Tensor Manipulation

Hyehyun Kwon, Dongmyeong Lee

Contents

- Create tensor
  - -Tensor data-type
  - -Precision of floating-point
- Broadcasting
- Calculation
  - -Matrix multiplication
- Reshaping tensor
  - -View & squeeze & unsqueeze

Tensor data-type

- A majority of programing framework including Pytorch roughly provides 3 kinds of data-type, which are boolean, integer, floating point. These are divided by logical criteria.

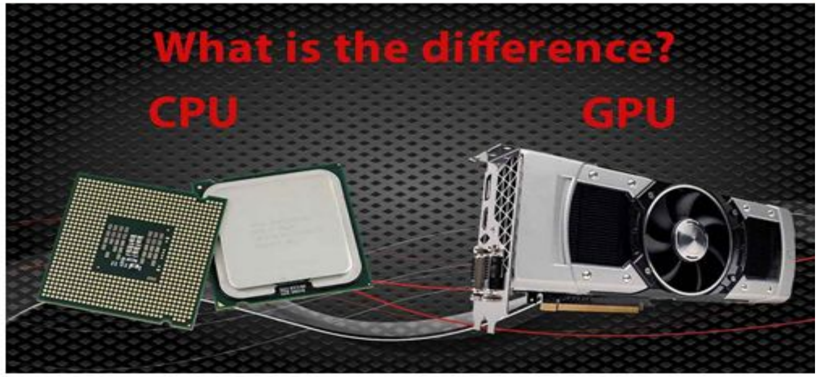
Logical

Physical

	Data type	dtype	CPU Tensor CPU tensor	GPU Tensor GPU tensor
Floating-point	32-bit floating point	torch.float32 or torch.float	torch.FloatTensor	torch.cuda.FloatTensor
	64-bit floating point	torch.float64 or torch.double	torch.DoubleTensor	torch.cuda.DoubleTensor
	16-bit floating point	torch.float16 or torch.half	torch.HalfTensor	torch.cuda.HalfTensor
Integer	8-bit integer (unsigned)	torch.uint8	torch.ByteTensor	torch.cuda.ByteTensor
	8-bit integer (signed)	torch.int8	torch.CharTensor	torch.cuda.CharTensor
	16-bit integer (signed)	torch.int16 or torch.short	torch.ShortTensor	torch.cuda.ShortTensor
	32-bit integer (signed)	torch.int32 or torch.int	torch.IntTensor	torch.cuda.IntTensor
	64-bit integer (signed)	torch.int64 or torch.long	torch.LongTensor	torch.cuda.LongTensor
Boolean	Boolean	torch.bool	torch.BoolTensor	torch.cuda.BoolTensor

Transfer CPU tensors into GPU tensors

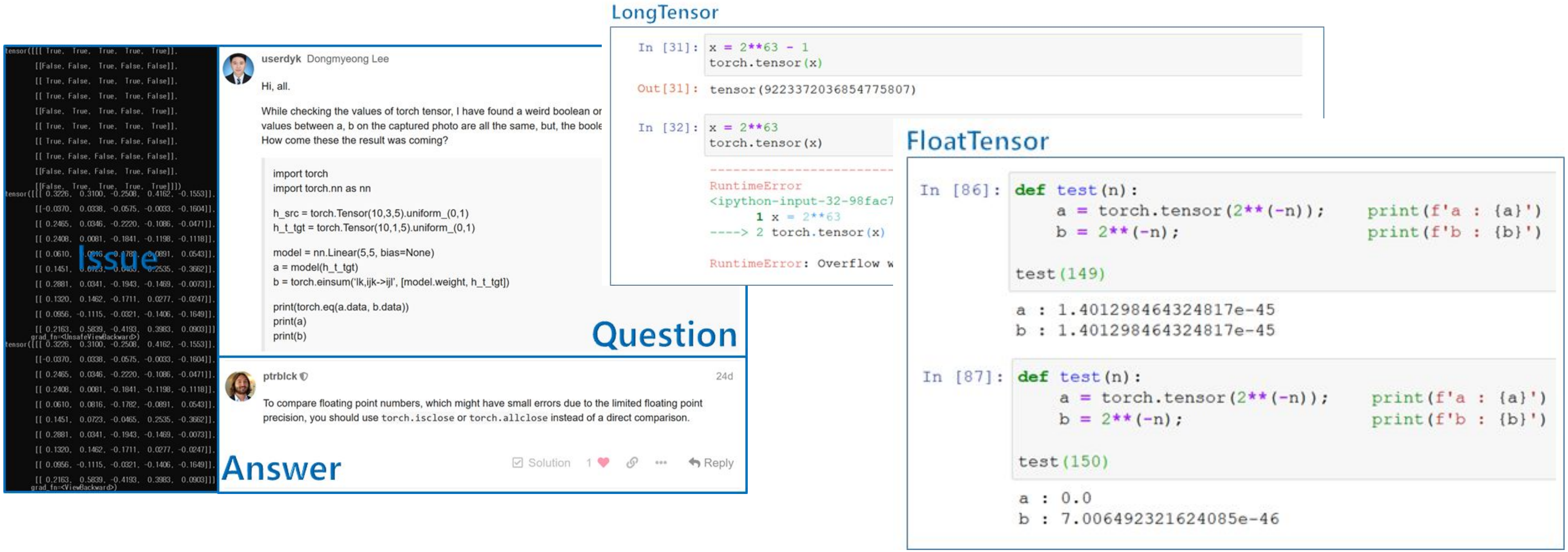
```
import torch

device = 'cuda' if  torch.cuda.is_available() else 'cpu'
x = torch.Tensor(3,3).uniform_(0,1)

print(x.type())              # torch.FloatTensor
print(x.to(device).type())     # torch.cuda.FloatTensor
```



GPU increased parallel process of operations as it has more Arithmetic logic units (ALU) compare to typical CPU

Figure: CPU vs GPU

Precision of floating-point(1)

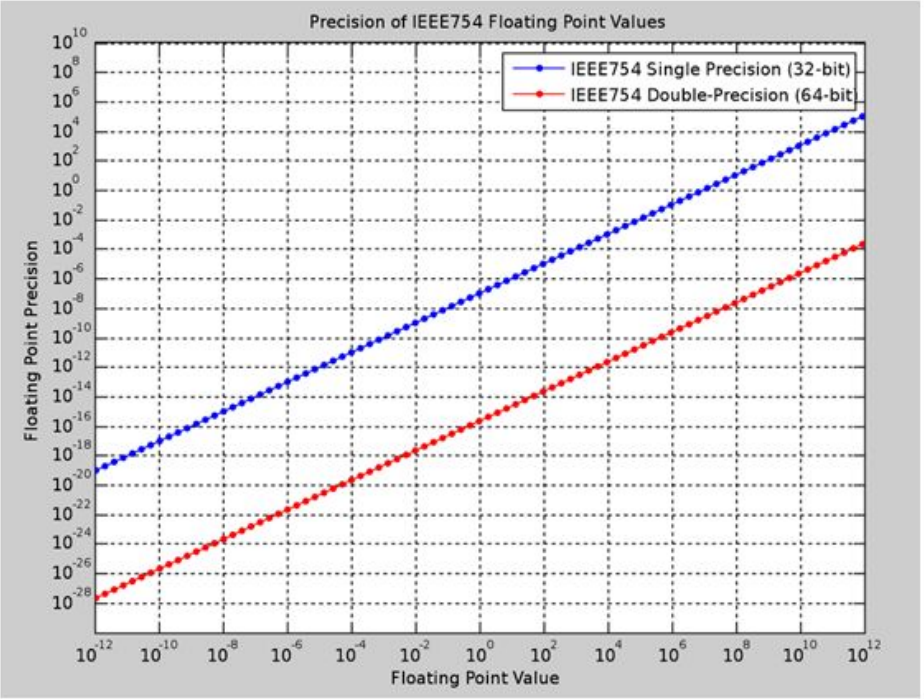


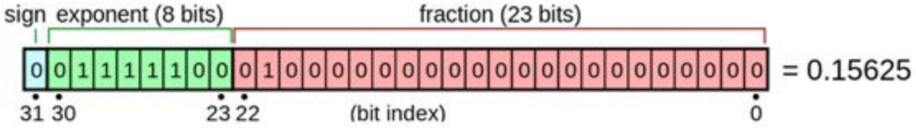
- The IEEE Standard for **Floating-Point Arithmetic** (**IEEE 754**) is a technical standard for floating-point arithmetic established in 1985 by the Institute of Electrical and Electronics Engineers.
  - Floating points calculation demands attention.
  - Even though mathematical expressions is equivalent, value of results could be different.

https://discuss.pytorch.org/t/comparsion-through-torch-eq/85385

Precision of floating-point(2)

Precision





Being against the completeness of the real numbers

$$(-1)^{b_{31}} \times 2^{(b_{30}b_{29}\dots b_{23})_2 - 127} \times (1.b_{22}b_{21}\dots b_0)_2$$

$$\text{value} = (-1)^{\text{sign}} \times 2^{(e-127)} \times \left(1 + \sum_{i=1}^{23} b_{23-i}2^{-i}\right)$$

32-bit floating-point

- **Sign bit**: 1 bit
- **Exponent width**: 8 bits
- **Significant precision**: 24 bits (23 explicitly stored)

64-bit floating-point

https://en.wikipedia.org/wiki/Single-precision_floating-point_format, https://en.wikipedia.org/wiki/Double-precision_floating-point_format

Broad Casting

The two matrices can be operated even when necessary rules are not established.

Broad Casting

The two matrices can be operated even when necessary rules are not established.

necessary rules for operating two matrices?

addition : same dimension

multiply : the number of columns in front has to equal the number of rows in back

Broad Casting

The two matrices can be operated even when necessary rules are *not established*.

-> dimension changes as needed

Broad Casting

The two matrices can be operated even when necessary rules are not established.

-> dimension changes as needed

HOW

```
m1 = torch.FloatTensor([[1,2]])
m2 = torch.FloatTensor([3])
```

1x1 -> 1x2

Broad Casting

The two matrices can be operated even when necessary rules are not established.

-> dimension changes as needed

PROBLEM

error of matrix multiplication

Matrix Multiplication(Linear Algebra)

First step : operation rule

$\vec{a}_1 = (a_{11}, a_{12}, \dots, a_{1n})$ $\vec{b}_1 = (b_{11}, b_{21}, \dots, b_{n1})$

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{pmatrix}$$

Gradient Batch

Matrix multiplication(C = AB ≠BA)

dot product $(\vec{a}_1 \cdot \vec{b}_1)$

$$\mathbf{C} = \begin{pmatrix} \boxed{a_{11}b_{11} + \cdots + a_{1n}b_{n1}} & a_{11}b_{12} + \cdots + a_{1n}b_{n2} & \cdots & a_{11}b_{1p} + \cdots + a_{1n}b_{np} \\ a_{21}b_{11} + \cdots + a_{2n}b_{n1} & a_{21}b_{12} + \cdots + a_{2n}b_{n2} & \cdots & a_{21}b_{1p} + \cdots + a_{2n}b_{np} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}b_{11} + \cdots + a_{mn}b_{n1} & a_{m1}b_{12} + \cdots + a_{mn}b_{n2} & \cdots & a_{m1}b_{1p} + \cdots + a_{mn}b_{np} \end{pmatrix}$$

Second step : linear mapping

Lecture 3 : Gradient Input

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \qquad \mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}$$

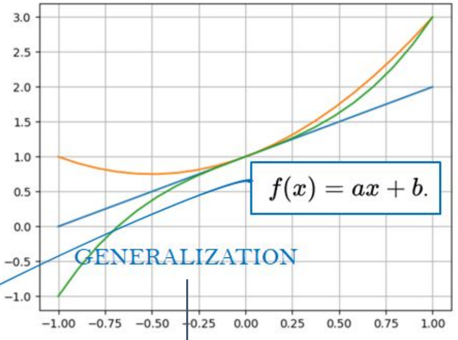
Matrix multiplication(y = Ax)

Linear mapping : $\mathbf{y} = A(\mathbf{x}) = \begin{pmatrix} a_{11}x_1 + \cdots + a_{1n}x_n \\ a_{21}x_1 + \cdots + a_{2n}x_n \\ \vdots \\ a_{m1}x_1 + \cdots + a_{mn}x_n \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix}$

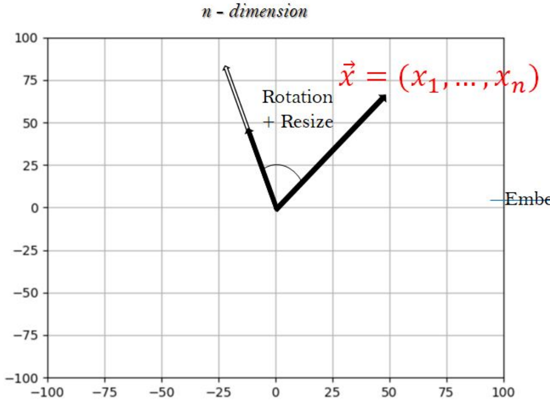
Gradient * Input Bias

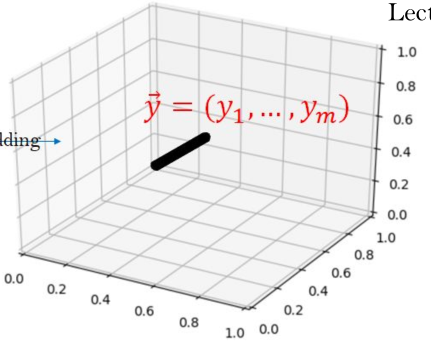
Lecture 4 : Multi-variate Regression

Lecture 2 : Uni-variate(Simple) Regression



Third step : vector analysis





Statistics, Machine Learning, Deep Learning, Reinforcement Learning, …
Essentials for handling Big Data

https://en.wikipedia.org/wiki/Matrix_multiplication

Matrix Multiplication

solving problem of broadcasting when multiply two different shape of matrices

Matrix Multiplication

solving problem of broadcasting when multiply two different shape of matrices

1. matrix.matmul()
2. torch.mm()

```
m1 = torch.FloatTensor([[1,2], [3,4]])
m2 = torch.FloatTensor([[1], [2]])
m3 = torch.mm(m1, m2)
print(m3)
```

```
tensor([[ 5.],
        [11.]])
```

solving problem of broadcasting when multiply two different shape of tensors

multiply tensors which are **more than three dimensions**

```
x1 = torch.FloatTensor([
    [[1,2,3],[4,5,6]],
    [[1,2,3],[4,5,6]],
])
x2 = torch.FloatTensor([
    [[1,2,3],[4,5,6],[7,8,9]],
    [[1,2,3],[4,5,6],[7,8,9]],
])
x3 = x1.matmul(x2) #==torch.matmul(x1, x2)
print(x3)
print("x1 shape", x1.shape)
print("x2 shape", x2.shape)
```

```
tensor([[[30., 36., 42.],
         [66., 81., 96.]],

        [[30., 36., 42.],
         [66., 81., 96.]]])
x1 shape torch.Size([2, 2, 3])
x2 shape torch.Size([2, 3, 3])
```

solving problem of broadcasting when multiply two different shape of tensors

multiply tensors which are **more than three dimensions**

```
A = torch.randn([5, 4, 10, 2, 3])
B = torch.randn([5, 4, 10, 3, 7])
C = torch.matmul(A, B)
print(C)
```

```
[[[ 1.1698e+00, -2.4340e+00, -1.2365e-01,  ..., -4.7121e-01,
   -1.8920e+00, -2.8990e+00],
  [ 5.9485e-01, -1.9962e+00,  6.6300e-02,  ..., -9.6229e-01,
   -4.9463e-01, -2.6750e+00]],

 [[-1.8881e+00, -2.8208e+00,  2.0184e+00,  ...,  4.1024e-01,
   -3.7134e-01,  1.6824e+00],
  [-2.1061e+00, -5.0559e+00,  3.1637e+00,  ...,  5.1328e-01,
   -1.8898e+00,  2.2678e+00]],

 [[-2.1815e-01,  3.3097e-01, -1.1918e-01,  ...,  1.1221e-01,
   -5.0288e-01,  7.1869e-01],
  [ 2.8319e+00, -2.2120e+00,  3.8287e-01,  ...,  8.3422e-01,
    1.7874e-01,  6.4380e-01]],
```

Matrix Multiplication

solving problem of broadcasting when multiply two different shape of tensors

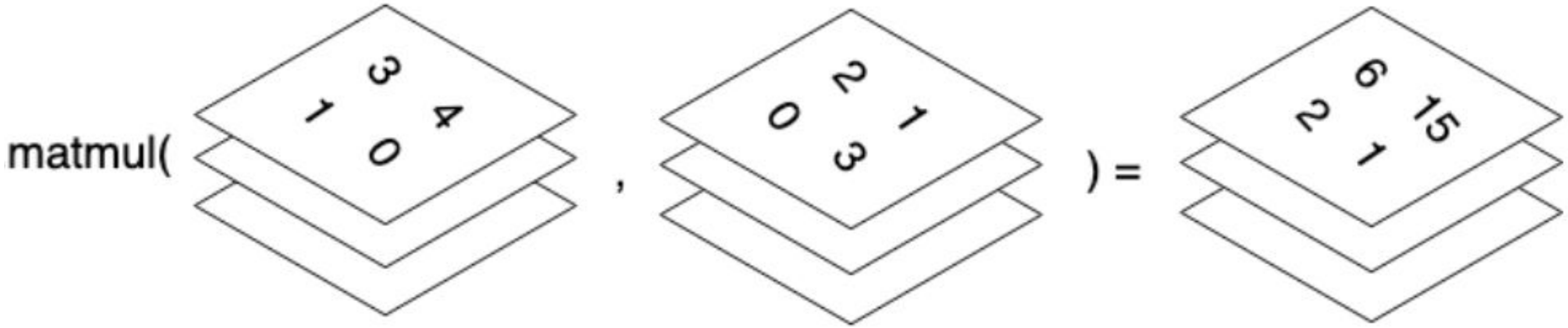
multiply tensors which are **three dimensions**

```
x1 = torch.FloatTensor([
    [[1,2,3],[4,5,6]],
    [[1,2,3],[4,5,6]],
])
x2 = torch.FloatTensor([
    [[1,2,3],[4,5,6],[7,8,9]],
    [[1,2,3],[4,5,6],[7,8,9]],
])
x3 = torch.bmm(x1,x2)
print(x3)
print("x1 shape", x1.shape)
print("x2 shape", x2.shape)
```

```
tensor([[[30., 36., 42.],
         [66., 81., 96.]],

        [[30., 36., 42.],
         [66., 81., 96.]]])
x1 shape torch.Size([2, 2, 3])
x2 shape torch.Size([2, 3, 3])
```

solving problem of broadcasting when multiply two different shape of tensors

multiply tensors which are **three dimensions**



https://bowbowbow.tistory.com/27

reshaping tensors

important to change shapes as needed

NOT REALLY changing the shape if there is no declaration of variable

reshaping tensors

diminishing dimension when dimension has only one element

```
x1 = torch.FloatTensor(10,1,3,1,4)
x2 = torch.squeeze(x1,1)
print(x1.size(),x2.size())
```

```
torch.Size([10, 1, 3, 1, 4]) torch.Size([10, 3, 1, 4])
```

can set which one to delete

View & Squeeze & Unsqueeze

reshaping tensors

diminishing dimension when dimension has only one element

adding new dimension

```
x1 = torch.FloatTensor(10,3,4)
x2 = torch.unsqueeze(x1, dim=1)
print(x1.size(),x2.size()) # torch.Size([10, 3, 4]) torch.Size([1, 10, 3, 4])
x3 = torch.unsqueeze(x1, dim=3)
print(x1.size(),x3.size()) # torch.Size([10, 3, 4]) torch.Size([10, 1, 3, 4])
```

```
torch.Size([10, 3, 4]) torch.Size([10, 1, 3, 4])
torch.Size([10, 3, 4]) torch.Size([10, 3, 4, 1])
```