NumPy的ndarray，但增加了对GPU加速计算的支持

1.创建张量

可以从已有的数据中创建张量，通过`torch.tensor()`方法：

```
list = [[1,2,3],[3,4,5]]
tensor = torch.tensor(list)
print(tensor)
"""
tensor([[1, 2, 3],
        [3, 4, 5]])
"""
```

使用内置函数创建：
torch内置较多的用于创建张量的函数，这里将他们以表格的形式列出。

函数名	参数	说明	示例

torch.arange(start,end,step)	start:开始
	end：结尾
	step:步长
		创建从start到end-1的范围等步长的张量，	`torch.arrange(0,12)`
			`# 创建从0到11的12个元素的一维量。`
			`# tensor([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])`

函数名	参数	说明	示例
		元素数量不骨固定	

torch.linspace(start,end,step)	start:开始
	end:结尾
	steps：数量
		创建从start到end-1数量为steps的张量，元素数量即为steps	# 从0到1的等间隔5个值 tensor
			torch.linspace(0, 1, steps=5)
			print(tensor)
			“”“
			tensor([0.0000, 0.2500, 0.5000
			0.7500, 1.0000])
			“”“

torch.eye(n)	n:指定创建nxn的矩阵	创建一个nxn的单位矩阵，对角元素为1，其余元素为0.	# 3x3单位矩阵 tensor = torch.eye(3) print(tensor) """ tensor(1., 0., 0.], [0., 1., 0.] [0., 0., 1.) """
torch.full((row,col),vlaue)	row:行数 col：列数 value：填充值	以给定值value填充生成row*col大小的矩阵。	`# 所有元素都为7的2x3张量` `tensor = torch.full((2, 3)` `7)` `print(tensor)` `"""` `tensor(7, 7, 7], [7, 7,` `"""`

torch.rand(row,col)	row:行
	col：列
		创建一个row行col列的随机元素值的张亮	# 创建一个随机张量
			random_tensor = torch.rand(2
			print(random_tensor)

torch.empty_like(tensor)	tensor:参考张量	参考tensor张量创建一个未初始化的相同形状的张量	data = torch.rand(3,3) data1 = torch.empty_like(data) print(data1)
torch.zeros_like(tensor)	同上	参考tensor张量创建一个形状相同的，值全为0的张量。	data = torch.full((5,6),0.01) tensor = torch.zeros_like(data) print(tensor)
torch.ones_like(tensor)	同上	与zeros_like类似，创	data = torch.linspace(0,1,10) tensor = torch.ones_like(data) print(tensor)

函数名	参数	说明	示例
		建全为1的张量。	
torch.rand_like(tensor)	同上	与上述类似，创建一个内容随机的张量	data = torch.eye(5) x=torch.rand_like(data) print(x)
torch.randint(start,end,type)	start:int 开始值 end: int 结束值 type:tuple 形状	创建元素值范围在[start,end)之间的整数张亮，形状为type	data = torch.randint(0,20,(3,3)) print(data)
torch.normal(avg,std,type)	avg:均值 std:方差 type:形状	创建服从标准正态分布的张量、	data = torch.normal(125.32,10 (5,8)) print(data)
torch.bernoulli(tensor,probability)	tensor：参考张量 probability:概率	从tensor中创建一个概率为probability的服从伯努利分布的张量	data = torch.bernoulli(torch.rand(5,5), print(data)
torch.bartlett_window(length)	length:长度	创建一个长度为Length的Bartlett窗口张亮，通常用于信号处理。	data = torch.bartlett_window(1 print(data)
torch.hamming_window(length)	length：长度	创建一个长度为10的Hamming窗口张量，常用于信号处理。	data = torch.hamming_window print(data)
torch.hann_window(length)	同上	创建一个hann窗口张量，常	data = torch.hann_window(20) print(data)

函数名	参数	说明	示例
		用于信号处理。	

2.针对张量的属性和方法

张量实际上是Torch中的一种自定义数据类型对象，因此有针对张量的一系列属性和方法。

张量的常见属性：

属性	说明	示例代码	输出
tensor.shape	用于获取张量的形状	torch.Size([3, 4])	/
tensor.size	同上	torch.Size([5,6])	/
tensor.dtype	用于获取张量的数据类型，在一个张量中，所有的元素具有同一个数据类型	data = torch.randint(0,20,(3,3)) print(data.dtype)	torch.int64
tensor.device	获取张量所在的设备	tensor = torch.rand(3, 4) print(tensor.device) # 输出: cpu if torch.cuda.is_available(): tensor = tensor.to('cuda') print(tensor.device) # 输出: cuda:0	cpu
requires_grad	张量是否需要计算梯度	data = torch.rand(5,8,requires_grad=True) print(data.requires_grad) a = torch.rand_like(data) print(a.requires_grad)	True False
is_leaf	表示张量是否是计算图中的叶子节点。只有叶子节点可以计算梯度。	x = torch.rand(3, 4, requires_grad=True) y = x + 2 print(x.is_leaf) # 输出: True print(y.is_leaf) # 输出: False	True False
grad	储存张量的梯度，只在 `requires_grad=True` 的情况下有效。	x = torch.tensor(3.33, requires_grad=True) y = x ** 2 y.backward() print(x.grad)	tensor(6.66(
is_cuda	表示张量是否储存在GPU上	tensor = torch.rand(3, 4) print(tensor.is_cuda) # 输出: False i·f torch.cuda.is_available(): tensor = tensor.to('cuda')	False

属性	说明	示例代码	输出
		print(tensor.is_cuda) # 输出: True	
T	张量的转置，适用于二维向量。	tensor = torch.randint(0,100, (10,12)) print(tensor.T)	...
data	访问张量的数据，返回一个不需要梯度的张量	tensor = torch.rand(3, 4, requires_grad=True) print(tensor.data)	...
names	张量的命名维度（适用于命名张量）。	tensor = torch.rand(3, 4, names=('batch', 'features')) print(tensor.names)	('batch', 'features')

同样的，作为一个对象，张量不仅具有丰富的属性，还具有可供开发者调用的方法：

方法名	参数	说明	示例
numel()	None	获取张量的元素总数	data = torch.ra print(data.nume
view(row,col)	row:变形后的行 col:变形后后的列	改变张量的形状	tensor = torch.r reshaped_tens tensor.view(3, 2 print(reshaped_
reshape(row,col)	同上	与view类似，但是更加灵活和常用。	tensor = torch.r reshaped_tens tensor.reshape print(reshaped_
transpose(dim0,dim1)	dim0:第一个参与交换的维度 dim1:第二个参与交换的维度。	等同于torch.transpose(input,dim0,dim1) 交换张量的两个维度，例如对于一个二维张量（矩阵）进行交换就是将矩阵的行列交换。	tensor = torch.t 5, 6) print("Original T print(tensor) transposed_ter tensor.transpos print("\nTransp print(transpose
permute(dim0,dim1,dim2...)	dimx:要排列的维度	将张量的维度按指定顺序排列。	tensor = torch.r permuted_tens tensor.permute print(permuted_

方法名	参数	说明	示例
squeeze()	None	移除长度为1的维度	tensor = torch.r print(tensor.squ
unsqueeze(index)	index：位置	在指定位置添加一个维度	tensor = torch.r print(tensor.uns
add(n)/sub(n)/mul(n)/div(n)	n：被运算数	元素级加减乘除法，即针对每一个元素的操作。	tensor = torch.r print(tensor) print(tensor.add print(tensor.sub print(tensor.mu print(tensor.div
matmul(tensor)	tensor：另一个二维张量	矩阵乘法	tensor = torch.r tensor1 = torch print(tensor.ma
pow(n)	n:n次幂	元素级幂运算	tensor = torch.tensor([16 print(tensor.pow
exp()	None	元素级指数运算。	..
sum()	None	张量求和，返回包含一个元素的标量。	data = torch.no (3,4)) print(data) print(data.sum(
mean()	None	计算张量均值，返回一个标量型张量	tensor = torch.r result = tensor. print(result)

方法名	参数	说明	示例
max()	None	返回张量中的最大值	tensor = torch.tensor([1, print(tensor.ma
min()	None	返回张量中最小值	...
argmax()/argmin()	None	返回张量中最大、最小值的索引位置。	
clone()	None	克隆张量	
detach()	None	返回新的张量，从当前计算图中分离出来。	tensor = torch.r requires_grad= detached_tens tensor.detach() print(detached_
to(param)	param: 设备或者数据类型	将张量移动到某个设备，或者转换数据类型。 注意：通过to方法转换数据类型返回新的张量而不是修改原张量类型。	data = torch.ra data = data.to(t print(data.dtype
cpu()/cuda()	None	将张量移动到CPU或者GPU	
numpy()	None	将张量转为Numpy数组	
item()	None	将单元素张量转为Python标量数据类型。	

张量的运算

如上述关于张量的一系列方法，张量的元素级运算也可以通过标准的Python运算符实现。
这里底层的实现是Torch在Tensor类中重写了 `__add__()` 等用于运算的方法：

```
1804        def __abs__(self) -> Tensor: ...
1805        def __add__(self, other: Any) -> Tensor: ...
1806        @overload
1807        def __and__(self, other: Tensor) -> Tensor: ...
1808        @overload
1809        def __and__(self, other: Union[Number, _complex]) -> Tensor:
1810        @overload
1811        def __and__(self, other: Any) -> Tensor: ...
1812        def __bool__(self) -> builtins.bool: ...
1813        def __complex__(self) -> builtins.complex: ...
1814        def __div__(self, other: Any) -> Tensor: ...
1815        def __eq__(self, other: Any) -> Tensor: ...  # type: ignore[o
1816        def __float__(self) -> builtins.float: ...
1817        def __floordiv__(self, other: Any) -> Tensor: ...
1818        def __ge__(self, other: Any) -> Tensor: ...
1819        def __getitem__(self, indices: Union[Union[SupportsIndex, Uni
1820        def __gt__(self, other: Any) -> Tensor: ...
1821        def __iadd__(self, other: Any) -> Tensor: ...
1822        @overload
```

在使用中不必关注这些。

标准算数运算

包含加减乘除和幂运算都是受支持的元素级标准运算符。

Example：

```
tensor = torch.tensor([25,16,9,49,81,36])
print(tensor + 10)
print(tensor - 5)
print(tensor * 1.5)
print(tensor / 1.5)
print(tensor ** 0.5)
```

Output:

```
tensor([35, 26, 19, 59, 91, 46])
tensor([20, 11,  4, 44, 76, 31])
tensor([ 37.5000,  24.0000,  13.5000,  73.5000, 121.5000,  54.0000])
tensor([16.6667, 10.6667,  6.0000, 32.6667, 54.0000, 24.0000])
tensor([5., 4., 3., 7., 9., 6.])
```

其他运算

1. **张量组合**
   我们还可以通过.cat()方法将两个张量结合起来。

语法：

```
torch.cat((X,Y),dim=z)
```

其中：

- X：要被合并的第一个张量
- Y：要被合并的第二个张量
- z：在哪个维度进行合并，例如dim=0则在行合并，dim=1则在列合并。

Example:

```
X = torch.arange(15,dtype=torch.float32).reshape((3,5))
Y = torch.randint(0,20,(5,5))
print("tensor X:",X)
print("\r\ntensor Y:",Y)
print("\r\n combine with dim 0",torch.cat((X,Y),dim=0))
print("\r\n combine with dim 1",torch.cat((X,Y),dim=1))
```

Output:

```
tensor X: tensor([[ 0.,  1.,  2.,  3.,  4.],
        [ 5.,  6.,  7.,  8.,  9.],
        [10., 11., 12., 13., 14.]])

tensor Y: tensor([[11, 14,  9,  7,  6],
        [ 7,  4,  1,  2,  9],
        [ 2, 14,  6,  8, 12]])

 combine with dim 0 tensor([[ 0.,  1.,  2.,  3.,  4.],
        [ 5.,  6.,  7.,  8.,  9.],
        [10., 11., 12., 13., 14.],
        [11., 14.,  9.,  7.,  6.],
        [ 7.,  4.,  1.,  2.,  9.],
        [ 2., 14.,  6.,  8., 12.]])

 combine with dim 1 tensor([[ 0.,  1.,  2.,  3.,  4., 11., 14.,  9.,  7.,
6.],
        [ 5.,  6.,  7.,  8.,  9.,  7.,  4.,  1.,  2.,  9.],
        [10., 11., 12., 13., 14.,  2., 14.,  6.,  8., 12.]])
```

Notice:根据几次调整shape和dim的值发现，在某一个维度合并，必须确保其他的维度的size是相同的，而当前的维度可以不同。

例如：

```
X = torch.arange(32,dtype=torch.float32).reshape((1,2,16))
Y = torch.randint(0,16,(1,2,8))
print("tensor X:",X)
print("\r\ntensor Y:",Y)
print("\r\n combine with dim 2",torch.cat((X,Y),dim=2))
```

合并在第二维度合并，则第零维和第一维的size必须相同，而第二维可以不同，如果将X,Y的shape调整为：

```
X = torch.arange(32,dtype=torch.float32).reshape((1,2,16))
Y = torch.randint(0,16,(1,1,16))
```

则会抛出如下错误：

```
RuntimeError: Sizes of tensors must match except in dimension 2. Expected
size 2 but got size 1 for tensor number 1 in the list.
```

表明在第二维度的期望尺寸与实际尺寸不符合，期望的尺寸是2，但是实际的尺寸是1。

2. **通过逻辑运算构建二元张量**

当对两个形状相同的张量进行逻辑运算时，产生的结果是由布尔值组成的相同尺寸的张量。

Example：

```
# 按逻辑产生二元张量
X = torch.arange(10,dtype=torch.float32).reshape((2,5))
Y = torch.randint(0,10,(2,5))
Z = X == Y
print("tensor X:",X)
print("\r\ntensor Y:",Y)
print("X == Y: ",Z)
print("X != Y: ",~Z)
```

Output:

```
tensor X: tensor([[0., 1., 2., 3., 4.],
        [5., 6., 7., 8., 9.]])

tensor Y: tensor([[3, 5, 5, 6, 4],
        [6, 7, 4, 2, 9]])
X == Y:  tensor([[False, False, False, False,  True],
        [False, False, False, False,  True]])
```

```
X != Y:  tensor([[ True,  True,  True,  True, False],
        [ True,  True,  True,  True, False]])
```

广播机制

在PyTorch中，广播机制（broadcasting）允许不同形状的张量在数学运算中自动扩展为相同的形状，从而实现元素级操作。广播机制使得编写代码更加简洁和高效，无需手动调整张量的形状。

tensor的广播机制与Numpy的广播机制类似，主要有以下三个原则：

1. 当两个张量维度不同时，会将较小的维度增加一个维度来匹配较大的维度
2. 如果两个张量在某个维度上不同，但是有一个张量在该维度上的大小是1，则会在这个大小为1的维度上复制以和较大的维度相同。
3. 如果两个张量维度不同，并且大小又都不为1，则无法进行广播，也就是无法进行运算。

总结一下，核心就是：必须有一个维度大小为1才可以进行广播机制来使得两个不同大小的维度进行广播运算。

Example：

```
X = torch.rand(1,3,dtype=torch.float32)
Y = torch.normal(1.33,5,(2,1))
Z = X+Y
print("tensor X:",X)
print("\r\ntensor Y:",Y)
print("X + Y: ",Z)
```

张量X和张量Y在0和1维的大小不同，但是都有一个大小为1的维度，那么此时可以进行广播机制的运算，产生的结果是将X的0维大小复制为2，将Y的1维大小复制为，然后进行运算。

Output：

```
tensor X: tensor([[0.9192, 0.4674, 0.6165]])

tensor Y: tensor([[-3.2659],
        [ 5.1828]])
X + Y:  tensor([[-2.3467, -2.7986, -2.6494],
        [ 6.1020,  5.6502,  5.7993]])
```

例如：

```
X = torch.rand(1,3,dtype=torch.float32)
Y = torch.normal(1.33,5,(2,2))
```

```
Z = X+Y
```

X和Y在第一维的尺寸不同，且都不为1，则无法进行广播运算。

广播运算还有一个值得注意的点是：

如果对于一个神经网络模型，总是在没有出错的前提下产生不可预料的结果，则需要检测是否因为某个张量的某个维度值为1，导致进行广播运算。

因此，我们通常会在模型前向传播之前，调用张量的 `squeeze()` 归一化方法来移除大小为1的维度来防止广播机制产生意料外的结果。

张量值的修改

与访问张量的值相同，修改张量的值即通过访问张量对应的值，然后修改即可。

Example：修改第二行第四列的值：

```
X = torch.randint(0,20,(3,5))
print("tensor X:",X)
X[1,3]=114514
print("tensor X after edit:",X)
```

Output:

```
tensor X: tensor([[ 7,  7, 13, 12,  9],
        [ 6,  4,  4,  1,  3],
        [ 4, 14,  3, 18,  4]])
tensor X after edit: tensor([[     7,      7,     13,     12,      9],
        [     6,      4,      4, 114514,      3],
        [     4,     14,      3,     18,      4]])
```

Example：修改第三行，第1到4列的值：

```
X = torch.randint(0,20,(3,5))
print("tensor X:",X)
X[2,0:4]=torch.tensor([0,1,2,3])
print("tensor X after edit:",X)
```

Output:

```
tensor X: tensor([[17, 11,  3,  2, 18],
        [ 9, 15,  5,  0,  7],
        [15, 17,  3,  8,  9]])
tensor X after edit: tensor([[17, 11,  3,  2, 18],
```

```
        [ 9, 15,  5,  0,  7],
        [ 0,  1,  2,  3,  9]])
```

Examle: 修改第一行的值全为0：

```
X = torch.randint(0,20,(3,5))
print("tensor X:",X)
X[0,:]=0
X[0,:]=torch.tensor([0])
X[0,:]=torch.tensor([0,0,0,0,0])
# 以上三种方法均可
print("tensor X after edit:",X)
```

内存分配相关

由于Python的性能本身就比较有限，而在Torch中又有很多操作会导致内存的重新分配。
大体来说，调用张量类的一系列方法，例如`clone()`、`int()`、`view()`等方法通过一个张量声明另一个张量都会产生新的内存分配，这里由于声明新的变量了，所以无法避免重新分配内存。

这里主要讨论算数运算导致的内存分配。

例如：

```
X = torch.rand(1,3,dtype=torch.float32)
before = id(X)
X = X + 1
id(X) == before
# False
```

这里很显然，当X对自身进行数学运算时，产生了新的内存分配。

解决这种问题，我们可以使用**原地操作。**

PyTorch 提供了许多原地操作，这些操作会直接修改原始张量而不分配新的内存。原地操作通常以 `_` 结尾，例如 `add_()`，`sub_()`，`mul_()`，`div_()` 等。

Example:

```
X = torch.rand(1,3,dtype=torch.float32)
before = id(X)
print("tensor X:",X)
X.add_(1)
print("tensor X:",X)
id(X) == before
```

Output:

```
tensor X: tensor([[0.2325, 0.6518, 0.9731]])
tensor X: tensor([[1.2325, 1.6518, 1.9731]])
True
```

同时也可以预先分配内存并重用，在计算之前预先分配好内存，并在计算过程中重用这些内存，而不是每次操作都分配新的内存。

Example：

```
a = torch.rand(2, 3)
result = torch.empty(2, 3)  # 预先分配内存
before = id(result)
torch.add(a, a, out=result)  # 重用预分配的内存
print(result)
id(result) == before
```

Output:

```
tensor([[0.5254, 1.9610, 1.6119],
        [0.2004, 0.8637, 1.8431]])
True
```

即在算法运算方法中指定out参数为预先分配的变量。