> 授人以鱼不如授人以渔，原汁原味的知识才更富有精华，本文只是对张量基本操作知识的理解和学习笔记，看完之后，想要更深入理解，建议去 pytorch 官方网站，查阅相关函数和操作，英文版在这里，中文版在这里。本文的代码是在 `pytorch1.7` 版本上测试的，其他版本一般也没问题。

一，张量的基本操作

`Pytorch` 中，张量的操作分为**结构操作和数学运算**，其理解就如字面意思。结构操作就是改变张量本身的结构，数学运算就是对张量的元素值完成数学运算。

- 常使用的张量结构操作：维度变换（`tranpose`、`view` 等）、合并分割（`split`、`chunk`等）、索引切片（`index_select`、`gather` 等）。
- 常使用的张量数学运算：标量运算、向量运算、矩阵运算。

二，维度变换

2.1，squeeze vs unsqueeze 维度增减

- `squeeze()`：对 tensor 进行维度的压缩，去掉维数为 `1` 的维度。用法：`torch.squeeze(a)` 将 a 中所有为 1 的维度都删除，或者 `a.squeeze(1)` 是去掉 `a` 中指定的维数为 `1` 的维度。
- `unsqueeze()`：对数据维度进行扩充，给指定位置加上维数为 `1` 的维度。用法：`torch.unsqueeze(a, N)`，或者 `a.unsqueeze(N)`，在 `a` 中指定位置 `N` 加上一个维数为 `1` 的维度。

`squeeze` 用例程序如下：

```
a = torch.rand(1,1,3,3)
b = torch.squeeze(a)
c = a.squeeze(1)
print(b.shape)
print(c.shape)
```

程序输出结果如下：

```
torch.Size([3, 3])
torch.Size([1, 3, 3])
```

`unsqueeze` 用例程序如下：

```
x = torch.rand(3,3)
y1 = torch.unsqueeze(x, 0)
y2 = x.unsqueeze(0)
print(y1.shape)
print(y2.shape)
```

程序输出结果如下：

```
torch.Size([1, 3, 3])
torch.Size([1, 3, 3])
```

2.2，transpose vs permute 维度交换

`torch.transpose()` 只能交换两个维度，而 `.permute()` 可以自由交换任意位置。函数定义如下：

```
transpose(dim0, dim1) → Tensor  # See torch.transpose()
permute(*dims) → Tensor  # dim(int). Returns a view of the original tensor with its dimensions permuted.
```

在 CNN 模型中，我们经常遇到交换维度的问题，举例：四个维度表示的 tensor：[batch, channel, h, w]（nchw），如果想把 channel 放到最后去，形成 [batch, h, w, channel]（nhwc），如果使用 torch.transpose() 方法，至少要交换两次（先 1 3 交换再 1 2 交换），而使用 .permute() 方法只需一次操作，更加方便。例子程序如下：

```
import torch
input = torch.rand(1,3,28,32)                    # torch.Size([1, 3, 28, 32]
print(b.transpose(1, 3).shape)                   # torch.Size([1, 32, 28, 3])
print(b.transpose(1, 3).transpose(1, 2).shape)   # torch.Size([1, 28, 32, 3])

print(b.permute(0,2,3,1).shape)                  # torch.Size([1, 28, 28, 3]
```

三，索引切片

3.1，规则索引切片方式

张量的索引切片方式和 numpy、python 多维列表几乎一致，都可以通过索引和切片对部分元素进行修改。切片时支持缺省参数和省略号。实例代码如下：

```
>>> t = torch.randint(1,10,[3,3])
>>> t
tensor([[8, 2, 9],
        [2, 5, 9],
        [3, 9, 9]])
>>> t[0] # 第 1 行数据
tensor([8, 2, 9])
>>> t[2][2]
tensor(9)
>>> t[0:3,:]  # 第1至第3行，全部列
tensor([[8, 2, 9],
        [2, 5, 9],
        [3, 9, 9]])
>>> t[0:2,:]  # 第1行至第2行
tensor([[8, 2, 9],
        [2, 5, 9]])
>>> t[1:,-1]  # 第2行至最后行，最后一列
tensor([9, 9])
>>> t[1:,::2] # 第1行至最后行，第0列到最后一列每隔两列取一列
tensor([[2, 9],
        [3, 9]])
```

以上切片方式相对规则，对于不规则的切片提取,可以使用 torch.index_select，torch.take，torch.gather，torch.masked_select。

3.2，gather 和 torch.index_select 算子

> gather 算子的用法比较难以理解，在翻阅了官方文档和网上资料后，我有了一些自己的理解。

1，gather 是不规则的切片提取算子（Gathers values along an axis specified by dim. 在指定维度上根据索引 index 来选取数据）。函数定义如下：

```
torch.gather(input, dim, index, *, sparse_grad=False, out=None) → Tensor
```

参数解释：

- input (Tensor) – the source tensor.
- dim (int) – the axis along which to index.

- `index` (LongTensor) – the indices of elements to gather.

`gather` 算子的注意事项：

- 输入 `input` 和索引 `index` 具有相同数量的维度，即 `input.shape = index.shape`
- 对于任意维数，只要 `d != dim`，index.size(d) <= input.size(d)，即对于可以不用索引维数 `d` 上的全部数据。
- 输出 `out` 和 索引 `index` 具有相同的形状。输入和索引不会相互广播。

对于 3D tensor，`output` 值的定义如下：
`gather` 的官方定义如下：

```
out[i][j][k] = input[index[i][j][k]][j][k]  # if dim == 0
out[i][j][k] = input[i][index[i][j][k]][k]  # if dim == 1
out[i][j][k] = input[i][j][index[i][j][k]]  # if dim == 2
```

通过理解前面的一些定义，相信读者对 `gather` 算子的用法有了一个基本了解，下面再结合 2D 和 3D tensor 的用例来直观理解算子用法。
（1），对于 2D tensor 的例子：

```
>>> import torch
>>> a = torch.arange(0, 16).view(4,4)
>>> a
tensor([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11],
        [12, 13, 14, 15]])
>>> index = torch.tensor([[0, 1, 2, 3]])  # 选取对角线元素
>>> torch.gather(a, 0, index)
tensor([[ 0,  5, 10, 15]])
```

`output` 值定义如下：

```
# 按照 index = tensor([[0, 1, 2, 3]])顺序作用在行上索引依次为0,1,2,3
a[0][0] = 0
a[1][1] = 5
a[2][2] = 10
a[3][3] = 15
```

（2），索引更复杂的 2D tensor 例子：

```
>>> t = torch.tensor([[1, 2], [3, 4]])
>>> t
tensor([[1, 2],
        [3, 4]])
>>> torch.gather(t, 1, torch.tensor([[0, 0], [1, 0]]))
tensor([[ 1,  1],
        [ 4,  3]])
```

`output` 值的计算如下：

```
output[i][j] = input[i][index[i][j]]  # if dim = 1
output[0][0] = input[0][index[0][0]] = input[0][0] = 1
output[0][1] = input[0][index[0][1]] = input[0][0] = 1
output[1][0] = input[1][index[1][0]] = input[1][1] = 4
output[1][1] = input[1][index[1][1]] = input[1][0] = 3
```

总结：**可以看到 `gather` 是通过将索引在指定维度 `dim` 上的值替换为 `index` 的值，但是其他维度索引不变的情况下获取 `tensor` 数据。**直观上可以理解为对矩阵进行重排，比如对每一行(dim=1)的元素进行变换，比如 `torch.gather(a, 1, torch.tensor([[1,2,0], [1,2,0]]))` 的作用就是对 矩阵 `a` 每一行的元素，进行 `permtute(1,2,0)` 操作。
2，理解了 `gather` 再看 `index_select` 就很简单，函数作用是返回沿着输入张量的指定维度的指定索引号进行索引的张量子集。函数定义如下：

```
torch.index_select(input, dim, index, *, out=None) → Tensor
```

函数返回一个新的张量，它使用数据类型为 `LongTensor` 的 `index` 中的条目沿维度 `dim` 索引输入张量。返回的张量具有与原始张量（输入）相同的维数。 维度尺寸与索引长度相同； 其他尺寸与原始张量中的尺寸相同。
实例代码如下：

```
>>> x = torch.randn(3, 4)
>>> x
tensor([[ 0.1427,  0.0231, -0.5414, -1.0009],
        [-0.4664,  0.2647, -0.1228, -1.1068],
        [-1.1734, -0.6571,  0.7230, -0.6004]])
>>> indices = torch.tensor([0, 2])
>>> torch.index_select(x, 0, indices)
tensor([[ 0.1427,  0.0231, -0.5414, -1.0009],
        [-1.1734, -0.6571,  0.7230, -0.6004]])
>>> torch.index_select(x, 1, indices)
tensor([[ 0.1427, -0.5414],
        [-0.4664, -0.1228],
        [-1.1734,  0.7230]])
```

四，合并分割

4.1，torch.cat 和 torch.stack

可以用 `torch.cat` 方法和 `torch.stack` 方法将多个张量合并，也可以用 `torch.split` 方法把一个张量分割成多个张量。 `torch.cat` 和 `torch.stack` 有略微的区别，`torch.cat` 是连接，不会增加维度，而 `torch.stack` 是堆叠，会增加一个维度。两者函数定义如下：

```
# Concatenates the given sequence of seq tensors in the given dimension. All tensors
must either have the same shape (except in the concatenating dimension) or be empty.
torch.cat(tensors, dim=0, *, out=None) → Tensor
# Concatenates a sequence of tensors along **a new** dimension. All tensors need to be
of the same size.
torch.stack(tensors, dim=0, *, out=None) → Tensor
```

`torch.cat` 和 `torch.stack` 用法实例代码如下：

```
>>> a = torch.arange(0,9).view(3,3)
>>> b = torch.arange(10,19).view(3,3)
>>> c = torch.arange(20,29).view(3,3)
>>> cat_abc = torch.cat([a,b,c], dim=0)
>>> print(cat_abc.shape)
torch.Size([9, 3])
>>> print(cat_abc)
tensor([[ 0,  1,  2],
        [ 3,  4,  5],
        [ 6,  7,  8],
        [10, 11, 12],
        [13, 14, 15],
```

```
        [16, 17, 18],
        [20, 21, 22],
        [23, 24, 25],
        [26, 27, 28]])
>>> stack_abc = torch.stack([a,b,c], axis=0)  # torch中dim和axis参数名可以混用
>>> print(stack_abc.shape)
torch.Size([3, 3, 3])
>>> print(stack_abc)
tensor([[[ 0,  1,  2],
         [ 3,  4,  5],
         [ 6,  7,  8]],

        [[10, 11, 12],
         [13, 14, 15],
         [16, 17, 18]],

        [[20, 21, 22],
         [23, 24, 25],
         [26, 27, 28]]])
>>> chunk_abc = torch.chunk(cat_abc, 3, dim=0)
>>> chunk_abc
(tensor([[0, 1, 2],
         [3, 4, 5],
         [6, 7, 8]]),
 tensor([[10, 11, 12],
         [13, 14, 15],
         [16, 17, 18]]),
 tensor([[20, 21, 22],
         [23, 24, 25],
         [26, 27, 28]]))
```

4.2，torch.split 和 torch.chunk

`torch.split()` 和 `torch.chunk()` 可以看作是 `torch.cat()` 的逆运算。`split()` 作用是将张量拆分为多个块，每个块都是原始张量的视图。`split()` 函数定义如下：

```
"""
Splits the tensor into chunks. Each chunk is a view of the original tensor.
If split_size_or_sections is an integer type, then tensor will be split into equally
sized chunks (if possible). Last chunk will be smaller if the tensor size along the
given dimension dim is not divisible by split_size.
If split_size_or_sections is a list, then tensor will be split into
len(split_size_or_sections) chunks with sizes in dim according to
split_size_or_sections.
"""
torch.split(tensor, split_size_or_sections, dim=0)
```

`chunk()` 作用是将 `tensor` 按 `dim`（行或列）分割成 `chunks` 个 `tensor` 块，返回的是一个元组。`chunk()` 函数定义如下：

```
torch.chunk(input, chunks, dim=0) → List of Tensors
"""
Splits a tensor into a specific number of chunks. Each chunk is a view of the input
tensor.
Last chunk will be smaller if the tensor size along the given dimension dim is not
divisible by chunks.
Parameters:
    input (Tensor) – the tensor to split
    chunks (int) – number of chunks to return
    dim (int) – dimension along which to split the tensor
"""
```

实例代码如下：

```
>>> a = torch.arange(10).reshape(5,2)
>>> a
tensor([[0, 1],
        [2, 3],
        [4, 5],
        [6, 7],
        [8, 9]])
>>> torch.split(a, 2)
(tensor([[0, 1],
         [2, 3]]),
 tensor([[4, 5],
         [6, 7]]),
 tensor([[8, 9]]))
>>> torch.split(a, [1,4])
(tensor([[0, 1]]),
 tensor([[2, 3],
         [4, 5],
         [6, 7],
         [8, 9]]))
>>> torch.chunk(a, 2, dim=1)
(tensor([[0],
        [2],
        [4],
        [6],
        [8]]),
tensor([[1],
        [3],
        [5],
        [7],
        [9]]))
```

五，卷积相关算子

5.1，上采样方法总结

上采样大致被总结成了三个类别：

1. 基于线性插值的上采样：最近邻算法（`nearest`）、双线性插值算法（`bilinear`）、双三次插值算法（`bicubic`）等，这是传统图像处理方法。
2. 基于深度学习的上采样（转置卷积，也叫反卷积`Conv2dTranspose2d`等）
3. `Unpooling`的方法（简单的补零或者扩充操作）

> 计算效果：最近邻插值算法 < 双线性插值 < 双三次插值。计算速度：最近邻插值算法 > 双线性插值 > 双三次插值。

5.2，F.interpolate 采样函数

> Pytorch 老版本有 `nn.Upsample` 函数，新版本建议用 `torch.nn.functional.interpolate`，一个函数可实现定制化需求的上采样或者下采样功能，。

`F.interpolate()` 函数全称是 `torch.nn.functional.interpolate()`，函数定义如下：

```
def interpolate(input, size=None, scale_factor=None, mode='nearest',
align_corners=None, recompute_scale_factor=None):  # noqa: F811
    # type: (Tensor, Optional[int], Optional[List[float]], str, Optional[bool],
Optional[bool]) -> Tensor
    pass
```

参数解释如下：

- `input`(Tensor)：输入张量数据；
- `size`： 输出的尺寸，数据类型为 tuple： ([optional D_out], [optional H_out], W_out)，和 `scale_factor` 二选一。
- `scale_factor`：在高度、宽度和深度上面的放大倍数。数据类型既可以是 int`——表明高度、宽度、深度都扩大同一倍数；也可是`tuple`——指定高度、宽度、深度等维度的扩大倍数。
- `mode`： 上采样的方法，包括最近邻（`nearest`），线性插值（`linear`），双线性插值（`bilinear`），三次线性插值（`trilinear`），默认是最近邻（`nearest`）。
- `align_corners`： 如果设为 `True`，输入图像和输出图像角点的像素将会被对齐（aligned），这只在 `mode = linear, bilinear, or trilinear` 才有效，默认为 `False`。

例子程序如下：

```
import torch.nn.functional as F
x = torch.rand(1,3,224,224)
y = F.interpolate(x * 2, scale_factor=(2, 2), mode='bilinear').squeeze(0)
print(y.shape)   # torch.Size([3, 224, 224)
```

5.3，nn.ConvTranspose2d 反卷积

转置卷积（有时候也称为反卷积，个人觉得这种叫法不是很规范），它是一种特殊的卷积，先 `padding` 来扩大图像尺寸，紧接着跟正向卷积一样，旋转卷积核 180 度，再进行卷积计算。

参考资料

- pytorch演示卷积和反卷积运算
- torch.Tensor
- PyTorch学习笔记(10)——上采样和PixelShuffle
- 反卷积 Transposed convolution
- PyTorch中的转置卷积详解——全网最细
- 4-1,张量的结构操作